Technische Universität Ilmenau
Fakultät für Automatisierung und Informatik
Fachgebiet Prozessinformatik

Studienarbeit Sommersemester 2003

zum Thema

Reverse Engineering von Entwurfsmustern

Jens Herbig
28806

7.Juli 2003

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Diese Arbeit beschäftigt sich mit dem Reverse Engineering von Entwurfsmustern.
Als Grundlage wird das CASE-Tool Together verwendet. Das Augenmerk soll vor allem auf der Entwicklung von einem Modul liegen, welches das Finden von Entwurfsmustern ermöglicht. Es werden die Probleme bei der Entwicklung der Algorithmen in Together aufgezeigt und Lösungsvorschläge dargestellt.
Das Vorgehen soll anhand von Beispielen erläutert werden. Hierzu sind die Entwurfsmuster Fabrikmethode, abstrakte Fabrik und Beobachter implementiert worden, deren Suchalgorithmen in der Diplomarbeit von Herrn Sebastian Naumann [Nau01] vorgestellt wurden.
Zum Testen des Modules wurde gleichzeitig ein kleines Beispielprojekt angelegt, welches die implementierten Muster enthält. In Abbildung 1 ist das Klassendiagramm dieses Projektes dargestellt.
Die Erläuterungen der einzelnen Muster befindet sich in Abschnitt 3.
Die 10 Klassen haben die folgende Bedeutung:

Data Die Klasse Data bietet eine Schnittstelle für allgemeine Produktdaten. Es werden die Variablen *Menge, Einzelpreis* und *Artikeltext* inklusive der *setter* und *getter* – Methoden zur Verfügung gestellt.

Produktliste Hier wird eine Liste von Data–Objekten verwaltet. Die Klasse wurde als *Singleton* implementiert, da es nur eine Produktliste gebe darf. Weiterhin stellt die Klasse ein konkretes Subjekt des *Observer*–Musters dar. Hier müssen sich alle Ansichten, welche die Produktliste darstellen wollen, anmelden und werden bei Änderung entsprechend des *Observer*–Musters aktualisiert.

Subjekt Bereitgestellt wird in dieser Klasse das Interface des abstrakten Subjektes des *Observer*–Musters. Hier sind die notwendigen Methoden (*attach, detach* und *inform*) definiert.

Observer Den abstrakten Beobachter stellt diese Klasse dar. Das Interface definiert die *update* Methode, welche die abgeleiteten Klassen überschreiben müssen. Bei Änderung der Daten dient sie der Anzeigenaktualisierung.

PrintProdukte Abgeleitet von der Klasse *Observer* wird der konkrete Beobachter dargestellt. Er soll den Inhalt der Produktliste auf dem Bildschirm unter Zuhilfenahme der Funktion *update* aktualisieren.

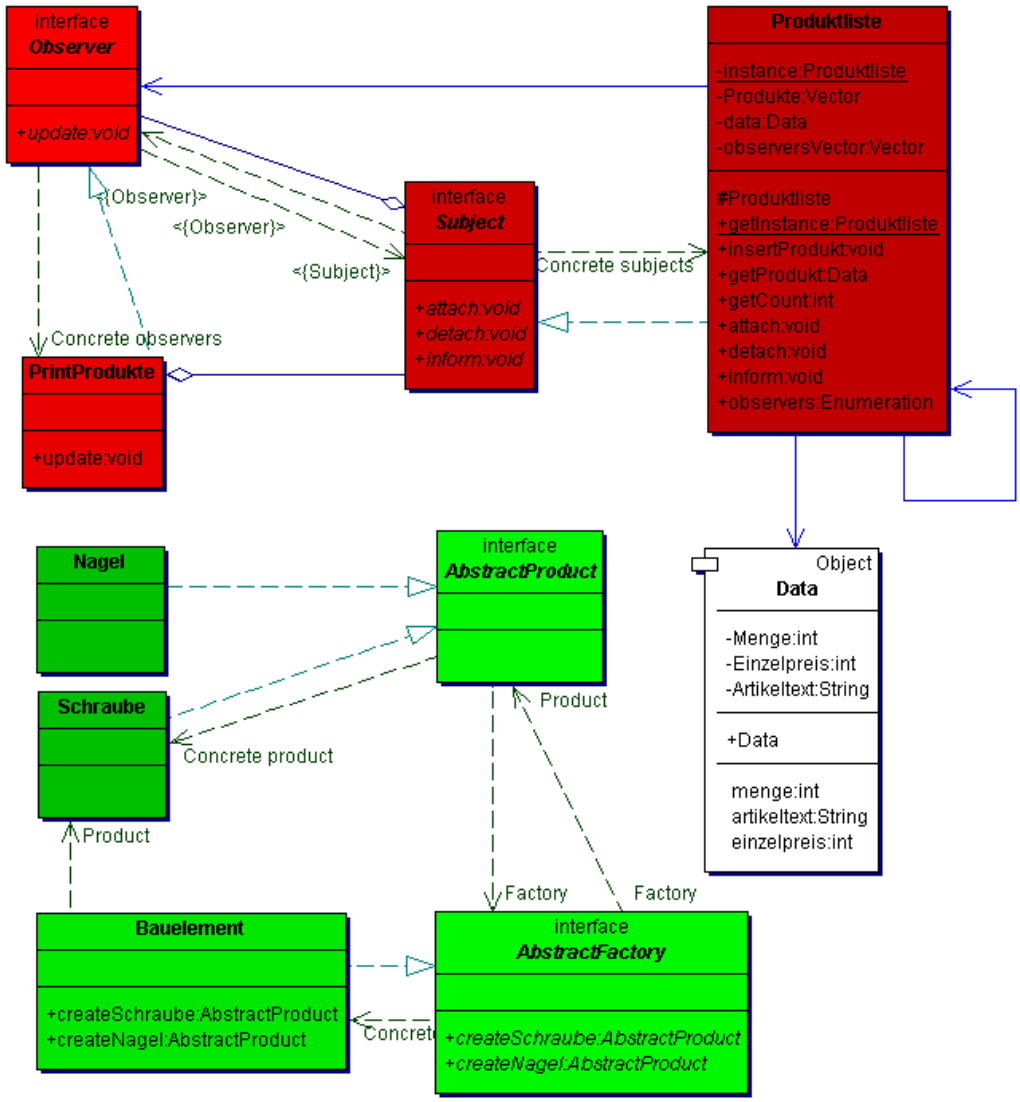

Abbildung 1: das Beispiel – Projekt

AbstractProduct Die Klasse stellt das abstrakte Produkt des Musters *abstrakte Fabrik* dar.

Nagel und Schraube Beide sind von der abstrakten Produktklasse abgeleitet und soll zur Erstellung der beiden verschiedenen Produkte dienen.

AbstractFactory Das Interface definiert die "Grundmethoden" zur Erstellung eines Nagels und einer Schraube. Diese beiden Methoden sind als *Fabrikmethodenmuster* implementiert.

Bauelement Die konkrete Fabrik definiert die konkrete Implementierung der beiden Fabrikmethoden, welche vom implementierten Interface bereitgestellt wurden.

Nun folgt eine Übersicht, welche jede Klasse und das Muster, zu welchem die Klasse gehört, auflistet. In der dritten Spalte werden die Positionen der innerhalb des Musters deutlich gemacht.
Um zu testen, ob das Modul Mustererkennung ohne Fehler funktioniert, ist es nur nötig, die Tabelle mit der Ausgabe des Modules zu vergleichen. Stimmen die Angaben nicht überein, so ist ein Muster nicht ordnungsgemäß erkannt wurden.
Ausnahme ist das Muster der Fabrikmethode. Da zur Erkennung einer *abstrakten Fabrik* das Muster der *Fabrikmethode* zwingend notwendig ist, werden Fabrikmethoden, welche zu einer Fabrik gehören, nicht noch einmal extra aufgelistet.

Klasse	**Muster**	**Position im Muster**
Data	–	–
Produktliste	Singleton, Observer	Singleton (Singleton), Konkretes Subjekt (Observer)
Subject	Observer	Abstraktes Subjekt
Observer	Observer	Abstrakter Observer
PrintProdukte	Observer	Konkreter Observer
AbstractProduct	Abstrakte Fabrik	Abstraktes Produkt
Nagel	Abstrakte Fabrik	Konkretes Produkt
Schraube	Abstrakte Fabrik	Konkretes Produkt
AbstractFactory	Abstrakte Fabrik, Fabrikmethode	Abstrakte Fabrik (Abstrakte Fabrik), Abstrakter Erzeuger (Fabrikmethode)
Bauelement	Abstrakte Fabrik, Fabrikmethode	Konkrete Fabrik (Abstrakte Fabrik), Konkreter Erzeuger (Fabrikmethode)

2 Das Tool Together

In diesem Abschnitt soll ein kurzer Einblick in das Case–Tool Together gegeben werden. Weiterhin wird auf die *Open–API* und ihre Besonderheiten eingegangen und am Ende dieses Abschnittes kurz die allgemeine Modulentwicklung in Together vorgestellt.

2.1 Eine kurze Übersicht

Hier wird kurz auf den Funktionsumfang von Together eingegangen. Tiefgründigere Informationen können im Handbuch oder in der Hauptseminararbeit von Herrn Jens Klappstein [Klapp02] nachgelesen werden. Es sollen nur wichtige und für das Auffinden von Mustern notwendige Funktionen vorgestellt werden.
Together ist ein Case–Tool der Firma Togethersoft. Es soll den Entwicklern von Software in allen Bereichen unterstützen. Hierzu befindet sich eine integrierte Entwicklungsumgebung (IDE) im Funktionsumfang des Case–Tools. Diese dient der Erstellung, Kompilierung und dem Debuggen von Programmen. Unterstützt werden die folgenden Programmiersprachen:

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

Weiterhin bietet Together eine umfangreiche Unterstützung von UML. Es können UML–Diagramme aus dem Quellcode erzeugt und ständig synchron gehalten werden. Dem Entwickler wird es außerdem ermöglicht, alle Stellen zu finden, wo eine bestimmte Methode benutzt wird. Dies stellt vor allem beim Testen, ob eine Methode überhaupt verwendet wird, eine einfache und schnelle Methode dar.
Da es in dieser Arbeit um das Reverse Engineering geht, wird auch das von Together unterstützt. Hierzu gibt man Together den Pfad vor, in dem die Quelldateien liegen. Anschließend liest Together alle Dateien ein (inclusive aller Unterverzeichnisse, welche dann als Pakete dargestellt werden) und erstellt ein *default*–Klassendiagramm.
Wie man bei der kleinen Auswahl an Funktionalitäten schon sehen kann, ist Together sehr mächtig, was sich im Preis von 5000 - 13000 Euro pro Lizenz auch wiederspiegelt.

2.2 Together‘s Open API

Die Open API von Together basiert auf Java. Das Programm selbst bringt eine umfangreiche, aber meines Erachtens nach unübersichtliche und nicht ganz aktuelle Dokumentation mit. Vor allem ist das Prinzip sehr schwer zu verstehen, nach welchem die Open API die Elemente anspricht. Hier fehlt ein Diagramm, welches dies verdeutlicht. Die API teilt sich in sieben Teilbereiche. Für das Modul zur Mustererkennung sind allerdings bisher nur drei notwendig. Diese Bereiche werden im Folgenden kurz erläutert:

IDE Sie bietet den Zugriff auf die komplette IDE Funktionalität. Hier kann man auf verschiedeneste sichtbare Elemente zugreifen, die dem Nutzer z.B. Nachrichten übermitteln. Außerdem kann man auch auf die Diagramme zugreifen und sich einen Zeiger auf diese holen.

RWI Dieses Interface ist das Read Write Interface. Es ist möglich, auf Elemente innerhalb der Diagramme zugreifen und sie zu verändern, was allerdings auch eine Codeänderung nach sich zieht. In den meisten Fällen ist es möglich, eine Aufzählung der Elemente zu bekommen. Dabei werden gleichartige Objekte als eine Liste zurückgegeben. Diese kann man dann durchlaufen, um alle Elemente entsprechend zu bearbeiten.

SCI Das Source Code Interface ist wohl das mächtigste, aber auch umfangreichste der Interfaces. Mit Hilfe dieses kann man den kompletten Quellcode des geöffneten Projektes durchgehen, jedes einzelne Statement abfragen und auch Änderungen im Code vornehmen. Allerdings muss man auch bedenken, dass dieses Interface programmiersprachenabhängig ist. Vor allem beim Schreiben des Modules ist es notwendig, das zu beachten. Insbesondere die Fehlerbehandlung sollte vorher genau durchdacht werden z. B. mit der Frage: ”Was können für Probleme in welcher Sprache auftreten”. Ein Beispiel hierfür ist die Definition einer virtuellen Funktion. In Java sind grundsätzlich alle Funktionen virtuell,es sei denn, sie werden als final definiert. In C++ muss die Deklaration auf *virtual* explizit angegeben werden.
Auf das SCI – Interface sollte, aufgrund der Komplexität, nur in schwierigen Fällen zugegriffen werden. Leider ist es aber oft unumgänglich.

2.3 Modulentwicklung in Together

Zur Mustersuche in Together ist es notwendig, ein Modul zu erstellen. Nach der Einbindung in Together steht es jedem Nutzer zur Verfügung.
Ein Modul ist eine Java-Klasse, welche von der OpenAPI Klasse *IdeScript* abgeleitet wird. Um ein funktionsfähiges Modul zu erhalten, muss dann noch die geerbte Methode *run* überschrieben werden. Dies ist erforderlich, da nach Start des Modules diese

Methode ausgeführt wird. Ihr wird als Parameter ein IdeContext - Objekt übergeben. Darin sind alle markierten Elemente von dem aktuellen Klassendiagramm enthalten.
In dem erstellten Modul wurde dieses Objekt allerdings nicht weiter genutzt. Da das Modul für das Reverse Engineering von Entwurfsmuster gedacht ist, werden beim Start alle im Projekt befindlichen Klassen durchsucht, um sämtliche implementierten Muster zu finden. Um das zu ermöglichen, wird zu Beginn ein Zeiger auf das Modell mit Hilfe des Objektes *RwiModelAccess* geholt. Anschließend ist es nötig, alle *root – Pakete* zu bestimmen. Dies ist mit der Methode *RwiModel.rootPackages* möglich. Mittels dieser Aufzählung von Paketen kann man nun alle Unterpakete und Elemente, die in den Paketen enthalten sind, nacheinander durchsuchen. Gleichzeitig wird für jedes spezielle Element eine eigene Methode aufgerufen, in welcher die gesonderte Behandlung erfolgt.
Da jedes Paket wiederum *Unterpakete* enthalten kann, werden diese dann rekursiv durchsucht (als root Pakete). In einem Paket gibt es Nodes (Knoten). Das können alle Elemente sein, wie z.B. Klassen oder Interfaces. Die Knoten werden dann wieder jeder für sich behandelt.
Der genutzte Aufbau des Auffindens aller Elemente innerhalb des Projektes hat den Vorteil, dass man das Modul relativ schnell um andere Aufgaben erweitern kann. So wäre es zum Beispiel möglich, eine Methode zu schreiben, welche alle bisher gelesenen Pakete in einer bestimmten Farbe darstellt. Zur Umsetzung ist nur der Aufruf einer Methode, die eine übergebene Klasse färbt, in der Methode zur Paketbearbeitung notwendig. So werden alle Klassen erreicht und gefärbt.
An dieser Stelle soll nun kurz der Zusammenhang der Klassen innerhalb der Klassendiagramme erläutert werden. Hierzu sei zuvor noch kurz erwähnt, dass die folgenden Eigenschaften der Objekte (vom Typ RwiElement) extrem wichtig sind, um verschiedene Eigenschaften des Elementes zu bestimmen. Zum einen gibt es zu dem Objekt RwiProperty die verschiedensten Eigenschaften, die ein Element haben kann. Um auf eine der Eigenschaften zuzugreifen, wird der Name in Großbuchstaben an das Objekt angefügt. (z.B. RwiProperty.NAME, um den Namen des Elementes zu identifizieren)
Die andere Eigenschaft ist das RwiShapeTyp Objekt. Mit Hilfe dieses kann man beispielsweise mit folgendem Ausdruck bestimmen, ob es sich um eine Klasse handelt.

```
if ( RwiShapeTyp.CLASS.equals(
        element.getProperty(RwiProperty.SHAPE_TYP)))
```

Der dargestellte, recht umfangreiche Ausdruck wird häufig benötigt, da man wichtige Abfragen bzgl. des Elementtyps tätigen muss. Für Informationen weiterer Eigenschaften soll hier auf die OpenAPI Dokumentation verwiesen werden [OpenAPI].
In der Abbildung 2 werden kurz die wichtigsten Funktionen der OpenAPI zum Durchsuchen einer Klasse angegeben. Vergleich hierzu [Klapp02]

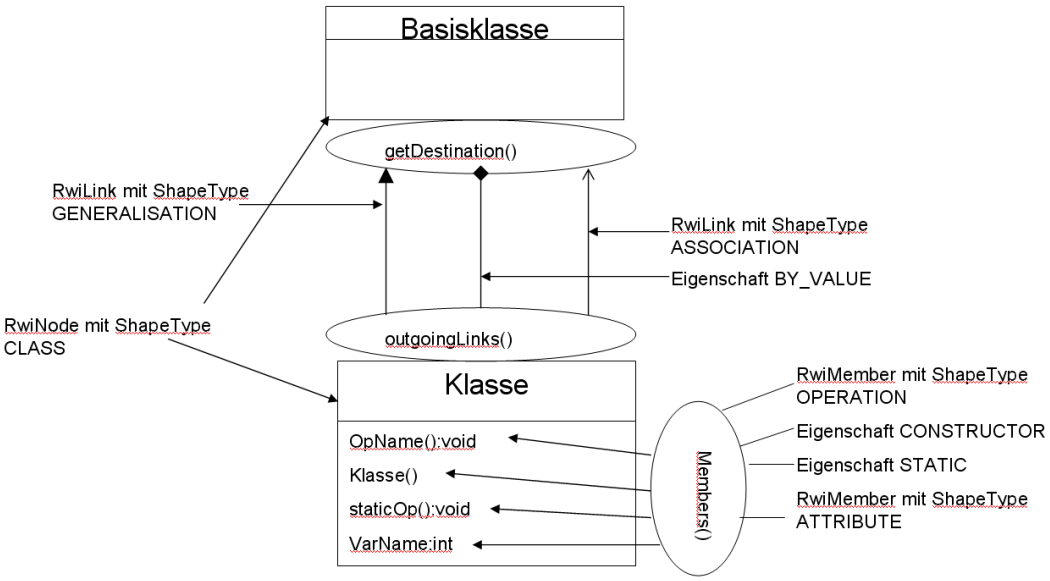

Abbildung 2: openAPI Funktionen zum Durchsuchen einer Klasse

Da das Modul ein einfaches Java Programm darstellt ist es auch möglich, jedes implementierte Muster in eine eigene Klasse auszulagern. Es wird nur eine "Startklasse" benötigt, die, wie schon angesprochen, das Interface *IdeScript* implementiert und die *run*–Methode überschreibt. Innerhalb dieser können dann die jeweiligen Instanzen der Mustererkennungsklassen erstellt und die Erkennung angestoßen werden. Somit ist eine Vielzahl von verschiedenen Implementierungsvarianten für das Modul Mustererkunng denkbar.
Im weiteren Verlauf dieser Arbeit soll es nun um das entwickelte Modul Mustererkennung gehen. Dabei wird zu Beginn jede Funktion ausführlich dargestellt. Im Anschluss daran werden Probleme, welche während der Implementierung aufgetreten sind, kurz dargestellt und Lösungsvorschläge gegeben.

3 Das Modul Mustererkennung

Um das entwickelte Modul Mustererkennung erfolgreich in Together einbinden zu können, sind folgende Schritte notwendig:
Das Verzeichnis Mustererkennung, das die Dateien `manifest.mf` und `Mustererkennung.class` enthält, muss in das Verzeichnis

`\$Together\modules\com\togethersoft\modules\`

kopiert werden. Hierbei stellt `\$Together` das Homeverzeichnis des Tools dar. Die Datei `manifest.mf` hat die Aufgabe, das Modul im Together in das richtige Untermenu anzuordnen und bestimmt die Lage der `Mustererkennung.class` Datei. Der Inhalt der Datei `manifest.mf` lautet folgendermaßen:

```
Name: Muster (Java)
Time: User
Main-Class: com.togethersoft.modules.Mustererkennung.Mustererkennung
Folder: "own/Mustererkennung"
```

Hierbei haben die Werte die folgende Bedeutungen:

Name: unter welchem das Modul im Together erscheinen soll

Main-Class: in welchem Paket die Class Datei lokalisiert ist

Folder: Unterpunkt, unter welchem das Modul in der Liste erscheinen soll

Nach einem Neustart von Together sollte das Modul unter dem angegebenen Namen im Unterpunkt in der Struktur startbar sein.

Das Modul wurde in der Art aufgebaut, dass zu Begin die Funktion *run* aufgerufen wird. Hier werden nun nach der Bestimmung der *root–Pakete* all diese und auch die Unterpakete nacheinander verarbeitet und die globalen Variablen gefüllt. Die Erlüterung der Variablen befindet sich auf Seite **??**. Wichtig hierbei ist, dass bei dem Suchen der Muster nun nicht jedes mal neu alles eingelesen werden muss. Es wird ausschließlich auf den Variablen gearbeitet.
Nach dem "Füllen" dieser werden nun nacheinander alle gefundenen Klassen durchlaufen und jede wird auf das Muster geprüft. Im Anschluß werden die Ergebnisse mit Hilfe der Ausgabefunktionen auf der MessagePane ausgegeben und das Modul beendet den Suchlauf.

3.1 Allgemeine Funktionsaufrufe

Folgende Funktionen dienen dem allgemeinen Programmablauf und werden im Weiteren näher beschrieben:

- run(IdeContext)
- faerbeKlasseInDiagramm(String,String,String,String)
- print(String)
- packageProcessing(RwiPackage)
- doActionUponPackage(RwiPackage)
- doActionUponNode(RwiNode)
- doActionUponClass(RwiNode)
- associationToClass(String,String)
- getSuperClasses(RwiNode)
- getImplementedClasses(RwiNode)
- getAggregation(RwiNode)
- getAssociation(RwiNode)
- getSubClasses(RwiNode)
- addSubClass(String, String)
- appenVector(Vector,Vector)
- getNamesFromVector(Vector)

3.1.1 Die Funktion run(IdeContext)

Diese Funktion wird beim Start des Modules aufgerufen. Als Parameter wird ein IdeContext übergeben, das Informationen über markierten Elemente zum Startzeitpunkt des Models enthält. Dieser Context wird allerdings nicht weiter beachtet, sondern es wird sich auf alle Klassen im Projekt bezogen.
Zu Beginn wird getestet, ob überhaupt ein Projekt geöffnet ist. Ist dies der Fall, wird eine Verbindung zum Modell hergestellt, und es werden die RootPackages bestimmt. Diese werden der Reihe nach durchlaufen und über die Funktion PackageProcessing verarbeitet.
Im Anschluss wird die Initialfarbe aller Klassen des `default` –Klassendiagrammes auf

"weiss" gesetzt. Dazu dient die im weiteren Verlauf beschriebene Funktion *faerbeKlasseInDiagramm*.
Nachdem alle Vorarbeiten beendet sind, werden sämtliche Klassen durchlaufen und nach Mustern durchsucht. Am Ende erfolgt der Aufruf der Ausgabefunktionen für die einzelnen Muster, um alle Gefundenen auszugeben.

3.1.2 Die Funktion faerbeKlasseInDiagramm(String,String,String,String)

In der Funktion faerbeKlasseInDiagramm soll die übergebene Klasse (4. Parameter) in der übergebenen RGB-Farbe (erster bis dritter Parameter) gefärbt werden. Ziel ist die Veranschaulichung der gefundenen Klassen und der zugehörigen Muster.
Die Funktion holt sich zunächst das `default` Diagramm. Danach werden alle enthaltene Elemente durchlaufen. Wenn eine Klasse gefunden wurde, die mit dem übergebenen Klassennamen übereinstimmt, wird die Klasse entsprechend gefärbt.
Eine Besonderheit gibt es: Wenn der Klassenname `"ALL"` ist, so werden alle Klassen mit der Farbe gefüllt. Dies dient zum Färben aller Klassen mit derselben Farbe zu Beginn der Mustererkennung.

3.1.3 Die Funktion print(String)

Sie gibt den übergebenen String auf der Message Pane aus und soll den Code kürzer und übersichtlicher gestalten.

3.1.4 Die Funktion packageProcessing(RwiPackage)

Das übergebene Paket wird sogleich zur Funktion `doActionUponPackage` weitergereicht. Danach werden alle Unterpakete bestimmt und die Funktion rekursiv aufgerufen, um alle Pakete des Projektes zu erfassen.

3.1.5 Die Funktion doActionUponPackage(RwiPackage)

Es werden alle Knoten für das übergebene Paket bestimmt und zur Weiterverarbeitung an die Funktion `doActionUponNode` weitergereicht.
An dieser Stelle kann auch eine besondere Behandlung von Paketen erfolgen, da alle Pakete in dieser Funktion mindestens einmal aufgeführt werden.

3.1.6 Die Funktion doActionUponNode(RwiPackage)

Bei der Bestimmung der Art des übergebenen Knotens ist die Eigenschaft *"Class"* das nötige Suchkriterium. Ist eine gefunden, wird diese in die globalen Variablen `_classes` und `_classNamesToClass` eingefügt. Danach erfolgt das Durchreichen der übergebene (Klassen-)Knoten an die Funktion `doActionUponClass`.

3.1.7 Die Funktion doActionUponClass(RwiPackage)

Alle relavanten Klasseninformationen der übergebenen Klasse werden in dieser Funktion extrahiert. Hierzu werden die Member der Klasse bestimmt, nacheinander durchlaufen und auf Attribute und Operationen überprüft. Entsprechend der gefundenen Eigenschaft erfolgt nun das Zufügen der Member zu den lokalen Vektoren *attributes* bzw. *operations*. Nach dem Durchlauf wird den globalen Hashtables `_attributes` und `_operations` die Kombination von Klassenname und dem Vektor von Attributen bzw. Operationen angefügt.
Im weiteren Verlauf erfolgt die Speicherung aller Superklassen, implementierter Klassen, Aggregationsklassen und Assoziationsklassen in den entsprechenden Hashtables mit der zugehörigen Klasseninstanz. Zu guter Letzt wird noch die Funktion `getSubClasses` aufgerufen. Der Grund des expliziten Aufrufes dieser Funktion liegt darin begründet, dass sie keinen Rückgabewert besitzen kann. Der Grund ist im entsprechenden Abschnitt auf Seite 15 erläutert.

3.1.8 Die Funktion associationToClass(String,String)

Die Aufgabe der Funktion associationToClass besteht in der Erkennung, ob die übergebene Klasse (1. Parameter) eine Assoziation zu der Superklasse (2. Parameter) oder zu einer der Unterklassen besitzt. Wird diese Verbindung nicht gefunden, wird false zurückgegeben.

3.1.9 Die Funktion getSuperClasses(RwiNode)

Hier ist die Bestimmung alle Oberklassen der Übergebenen und Rückgabe des entsprechenden Vectors das Ziel.
Dazu werden alle ausgehenden Links auf Generalisierung überprüft. Wenn diese Eigenschaft gefunden ist, wird das Ziel des Links dem lokaen "result"-Vektor hinzugefügt. Durch rekursiven Aufruf dieser Funktion werden auch alle Oberklassen der gefundenen bestimmt und mittels der Funktion `appendVector` angefügt. Das ist notwendig, da man sonst nur die direkten Oberklassen bestimmen kann.

3.1.10 Die Funktion getImplementedClasses(RwiNode)

Alle Klassen, welche die übergebene implementieren, werden in dieser Funktion als Vector zurückgegeben. Genau wie bei den Superklassen ist auch hier der rekursive Aufruf der Funktion notwendig, um alle weiteren Klassen zu bestimmt.

3.1.11 Die Funktion getAggregation(RwiNode)

Entsprechend der vorherigen Funktion erfolgt hier die Bestimmung aller aggregierten Klassen.

3.1.12 Die Funktion getAssociation(RwiNode)

Hier werden alle Klassen gefunden, welche eine Assoziation zu der übergebenen besitzen. Diese werden als Vektor zurückgeliefert.

3.1.13 Die Funktion getSubClasses(RwiNode)

In dieser Funktion werden alle Unterklassen der übergebenen bestimmt. Da die OpenAPI leider keine Funktion für "incommingLinks" unterstützt, muss improvisiert werden. "IncommingLinks" sollen hierbei die Seite der Generalisierungslinks darstellen, welche von der Unter– zur Oberklasse führen.
Zu Beginn werden alle Superklassen der übergebenen bestimmt. Diese haben als Unterklasse die Übergebene. Nun werden alle Oberklassen durchgegegangen und die Funktion `addSubClass` in der Art aufgerufen, dass die übergebene Klasse die Unterklasse ist, welche zu **jeder** der Oberklassen gehört.
Die gleiche Prozedur wird mit den implementierten Klassen gemacht. Auch diese werden alle durchlaufen und die übergebene Klasse wird als Unterklasse der Implementierten gespeichert.
Der Grund für den Rückgabewert *void* liegt darin begründet, dass hier kein Vektor mit **allen** Unterklassen zurückgegeben werden kann. Die Zugehörigkeit muss explizit über die Oberklassen durchgeführt werden.

3.1.14 Die Funktion addSubClass(String,String)

Die Super- und zugehörige Unterklasse werden an die Funktion übergeben. Ziel ist es, die Unterklassen in der globalen Variablen `_subClasses` zur Superklasse zugehörig abzuspeichern.
Zunächst werden alle bisherigen Unterklassen bestimmt. Existieren schon Einträge für die Unterklassen, wird dem lokalen Vektor `currentSubClasses` einfach die neue Unterklasse hinzugefügt. Ist der Vector leer, wird die lokale Variable initialisiert und die neue Unterklasse eingetragen.
Letztendlich wird der aktualisierte Vektor erneut in die globale Variable `_subClasses` eingefügt.

3.1.15 Die Funktion appendVector(Vector,Vector)

Ihr Ziel ist das Zusammenführen zweier Vektoren. Dabei wird Vektor 1 (Parameter 1) im Verlauf der Funktion an Vektor 2 (Parameter 2) angefügt.

3.1.16 Die Funktion getNameFromVector(Vector)

Durch sie wird ein Vektor, der die Namen der Objekte des übergebenen Vektors enthält, zurückgegeben. Dies ist nur bei Vektoren möglich, die RwiElemente enthalten. Die Einschränkung besteht, da zur Bestimmung der Namen die Eigenschaft `RwiProperty.NAME` verwendet wird und nur diese Elemente die Eigenschaft besitzen.

3.1.17 Die verwendeten globalen Variablen und ihre Bedeutung

In der folgenden Tabelle sind alle globalen Variablen aufgeführt. Sie dienen hauptsächlich der Speicherung der Klassenstruktur in dem geöffnetem Projekt. Auch für jedes Muster existieren Variablen, welche Informationen zu den Klassen, die das Muster implementieren, beinhalten.

Name der Variablen	Typ	Beschreibung
`_classes`	Vector	Instanzen aller Klassen
`_singleton`	Vector	Namen der Klassen, die Singleton sind
`_factoryMethod`	Vector	Name aller Klassen, die Fabrikmethode(n) enthalten
`_factory`	Vector	Name aller Fabrikklassen
`_observer`	Vector	Name aller Observerklassen
`_attributes`	Hashtable	Kombination von Klassenname(Key) und Vektor der Attribute
`_operations`	Hashtable	Kombination von Klassenname(Key) und Vektor der Operationen
`_superClasses`	Hashtable	Kombination von Klasseninstanz(Key) und Oberklassenvektor
`_subClasses`	Hashtable	Kombination von Klasseninstanz(Key) und Unterklassenvektor
`_classNameToClass`	Hashtable	Kombination von Klassenname(Key) und Klasseninstanz
`_implementedClasses`	Hashtable	Kombination von Klasseninstanz(Key) und Vektor der implementierten Klassen
`_aggregatedClasses`	Hashtable	Kombination von Klasseninstanz(Key) und Vektor der aggregierten Klassen

_associatedClasses	Hashtable	Kombination von Klasseninstanz(Key) und Vektor der assoziirenden Klassen
_factoryMethods	Hashtable	Kombination von Klassenname(Key) und Vektor der Fabrik-Operationen
_factoryProducts	Hashtable	Kombination von Klassenname(Key) und Vektor der Fabrikklassennamen
_abstractProduct	Hashtable	Kombination von Klassenname(Key) und Klassenname des abstrakten Produktes
_abstractFactory	Hashtable	Kombination von Klassenname(Key) und Vektor der Namen aller konkreten Fabriken
_concreteObserver	Hashtable	Kombination von Klassenname(Key) und Vektor der Namen aller konkreten Observer
_subject	Hashtable	Kombination von Klassenname(Key) und Namen des "obersten" Subjektes

3.2 Das Muster Singleton

Der Aufbau des Musters ist in Abbildung 3 aufgezeigt. Nach [Nau01] ist der folgende Suchalgorithmus notwendig, um es erkenn zu können:

```
für jede Klasse i (Singleton) do
  gibt es ein Attribut(Exemplar-Variable), das als Typ
    die eigene Klasse bzw. eine Oberklasse besitzt und statisch ist?
  ja: gibt es eine Methode (Exemplar-Operation), die als Rückgabetyp
    die eigene Klasse bzw. eine Oberklasse besitzt und die statisch ist?
  ja: gibt es keine public Konstruktoren, aber einen der protected
    bzw. private ist?
  ja: Muster gefunden
od
```

Probleme sind bei der Erkennung der Datentypen (z.B. Rückgabetyp) aufgetreten. Dieses Thema betrifft Zeiger und wird in Abschnitt 3.6.2 auf Seite 28 näher erläutert. Folgende Funktionen tragen zum Erkennen des Musters bei:

- getSingleton(RwiNode)
- staticAttributesWithSameTyp(RwiNode)

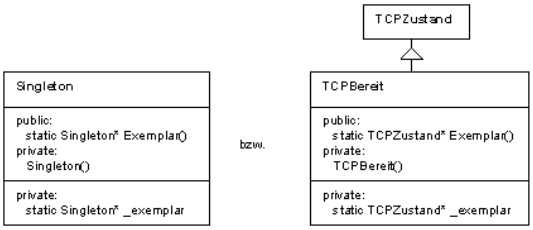

Abbildung 3: das Singleton – Muster

- staticOperationsWithSameTyp(RwiNode)
- noPublicConstructor(RwiNode)
- AusgabeSingleton()

3.2.1 Die Funktion getSingelton(RwiNode)

Diese Funktion ist einfach gehalten. Es werden nacheinander die Funktionen *staticAttributWithSameType*, *staticOperationsWithSameType* und *noPublicConstructor* auf die übergebene Klasse aufgerufen. Bei jedem Aufruf wird ein Merkmal des Musters überprüft (siehe [Klapp02]). Ist das Merkmal gefunden, wird die Zählvariable *zaehler* um eins erhöht. Das Singleton ist implementiert, wenn alle 3 Merkmale gefunden wurden und somit die Variable *zaehler* den Wert drei besitzt.
In dem Fall wird der Name der übergebenen Klasse an das globale Array *_singleton* angefügt.

3.2.2 Die Funktion staticAttributesWithSameTyp(RwiNode)

Ziel der Funktion staticAttributesWithSameTyp ist es, statische Attribute zu finden, die als Typ die eigene oder eine Oberklasse besitzen.
Anfangs werden alle Attribute der übergebenen Klasse bestimmt. Dieses Array wird in einer Schleife durchlaufen und auf die Property *RwiProperty.STATIC* überprüft. Wird die Eigenschaft gefunden, werden alle Namen der Oberklassen aus der globalen Variable *_superClasses* extrahiert und der Name der eigenen Klasse hinzugefügt. Das ist notwendig, da auch der Typ der Variablen die eigenen Klasse sein kann.
Wenn nun der Typ des Attributes mit dem Namen einer der bestimmten Klassen übereinstimmt, wird *true* zurückgegeben, andernfalls *false*.

3.2.3 Die Funktion staticOperationsWithSameTyp(RwiNode)

In dieser Funktion wird das Finden statischer Methoden fokusiert. Der Rückgabewert muss allerdings auch der eigenen oder einer Oberklasse entsprechen.

Der Ablauf erfolgt analog zur Funktion *staticAttributesWithSameTyp*. Der einzige Unterschied ist, dass hier alle Operationen anstatt der Attribute durchsucht werden.

3.2.4 Die Funktion noPublicConstructor(RwiNode)

Ihr Ergebnis ist das Vorhandensein eines nicht öffentlich Konstruktors. Nicht öffentlich heisst, dass er *private* oder *protected* sein kann.
Hierzu werden wieder alle Operationen der Klasse durchlaufen und nach der Eigenschaft *RwiProperty.CONSTRUCTOR* gesucht. Wenn diese Operation gleichzeitig nicht das Attribut *RwiProperty.PUBLIC* besitzt, ist ein nicht öffentlicher Konstruktor gefunden, und es wird true zurückgegeben.

3.2.5 Die Funktion AusgabeSingleton()

Bei der Ausgabe aller Singletons am Ende des Modules werden sämtliche Elemente der globale Variable *_singletons* durchlaufen und der Inhalt auf der Messagepane ausgegeben. Gleichzeitig wird jede im Hauptklassendiagramm vorhandene *Singleton*–Klasse zur Kennzeichnung mit Hilfe der Funktion *faerbeKlasseInDiagramm* gefärbt.

3.3 Das Muster Fabrikmethode

Der Aufbau des Musters ist in Abbildung 4 aufgezeigt. Nach [Nau01] ist der folgende Suchalgorithmus notwendig, um es erkennen zu können:

```
für jede Klasse i (Klasse mit Fabrikmethode) do
  für jede Methode j (Fabrikmethode) in Klasse i do
    erzeugt Methode j ein Objekt einer Klasse k (z.B. TuerMitZauberspruch),
      aber ist der Rückgabetyp l (z.B. Tuer) einer von k verschiedenen
      Klasse zugehörig?
    ja: ist Methode j polymorph?
    ja: ist die Klasse des Rückgabetyps l eine Oberklasse der Klasse
      des erzeugten Objekts k?
    ja: Muster gefunden
  od
od
```

Bei der Implementierung der Erkennung dieses Musters traten zwei größere Probleme auf:

- Die Virtualität von Methoden wird in den unterschiedlichen Programmiersprachen auch unterschiedlich dargestellt. Somit ist eine Unterscheidung zwischen bei-

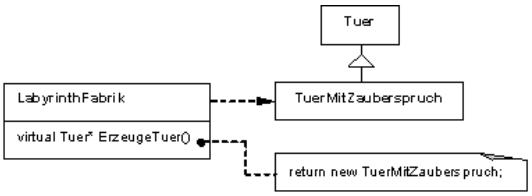

Abbildung 4: das Fabrikmethoden – Muster

spielsweise *Java* und *C++* unumgänglich. Das Problem wird in Abschnitt 3.6.1 auf Seite 27 ausführlich beschrieben.

- Die Erkennung des zurückgegebenen Datentyps macht es erforderlich, sich durch den Quellcode der implementierten Methoden der zu durchsuchenden Klasse zu bewegen. Da Together bei *C++* standardmäßig auf den Quellcode der Headerdateien zugreift, ist eine Angabe der *CPP–Datei* unvermeidbar. Dieses Thema wird in Abschnitt 3.6.1 besprochen.

Folgende Funktionen tragen zum Erkennen des Musters bei:

- getFabrikmethods(RwiNode)
- AusgabeFabrikmethods()

3.3.1 Die Funktion getFabrikmethode(RwiNode)

Diese Funktion ist sehr komplex und liefert alle Fabrikmethoden einer Klasse zurück, insofern sie welche besitzt.
Zu Beginn erfolgt eine Überprüfung, ob die "Klasse" ein *Interface* darstellt. Ist dem so, sind keine Methoden implementiert und es würde zu einem Fehler kommen, wenn man auf die Implementierung zugreifen möchte. Danach werden alle Operationen nacheinander durchlaufen und die folgenden Aktionen ausgeführt:
Es wird die Eigenschaft *RwiProperty.LANGUAGE* ausgelesen. Dieser String lautet *java*, falls es sich um eine Java-Klasse handelt. Nun muss zwischen Java und anderen Programmiersprachen unterschieden werden, da bei Java alle Methoden *virtuell* sind, es sei denn, sie sind mit *final* deklariert. Bei *C++* ist das beispielsweise nicht der Fall und eine Funktion muss explizit als *virtuell* gekennzeichnet werden. Als Indikator, ob die Funktion *virtuell* ist, dient die lokale Boolean Variable *virtuelleFkt*. Diese wird in den Zweigen der Unterscheidung der einzelnen Sprachen gesetzt, und es wird nur fortgefahren, wenn diese Variable *true* ist. Anfangs wird die Variable auf *false* gesetzt, da davon ausgegangen wird, dass die Funktion nicht virtuell ist.
Bei *Java* wird getestet, ob die Eigenschaft *RwiProperty.FINAL* nicht *wahr* und somit die Funktion *virtuell* ist.

Bei *C++* wird direkt auf die gesuchte Eigenschaft *RwiProperty.VIRTUAL* geprüft.
Nach dem Speichern des kompletten Code des Funktionsrumpfes in die lokale Variable *code* werden alle einzelnen Statements ausgelesen und in der Statementenumeration *statementenum* gespeichert. Bedingung dafür ist natürlich, dass der Code–Block nicht leer ist. Hier tritt das Problem der verschiedenen Programmiersprachen erneut auf. Bei *C++* ist es ein wenig umständlicher, auf die Implementierung einer Funktion zuzugreifen, da diese nicht in der Headerdatei steht. Weitere Informationen zu diesem Problem befinden sich auf Seite 27.
Bedingungen wie beispielsweise *if–then–else* Anweisungen werden als ein Statement behandelt. Wenn nun auf diese "Teilstatements" eingegangen werden muss, sind die "Statementblöcke" wieder als ein Codeblock auszulesen.
Im Anschluß werden nun alle Statements auf ein Return-Statement überprüft. Wenn eines gefunden wurde, wird der Typ des zurückgegebenen Wertes in die lokale Variable *opReturnType* gespeichert.
Nun wird der definierte Rückgabetyp der eigentlichen Funktion bestimmt. Die Überprüfung, dass der Rückgabewert und der zurückgegebene Wert unterschiedlich sind, erfolgt aus der Definition für das Finden des Musters (siehe [Nau01]). Weiterhin muss dann getestet werden, ob der Typ des Rückgabewertes in dem Projekt vorhanden ist. So werden alle anderen Rückgabetypen, die standardmäßig implementiert sind (z.B. Arrays, int u.s.w.), herausgefiltert. Dies ist notwendig, da ein Produkt (welches der Rückgabetyp repräsentiert) ja nur eine "eigene" Klasse sein kann.
Im weiteren Verlauf werden alle Ober– und Implementierungsklassen des zurückgegebenen Typs bestimmt. Nun erfolgt der Vergleich mit dem Rückgabewert der Funktion. Wenn dieser in dem Namensvektor vorhanden ist, wurde das Muster gefunden. Nun folgen noch einige Schritte zur Speicherung der gefundenen Fabrikmethode.
Dafür wird der lokale Vector *factoryMethodNames* mit dem Namen der Fabrikmethode gefüllt. In der Variablen werden alle Fabrikmethoden gesammelt und am Ende der Funktion zurückgegeben. Weiterhin wird überprüft, ob der zurückgegebene Wert schon in der lokalen Variable *factoryProducts* vorhanden ist. Das dient zur Speicherung der Produkte, die gebaut werden können. Falls der Wert noch nicht vorhanden ist, wird der Typ des zurückgegebenen Wertes der Variablen zugefügt. Dies ist notwendig, da alle Fabrikmethoden unterschiedliche Produkte bauen müssen. Nun folgt noch der Test, ob schon das abstrakte Produkt in der globalen Variable *_abstractProduct* für die Klasse existiert. Hier wird ebenfalls der Rückgabewert bei Nichtvorhandensein hinzugefügt. Das abstrakte Produkt ist in dem Fall der Rückgabetyp der Methode. Die Behandlung kann so erfolgen, da es nur ein abstraktes Produkt geben kann und alle konkreten Fabriken dieses in einer Fabrikmethode zurückgeben.
Bei gefundenen Fabrikmethoden wird nun der Vektor aller Produkte, die erstellt werden können (*factoryProducts*), in die globale Variable *_factoryProducts* zugehörig zur

Klasse eingetragen. Als letzter Schritt erfolgt die Rückgabe des Vektores mit den Fabrikmethoden an die aufrufende Funktion.

3.3.2 Die Funktion AusgabeFabrikmethods()

Wie auch die anderen Ausgabefunktionen dient diese zur übersichtlichen Ausgabe der restlichen Fabrikmethodenmuster, welche nicht in der abstrakten Fabrik abgebildet sind.
Hierbei wird der verbleibende Inhalt des Vectors *_factoryMethod* durchlaufen und zuerst der jeweilige Klassenname ausgegeben. Im weiteren Verlauf werden dann alle Methoden aus der globalen Variable *_factoryMethods* und alle Produkte aus der Variablen *_factoryProducts* ausgelesen und auf der MessagePane ausgegeben. Auch hier werden die entsprechenden Klassen im *Default*-Diagramm farblich markiert.

3.4 Das Muster abstrakte Fabrik

Der Aufbau des Musters ist in Abbildung 5) aufgezeigt. Nach [Nau01] ist der folgende Suchalgorithmus notwendig, um es erkennen zu können:

```
für jede Klasse i (konkrete Fabrik) do
  zaehler=0;
  für jede Methode j (Fabrikmethode) der Klasse i do
    erzeugt Methode j etwas?
    ja: ist Rückgabetyp eine Oberklasse von Erzeugtem?
    ja: zaehler+1;
  od
  zaehler größer oder gleich 2?
  ja: Muster gefunden;
od
```

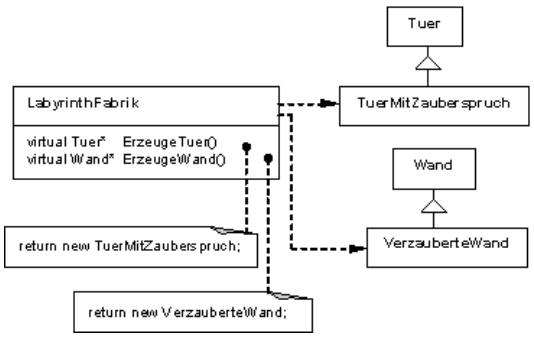

Abbildung 5: das abstrakte Fabrik – Muster

Problematisch ist in diesem Fall die Suche nach den abstrakten Fabriken. Die Lösung ist in der Funktion getAbstractFactoryClass gelöst und wird auf Seite **??** beschrieben. Folgende Funktionen tragen zum Erkennen des Musters bei:

- getFabrik(RwiNode)
- getAbstractFactoryClass(RwiNode)
- AusgabeFabrik()

3.4.1 Die Funktion getFabrik(RwiNode)

Die Funktion getFabrik stellt die Hauptfunktionalität für das Auffinden einer *abtrakten Fabrik* bereit.
Nach der Definition des Musters ist eine Klasse, die mindestens zwei Fabrikmethoden besitzt, eine *abstrakte Fabrik*. In dieser Funktion werden nun alle Fabrikmethoden für die übergebene Klasse bestimmt (mittels der schon auf Seite 20 beschriebenen Funktion *getFabrikmethode*) und im lokalen Vektor *fabrikMethoden* gespeichert. Falls es mindestens eine Fabrikmethode gibt, muss die Klasse in die globale Variable *_factoryMethod* aufgenommen und die Methoden zugehörig zur Klasse in die globale Variable *_factoryMethods* gespeichert werden, um später alle Fabrikmethoden einer bestimmten Klasse zu finden.
Wenn nun mindestens 2 Fabrikmethoden gefunden werden, handelt es sich um eine abstrakte Fabrik. Um diese abzuspeichern, werden folgende Schritte ausgeführt:
Zum Finden der abstrakte Fabrik dient die Methode *getAbstractFactoryClass* mit der übergebenen Klasse als Parameter. Ist der zurückgegebene Wert (in der Variablen *abstractFactorySuperClass*) ein leerer String (d. h. es existiert keine Oberklasse), wird in den implementierten Klassen nach einer Oberklasse gesucht. Wenn eine gefunden wurde, so ist sie die abstrakte Fabrik, wenn nicht, ist es die aktuelle Klasse. Falls der String nicht leer ist (d. h. es wurde eine Oberklasse gefunden), erfolgt wieder die Suche auf implementierte Oberklassen bezüglich der Variablen *abstractFactorySuperClass*. Werden keine gefunden, so ist die in der Variablen gespeicherte Klasse die abstrakte Fabrik, ansonsten ist es die implementierte.
Letztendlich müssen noch die Variablen *_factory* und *_abstractFactory* gefüllt werden. Hierzu werden alle bisherigen Klassen zur abstrakten Fabrik ausgelesen (über die Variable *_abstractFactory*). Gibt es keine bisherigen, wird ein neuer Vektor angelegt und die abstrakte Fabrik wird an den Vektor *_factory* angehängt. Nun wird noch die Klasse zum bisherigen hinzugefügt, und der Vektor wird korrespondierend zur abstrakten Fabrik in die Hashtable *_abstractFactory* gespeichert.

3.4.2 Die Funktion getAbstractFactoryClass(RwiNode)

Ziel der Funktion ist die Bestimmung der *abstrakte Fabrik*. Dazu werden alle Oberklassen der übergebenen bestimmt. Gibt es mehr als eine, so wird die Funktion rekursiv aufgerufen und der String der Oberklasse zurückgegeben. Falls es genau eine Oberklasse gibt, so ist sie mit hoher Wahrscheinlichkeit die *abstrakte Fabrik*. Wenn nun allerdings diese Klasse eine andere implementiert, so ist sie die *abstrakte Fabrik*. Diese Überprüfung findet in der Funktion *getFabrik()* statt.

3.4.3 Die Funktion AusgabeFabrik()

Die Ausgabe aller Fabriken funktioniert ähnlich der Ausgabe der Fabrikmethoden. Es werden alle Elemente des Arrays *_factory* durchlaufen. Jedes Element dieses Arrays stellt eine abstrakte Fabrik dar und der Name wird ausgegeben und auch die Klasse im *default*-Klassendiagramm markiert. Nun werden alle Unterklassen der Abstrakten–Fabrik–Klasse bestimmt. Diese werden aber erst im späteren Verlauf ausgegeben. Nach der Ausgabe des abstrakten Produktes folgen nun alle Unterklassen (Fabriken). Nachdem jetzt noch alle Produkte, die gebaut werden können, in die Variable *alleProdukte* geschrieben wurden, werden auch diese auf der Message–Pane ausgegeben. Wie bei den anderen Ausgabefunktionen werden alle Klassen entsprechend gefärbt.

3.5 Das Muster Beobachter

Der Aufbau des Musters ist in Abbildung 6 aufgezeigt. Nach [Nau01] ist der folgende Suchalgorithmus notwendig, um es erkennen zu können:

```
für jede Klasse i (Subjekt) do
  besitzt Klasse i 1-zu-n-Aggregation auf andere Klasse k (Beobachter)?
  ja: zaehler=0;
  für jede Methode j der Klasse i do
    besitzt Methode j als Parameter die Klasse k?
    ja: zaehler+1;
  od
  zaehler=2?
  ja: für alle Unterklassen l (konkrete Beobachter) von Klasse k do
    besitzt Klasse l Referenz auf Klasse i bzw. eine Unterklasse
      davon (konkretes Subjekt)?
    ja: Muster gefunden
  od
od
```

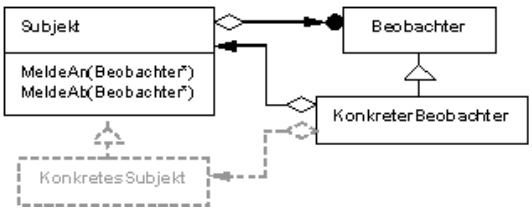

Abbildung 6: das Observer – Muster

Das größte Problem beim Finden des Musters war die Erkennung der 1–zu–n Aggregation zwischen Subjekt und Beobachter. Nicht immer wurde sie richtig, meist nur als Assoziation, beim Reverse Engineering in Together eingebunden. Näheres hierzu ist auf Seite 3.6.2 zu lesen.
Funktionen tragen zum Erkennen des Musters bei:

- getObserverRwiNode
- getOperationWithParameter(String, String)
- AusgabeObserver()

3.5.1 Die Funktion getObserver(RwiNode)

Zur Bestimmung aller Observerklassen werden zu Beginn alle aggregierten Klassen zur übergebenen bestimmt (vgl. [Nau01]). Die gefundenen Klassen werden nun nacheinander durchlaufen und die Funktion *getOperationWithParameter* mit den Parametern übergebene Klasse und aggregierte Klasse aufgerufen. Wenn die Anzahl der gefunden Operationen (also der Rückgabewert der aufgerufenen Funktion) genau zwei beträgt, so erfolgt die Ausführung der Schritte zu weiteren Erkennung:
Es werden alle Unterklassen der Aggregierten bestimmt und der *boolean* Wert der Variablen *ind* auf *true* gesetzt. Diese Variable indiziert, ob **alle** Unterklassen eine Assoziation zur Oberklasse besitzen. Wenn der Indikator noch auf *true* steht, wird das Vorhandensein einer Assoziation zur übergebenen Klasse geprüft und der Indikator entsprechend neu gesetzt. Eine Behandlung dieser Eigenschaft ist nicht sinvoll, wenn schon eine Klasse gefunden wurde, welche keine Aggregation hält, da nach dem Suchalgorithmus (vgl. [Nau01]) **alle** Unterklassen diese Eigenschaft besitzen müssen.
Falls der Indikator nach Durchlauf aller Unterklassen immer noch *wahr* ist, wurde das Muster gefunden.
Die aggregierte Klasse wird in die globale Variable *_observer* aufgenommen. Zur Bestimmung aller konkreten Observer ist es nötig, alle Unterklassen des abstrakten Observers zu bestimmt. Da dies die konkreten Observer darstellen, werden sie in die lokale Variable *concreteObserver* eingefügt und auch alle Unterklassen der Unterklasse. Zum

Schluss werden noch die beiden globalen Variablen *_concreteObserver* und *_subject* gefüllt.

3.5.2 Die Funktion getOperationWithParameter(String,String)

In dieser Funktion soll die Anzahl der Operationen bestimmt werden, welche als Parameter den Typ der übergeben aggregierten Klasse hat. Dies ist ein notwendiges Kriterium für das Observer Muster (vgl. [Nau01]). Hierzu werden alle Operationen bestimmt und durchlaufen. Im weiteren Verlauf werden alle Parameter bestimmt. Von diesen gefundenen Parametern werden noch die Subproperties *NAME* und *TYP* extrahiert. Der Name der aggregierten Klasse wird mit dem gefundenen Typ des Parameters verglichen, und bei Übereinstimmung wird die lokale Zählervariable *zaehler* erhöht. Zurückgegeben wird dann letztendlich der Wert dieser Variablen.

3.5.3 Die Funktion AusgabeObserver()

Zur Ausgabe aller gefunden Observer wird das bisher schon dargestellte Schema angewendet. Alle beteiligten Klassen werden eingefärbt. Zu Beginn wird der Observer ausgegeben, gefolgt von den konkreten Observern. Dann werden das Subjekt und die konkreten Subjekte, welche alles Unterklassen des Subjektes sind, in die MessagePane geschrieben.

3.6 Aufgetretene Probleme

In diesem Abschnitt sollen aufgetretene Probleme bei der Implementierung besprochen und kurze Lösungsvorschläge angegeben werden.
Die Probleme teilen sich dabei in die folgenden Kategorien ein:

- Probleme, die sich sprachübergreifend darstellen
- Probleme, die Together beim Reverse Engineering mit sich bringt

3.6.1 Probleme in unterschiedlichen Sprachen

Das wohl größte Problem ist, dass Together bei verschiedenen Programmiersprachen ein Reverse Engeneering durchführen kann. Dabei erkennt es alle Verbindungen (wie beispielsweise Assoziationen und Aggregationen) der Module, Klassen und anderer Objekte verschiedenartig "gut". Dies soll aber im nächsten Abschnitt besprochen werden. In dieser Arbeit sollen allerdings nur Probleme, die zwischen den Programmiersprachen *C++* und *JAVA* auftreten, besprochen werden, da eine Behandlung weiterer den Umfang der Arbeit sprengen würde. Weiterhin werden nur Probleme aufgezeigt, die mit den implementierten Mustern aufgetreten sind. Somit kann nicht ausgeschlossen

werden, dass noch viele weitere Probleme bei der Implemetierung anderer Muster auftreten können oder bisher nicht aufgefallen sind.

- Das erste Problem betrifft die Virtualität von Methoden. Während bei *Java* grundsätzlich alle Methoden virtuell sind, muss dies bei *C++* explizit angegeben werden. Die einzigen Java–Methoden, die nicht virtuell sind, müssen als *final* definiert werden. Dieses Problem betrifft vor allem das Muster Fabrikmethode von Abschnitt 3.3.
  Um es zu lösen, bedarf es einer Betrachtung aller möglichen Programmiersprachen und einer "exclusiven" Behandlung jeder. Zum Finden des Musters Fabrikmethode von Abschnitt 3.3 wird zu Beginn geprüft, in welcher Sprache die Klasse geschrieben ist. Aufgrund dieser Tatsache erfolgt im weiteren Verlauf die Erkennung der Virtualität der Methoden.

- Ein weiteres großes Problem ist die Unterscheidung zwischen Header– und Implementierungs – Dateien in *C++*. Da bei einigen Mustern im Quellcode nach bestimmten Formen bzw. Ausdrücken gesucht werden muss, wird die Implementierung der Klasse benötigt. Dies ist bei Java–Klassen kein Problem, da sie nur aus einer Datei bestehen. Problematisch sind hingegen die C–Klassen. Die Klassendiagramme bilden nur die Header–Dateien ab, auf welchen man dann entsprechend zugreifen kann.
  Das Problem betrifft von den implementierten Mustern nur das der Fabrikmethoden, da hier explizit auf die Implementierung der Methoden zugegriffen werden muss. Momentan ist das Problem in der Art gelöst, dass versucht wird, an die Implementierung der Methode zu kommen, egal um welche Sprache es sich handelt. Bei *C++* würde es so verlaufen, dass gleich auf die Header–Datei gesprungen wird und somit keine Implementierung gefunden werden kann. Das kann durch eine einfache *if*–Bedingung mit dem Test, ob der Codeblock leer ist, abgefangen werden. Um nun auch *C++*–Programme nach dem Fabrikmethodenmuster zu durchsuchen, ist die o.a. Implementierung nach einer Abfrage der Programmiersprache ratsam. Der Test, ob der Code leer bzw. *NULL* ist, sollte trotzdem bestehen bleiben, da es sein kann, das Methoden u.U. noch nicht implementiert sind.
  Mit der Behandlung wird es dann auch möglich sein, C++ Implementierungen zu durchsuchen und eine Erkennung zu ermöglichen.

3.6.2 Probleme, die von Together ausgehen

- Ein Problem zum Thema Laufsicherheit betrifft die Interface–Klassen. Da hier keine Methoden implementiert sein können, sind auch keine Abfragen zu diesen möglich. Eine einfache Lösung besteht darin, die Implementierungsklassen durch eine Abfrage immer dann abzufangen, wenn auf Implementierungsteile zugeriffen werden muss.

- Das nächste Problem erfolgt aus der Implementierung von Zeigern. Da z.B. das Singleton–Muster auch mit Zeigern implementiert werden kann (Zeiger auf die eigene Klasse), ist dies nicht so einfach zu erkennen. Hierbei erkennt Together den Typ einer Variablen dann inklusive des "*", ob mit oder ohne Leerzeichen. Da es eine solche Klasse nicht geben kann, ist es notwendig, den String, in welchem der Typ steht, zu bearbeiten. Zum einen kann man nach einem Leerzeichen suchen und alle folgenden Zeichen abschneiden, da der Name einer Klasse ja keine Leerzeichen enthalten darf. Zum anderen kann man nach einem "*" suchen, welches auch in keinem Klassennamen vorkommen darf. Der so modifizierte Namensstring gibt dann den "richtigen" Klassennamen an und kann verglichen werden.
  Dies wurde beispielsweise schon bei dem Singletonmuster umgesetzt. Es sollte geprüft werden, ob diese Behandlung vielleicht auch bei anderen Mustern sinnvoll ist, da ja auch Parameter Zeiger sein können.

- Ein weiteres Problem könnte in beide Kategorien gleichermaßen eingeordnet werden. Es geht um das Reverse Engineering von Projekten. Hierbei fiel auf, dass in Java Projekten die Aggregation zwischen zwei Klassen nur als Assoziation erkannt wurde. Dies ist bei C++ Projekten nicht so. Ein Grund für dieses unterschiedliche Verhalten konnte nicht festgestellt werden. Eine Möglichkeit ist, dass diese "fehlerhafte" Erkennung an der unterschiedlichen Implementationen von Aggregationen liegen kann. Werden z.B. aggregierte Klassen in einem Array gespeichert, kann Together nicht erkennen, von welchem Typ die Klassen sind, die in diesem Array stehen und die Aggregation wird deshalb nicht richtig erkannt.
  Eine automatisierte Lösung für dieses Problem kann leider nicht aufgezeigt werden. Einzige Möglichkeit wäre das Ändern per Hand aller solcher Assoziationen auf Aggregation, falls dies nötig ist.

3.6.3 Zusammenfassung der Probleme

Alles in allem existieren viele verschiedenartige Probleme bei der Mustererkennung. Man sollte sich vor der eigentlichen Implementierung Gedanken über diese machen und vielleicht schon im Vorfeld versuchen, Lösungen zu finden.

Da sie meist erst entdeckt werden, wenn “Beispielprojekte“ diese aufdecken, ist es fast unmöglich alle Probleme zu finden. Hier wäre es sinnvoll, wenn man verschiedene “Testprojekte“ sammelt, bei denen man mit Sicherheit weiss, ob die Muster implementiert sind und auf die Erkennung achtet, Fehlerquellen lokalisiert und vor allem auch dokumentiert. Diese Testprojekte sollten aber aus realen Projekten stammen, da ein eigenes Testprojekt unter Umständen genau die Eigenschaften der Muster implementiert und leicht veränderte (z.B. beim Singleton einen Zeiger auf die Klasse übergeben) nicht beachtet.

4 Bewertung der implementierten Algorithmen

In diesem Abschnitt werden die implementierten Algorithmen bezüglich ihrer Zuverlässigkeit kurz bewertet. Grundlage hierfür sollen die Erfahrungen bei der Entwicklung des Modules sein. Zu Beginn wird nun kurz tabellarisch aufgezeigt, mit welcher Wahrscheinlichkeit die Muster erkannt werden können. Hierzu wird zwischen den beiden Programmiersprachen *Java* und *C++* unterschieden. Der Grund hierfür wurde bereits in Abschnitt 3.6 beschrieben.

Muster	**C++**	**Java**
Singleton	sehr hohe Wahrscheinlichkeit	sehr hohe Wahrscheinlichkeit
Fabrikmethode	wird nur erkannt, insofern die Implementierung der Methoden in der Header Datei erfolgt ist (siehe Abschnitt 3.6)	gute Erkennung
abstrakte Fabrik	hängt von der Erkennung der abstrakten Fabrik ab, gutes Finden der abstrakten Fabrik	s. C++
Observer	gutes Finden, wenn Aggregation richtig erkannt werden	Wahrscheinlichkeit bei Reverse Engineering geringer, da Aggregation wahrscheinlich nicht richtig erkannt wird

4.1 Singleton

Das Singleton Muster ist wohl das einfachste aller hier implementierten. Die grundsätzliche Implementation wurde ein paar Änderungen in der Programmstruktur von [Klapp02] übernommen.
Auch die Erkennung des Musters wurde verbessert. Hierbei wurde das Problem der Zeiger (vgl. Seite 28) gelöst und implementiert.
Bei der Hauptseminararbeit von Herrn Klappstein [Klapp02] war es weiterhin so, dass alle Oberklassen, Unterklassen und sonstige Zusammenhänge jedes mal neu eingelesen werden. In der jetzigen Version werden diese Zusammenhänge in Vektoren und Hashtables zu Beginn gespeichert und die Suche wird auf den Variablen ausgeführt. Das bringt einen großen Geschwindigkeitsvorteil, da nicht wieder das gesamte Projekt durchsucht

werden muss.
Meiner Meinung nach ist das Finden eines Singleton Musters am zuverlässigsten, was natürlich auch durch die Einfachheit resultiert. Es kann trotzdem noch eigensinnige Implementierungen geben, welche nicht erkannt werden, aber im Grundsatz sollte *jedes* Singleton erkannt werden.

4.2 Fabrikmethode

Dieses Muster ist wesentlich komplexer als das des Singletons. Momentan ist ein sicheres Erkennung (vor allem bei C++ – Projekten) nicht möglich. Die Probleme sind im vorherigen Abschnitt schon erläutert worden. Da mit den Datentypen von Funktionen und Rückgabewerten gearbeitet wird, ist es wichtig zu wissen, wie Together in den einzelnen Programmiersprachen die Typen bestimmt. Hier sei auf das Problem der Zeigerdarstellung in C++ aus Seite 28 verwiesen.
Tests ergaben, dass Fabrikmethoden in Java – Projekten mit hoher Wahrscheinlichkeit erkannt werden, in anderen Programmiersprachen aber unter Umständen nicht. Eine vollständige Erkennung des Musters in C++ Projekten ist bei der momentanen Implementation auch nicht möglich, da nur auf Headerdateien zugegriffen werden kann.

4.3 Abstrakte Fabrik

Da das Muster der Abstrakten Fabrik hauptsächlich von dem Fabrikmethodenmuster abhängt, ist auch die Erkennung sehr eingeschränkt. Wenn die Fabrikmethoden gefunden werden, ist die Erkennung aller beteiligten Klassen sicher.

4.4 Beobachter

Die Erkennung eines Beobachtermusters hängt von der jeweiligen Programmiersprache und dem Reverse Engineering ab. Nach den Tests werden die Aggregationen beim Reverse Engineering von Java Projekten nicht richtig erkannt (siehe Seite 28). Meist wird daraus eine einfache Assoziation gemacht, welche dem Muster nicht genügt. Wenn dieses Problem gelöst wäre, ist das implementierte Modul recht gut für die Erkennung des Observer Musters geeignet.

5 Zusammenfassung

Alle hier implementierten Muster weisen gewisse Schwächen bei der Erkennung auf. Dies ist auf verschiedenste und teilweise bereits aufgezeigten Probleme zurückzuführen. Das Hauptproblem dabei ist die spezielle Implementierung jedes einzelnen Musters, welche von der Theorie abweichen und somit nicht hundertprozentig erkannt werden kann.
Together eignet sich recht gut für die Erkennung von Entwurfsmustern. Da aber jeder Algorithmus [Nau01] seine eigene Implementierung und Probleme mit sich bringt, ist ein gewisser Zeitaufwand zur genauen Planung der Implementation unumgänglich.
Weiterhin sollten im Vorfeld reale Projekte gesucht werden, die das zu implementierende Muster definitiv enthalten. So wird nach der Fertigstellung die Fehlersuche bei Nichterkennen um einiges erleichtert.

6 Ausblick

Es gibt noch viele Probleme, die bei der Implementierung beachtet werden müssen. Dies beginnt bei der Unterscheidung der verschiedenen Programmiersprachen und endet bei der Fehlerbehandlung, wenn beispielsweise manche Methoden nicht implementiert sind und auf deren Rumpf zugegriffen wird.
Eine Erweiterung ist die Erkennung von "fast" vollständigen Mustern, wenn also nur Teile erkannt werden. Dies könnte dem Nutzer als "Könnte Muster sein" bekannt gemacht werden. Vorteil wäre die Erkennung von Mustern, die u.U. leicht abgewandelt sind und somit nicht hundertprozentig mit den Suchkriterien übereinstimmen oder einfach falsch implementierte Muster (z.B. public definierter Konstruktor beim Singleton).
Insgesamt ist die Aufgabe der Implementierung weiterer Muster sehr weitreichend und sollte vor der eigentlichen Implementierung sinnvoll geplant werden.
Ein guter Ansatzpunkt für weitere Hauptseminare oder Studienarbeiten ist die Entwicklung einer übersichtlichen Oberfläche zur Darstellung aller gefundener Muster. Da Together–Module in Java geschrieben sind, ist es möglich, die verschiedenen Klassen von Java zur Darstellung einer solchen Oberfläche zu nutzen.

Anhang

A Funktionen

Hier sind alle implementierten Funktionen mit einer kurzen Beschreibung zusammengefasst.

Funktion	**Beschreibung**
run	Startfunktion zum Initialiseren
faerbeKlasseInDiagramm	färbt die übergebene Klasse mit der üebergebenen Farbe
print	Ausgabe des übergebenen Strings auf der Message Pane
packageProcessing	abarbeiten der übergebenen Paketes und aller Unterpakete
doActionUponPackage	abarbeiten des übergebenen Paketes
doActionUponNode	abarbeiten aller Nodes in dem Paket
doActionUponClass	hier werden die folgenden Variablen gefüllt: • `_attributes` • `_operations` • `_superClasses` • `_implementedClasses` • `_aggregatedClasses` • `_associatedClasses` weiterhin wird die Funktion getSubClasses aufgerufen, um alle Unterklassen der aktuellen zu bestimmen
getSuperClasses	hier werden alle Superklassen der übergebenen bestimmt und als entsprechender Vector zurückgeliefert
getImplementedClasses	Hier werden alle Klassen bestimmt, die die übergebene implementiert und als Vector zurückgegeben
getAggregation	hier werden alle Klassen bestimmt, die einen Aggregationslink zur übergebenen haben und als Vector zurückgegeben
getAssociation	hier werden alle Klassen bestimmt, die einen Assoziationslink zur übergebenen haben und als Vector zurückgegeben
getSubClasses	Bestimmung aller Unterklassen zur übergebenen

addSubClass	fügt übergebene Unterklassen in den Hashtable assoziirend zur Oberklasse ein
appendVector	fügt zwei Vektoren zusammen
getNamesFromVector	extrahiert aus dem übergebenen Vector das Property NAME und liefert es ebenfalls als Vector zurück

B Projekte, die auf Muster durchsucht wurden

Von folgenden Projekten habe ich den Source Code mittels Reverse Engineering in Together integriert. Im Anschluß wurde das erstellte Modul zur Erkennung aller Muster gestartet. Leider ist die Trefferquote nicht sehr hoch gewesen. Auf Fehlersuche gehen ist leider nicht sinnvoll gewesen, da ich keine ausreichende Dokumentation der Projekte gefunden habe, in welcher die verwendeten Muster dargestellt wurden.
Nun folgt eine kurze Tabelle mit getesteten Projekten. Die Erkennung von Fabrikmethoden in *C++* Projekten ist zudem momentan noch nicht möglich gewesen. Der Grund ist bei den Problemen näher erläutert worden.

Projekt	**Sprache**	**gefundene Muster**
TExmacs	C++	keine Muster gefunden
OpenOffice	C++	Fehler bei der Mustererkennung
MySQL++	C++	keine Muster gefunden
MySQL	C++	kein Refactoring möglich durch Fehler im Together
MegaMek	Java	keine Muster gefunden
Art of Illusion	Java	keine Muster gefunden
VDR	C++	2 Singleton Muster gefunden (nach Lösung des Zeigerproblemes von Seite 28)

Literatur

[OpenAPI] Dokumentation der OpenAPI, die Together beiliegt

[Klapp02] Hauptseminararbeit von Jens Klappstein, 28. Oktober 2002

[Nau01] Diplomarbeit zum Thema Reverse–Engineering von Entwurfsmustern, Sebastian Naumann, 3. Dezember 2001

[Rud] Achim Ruder, TogetherSoft, Achim.Ruder@togethersoft.de

[Schw] Markus Schwarz, Markus.Schwarz@borland.com