Entwicklung einer Sprache zur Unterstützung von Design Patterns und Implementierung eines dazugehörigen Compilers

Diplomarbeit

Universität Rostock, Fachbereich Informatik

vorgelegt von	Stefan Bünnig
geboren am	13.08.1974 in Rathenow
Matrikelnummer:	094200949
Betreuer:	Prof. Peter Forbrig
	Dr. Ralf Lämmel
Abgabedatum:	06.07.1999

Zusammenfassung

Diese Diplomarbeit stellt eine neue Programmiersprache zur direkten Unterstützung von Design Patterns vor. In der neu entwickelten Sprache *PaL* (*Pa*ttern *L*anguage) wurde der objektorientierte Gedanke weiter vollzogen, so daß nicht nur Klassen wiederverwendbar sind, sondern auch komplexere Strukturen. Dieser Ansatz soll es dem Programmierer ermöglichen, Design Patterns, wie sie in [1] beschrieben werden, leichter wiederverwenden zu können und nicht bei jeder Anwendung neu implementieren zu müssen.

Einführend werden die Schwierigkeiten von gewöhnlichen objektorientierten Programmiersprachen im Umgang mit Design Patterns aufgewiesen. Es wird ein pattern-orientiertes Modell entwickelt, das auf das objektorientierte Modell aufsetzt, bei dem aber die Wiederverwendung komplexer Strukturen im Vordergrund steht.

Der Hauptteil dieser Diplomarbeit befaßt sich mit dem Entwurf der Sprache *PaL* auf der Basis von Eiffel und mit der Entwicklung des Prototyps eines Compilers, der von *PaL* nach Eiffel übersetzt. Der Präcompiler wird mit Hilfe von LDL spezifiziert und implementiert und stellt die Semantik der Sprache *PaL* dar.

Die Diplomarbeit wird mit einem Beispiel abgeschlossen, das die neuen Möglichkeiten im Umgang mit Design Patterns demonstriert und einen Einstieg in die Sprache *PaL* bietet.

Anmerkung

Basierend auf dieser Diplomarbeit und auf [2] wurde ein Artikel [3] veröffentlicht. Die in diesem Artikel vorgestellte Sprache wurde noch einmal überarbeitet. Auch wenn die Ausdruckskraft der Sprache in [3] gleich der hier vorgestellten Sprache *PaL* ist, unterscheiden sich die Sprachkonstrukte.

Abstract

This master's thesis introduces a new programming language for the direct support of design patterns. The newly created language *PaL* extends the object oriented paradigm by the facility to reuse complex structures of classes. This approach allows the developer of large-scale software to apply and reuse design patterns as described in [1] without ever reimplementing the underlying idea.

The aim of the first part is to discuss problems and obstacles that occur when handling design patterns in an object oriented environment. A pattern oriented model is introduced based on the object oriented model emphasizing the reuse of complex class structures.

The main part of this thesis addresses the design of the pattern oriented programming language *PaL* which is syntactically based on Eiffel and the development of a prototype of a precompiler translating from *PaL* to Eiffel. The introduced precompiler is implemented in LDL and represents the semantics of *PaL*.

Concluding the thesis, an example is presented in order to demonstrate the new pattern oriented mechanisms supported by *PaL* and to provide an introduction into the usage of *PaL*.

Remarks

A paper [3] has been published based on [2] and this master's thesis. For this purpose, the prototype language *PaL* has been revised and modified after the final version of this thesis. Note, that language constructs may have been changed. However, the expressiveness of *PaL* in this thesis is equal to the language used in [3].

CR-Klassifikation:

D.1.5, D.2.1, D.2.2, D.2.3, D.2.13, D.3.1, D.3.2, D.3.3, D.3.4

Key-Words:

Compiler, Design Patterns, Eiffel, LDL, Objektorientierte Programmierung, PaL, Pattern Language, Software Entwicklung, Wartung, Wiederverwendung

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die objektorientierte Programmierung setzt sich mehr und mehr gegenüber der strukturierten Programmierung durch. Der Hauptgrund dafür ist die Möglichkeit der Erhöhung der Wiederverwendbarkeit, der besseren Wartbarkeit und Erweiterbarkeit und der leichteren Kombination von Softwarekomponenten. Dennoch impliziert der Einsatz objektorientierter Programmierung nicht sofort den Nutzen dieser Vorteile. Es erfordert schon ein gewisses Geschick, einen guten objektorientierten Entwurf anzufertigen. Es gilt, die relevanten Objekte zu finden, sie zu Klassen zu abstrahieren und Hierarchien und Beziehungen zwischen den Klassen aufzubauen. Der Entwickler hat aus einer großen Anzahl von Entwurfsmöglichkeiten diejenige herauszufinden, die wahrscheinlich am ehesten aus den oben genannten Möglichkeiten einen Nutzen zieht. Er muß dabei bereits beim Entwurf erahnen, wo die Software später erweitert wird oder wo die Komponente zusätzlich zum Einsatz kommen könnte. Auf Grund dieser Überlegungen muß der Entwickler den richtigen Abstraktionsgrad finden. Nun resultiert Wartbarkeit nicht nur aus dem Abstraktionsgrad, sondern auch aus der Lesbarkeit des Quellcodes. Lesbarkeit und Abstraktion widersprechen sich meist, was dem Entwickler das Finden des geeigneten Abstraktionsgrades nicht erleichtert.

In [1] werden Strukturen identifiziert, die Standardlösungen für wiederkehrende Designprobleme darstellen. Durch die Umsetzung dieser Design Patterns ist es möglich, eine bessere Wartbarkeit der entstehenden Software zu gewährleisten. Mit Hilfe dieser Sammlung kann man den Aufwand beim Design vom Entwurf einer Struktur zur Lösung eines bestimmten Problems auf die Auswahl eines Design Patterns beschränken. Es bleibt jedoch beim Recycling der Idee. Dem Programmierer wird es nicht erspart, das Pattern für den jeweiligen konkreten Einsatz stets neu zu implementieren.

In den folgenden Unterkapiteln wird der Ist-Zustand der Wiederverwendung in objektorientierten Programmiersprachen kurz dargestellt. Es werden die Grenzen der objektorientierten Programmierung im Umgang mit Design Patterns aufgezeigt. Im zweiten Kapitel wird das Modell einer patternorientierten Sprache entwickelt, das die wünschenswerte Wiederverwendung von Design Patterns und anderer Strukturen realisiert. Zu diesem Modell wird die Syntax der patternorientierten Sprache *PaL* konstruiert. In den nachfolgenden Kapiteln soll die Semantik anhand eines Präcompilers in die objektorientierte Programmiersprache Eiffel erläutert werden. Der Quelle-Quelle-Übersetzer wird durch die von LDL [5] bereitgestellten Spezifikationsformalismen implementiert. Um eine Grundlage dafür zu schaffen, werden im dritten Kapitel LDL und die relevanten Besonderheiten des Compilers SmallEiffel vorgestellt.

Diese Arbeit betrachtet das Thema des patternorientierten Programmiermodells von der praktischen Seite. Die Entwicklung einer konkreten Sprache und eines Compilers soll die tatsächliche Anwendbarkeit dieses Modells zum Ziel haben. Für die theoretischen Grundlagen des patternorientierten Programmiermodells wird hier auf die Arbeit [2] verwiesen.

1.1 Wiederverwendung im objektorientierten Programmiermodell

Die Umsetzungen des objektorientierten Modells in konkrete Programmiersprachen variieren von Sprache zu Sprache in Umfang und Semantik. Aus der Vielzahl der unterschiedlichen Paradigmen werden hier die von Eiffel näher betrachtet, da die gesamte Diplomarbeit auf diese Sprache aufbaut.

1.1.1 Vererbung

Die ursprüngliche Idee zur Wiederverwendung von Klassen war die Vererbung. Mit ihr ist es möglich, eine neue Klasse zu definieren, indem man eine vorhandene Klasse erweitert. Die Schnittstelle der Klasse kann dabei erweitert, jedoch nicht eingeschränkt werden. Zur Schnittstelle einer Klasse zählen die Namen der Attribute und Methoden, die in Eiffel unter dem Sprachkonstrukt Features zusammengefaßt werden. In Eiffel und anderen objektorientierten Programmiersprachen kann man konkrete Angaben machen,

welche Features für Objekte welcher Klasse sichtbar sein sollen. Die Schnittstelle einer Klasse zu anderen Klassen kann daher kleiner sein, als die Gesamtheit aller Features. Da aber für die Unterklasse die komplette Oberklasse sichtbar ist, bilden alle Features die Schnittstelle einer Klasse. Die Implementationen der Methoden bleiben der Unterklasse wie allen andern Klassen verborgen. Sie können nur unverändert übernommen oder neu definiert werden. Zusätzlich gehören zur Schnittstelle die Typen der Attribute, die der Übergabeparameter und wenn vorhanden, der Typ des Rückgabewertes der Methoden. Auf die Möglichkeit, diese Typen bei einer Vererbung zu ändern wird noch eingegangen.

Stehen zwei Klassen in einer Vererbungsbeziehung, so können Objekte der Oberklasse durch Objekte der Unterklasse substituiert werden. Diese Substitution ist möglich, da sich alle Features der Oberklasse auf die der Unterklasse abbilden lassen. Da zur Übersetzungszeit noch nicht feststeht, ob im Programm zur Laufzeit die Methode der Ober- oder der Unterklasse aufgerufen wird, verhält sich dieser Methodenaufruf polymorph. Dieser **Polymorphismus** stellt auch an die Typisierung der Features in einer Vererbungsbeziehung zweier Klassen gewisse Anforderungen. Es gibt drei Ansätze, die sich in Flexibilität und Sicherheit unterscheiden.

Die Einfachste Möglichkeit ist das Konzept der **No-Varianz**. Das heißt, daß sich die Typen der Attribute, die Typen der Übergabeparameter und die Typen der Rückgabewerte der Funktionen in einer Vererbung nicht ändern dürfen. Dieses Konzept ist typsicher, aber auch am wenigsten flexibel.

Bei dem Konzept der **Contra-Varianz** dürfen die Parameter einer Methode bei einer Vererbung allgemeiner werden. Das heißt, ist ein Parameter einer Methode von Klasse A vom Typ Klasse B, so darf der Parameter dieser Methode in einer Unterklasse von Klasse A vom Typ einer Oberklasse der Klasse B sein. Dieses Konzept ist bei Parametern, die an eine Methode übergeben werden, typsicher. Die Methode der Unterklasse kann in jedem Fall auch die Parameter der Oberklasse aufnehmen. Contra-Varianz auf den Rückgabewerten einer Methode ist nicht typsicher. Im Falle einer polymorphen Anwendung einer Methode kann es vorkommen, daß ein speziellerer Rückgabewert erwartet wird, als die Methode liefert.

Bei dem entgegengesetzten Konzept der **Co-Varianz** können die Parameter einer Methode bei einer Vererbung spezialisiert werden. Das heißt, ist ein Parameter einer Methode von Klasse A vom Typ Klasse B, so darf der Parameter dieser Methode in einer Unterklasse von Klasse A vom Typ einer Unterklasse der Klasse B sein. Das Konzept der Co-Varianz ist typsicher, wenn man es auf die Rückgabewerte einer Methode anwendet. Auch wenn die Methode im Sinne der Oberklasse verwendet wird, können immer die spezielleren Rückgabewerte entgegengenommen werden. Bei den Übergabeparametern ist dieses Konzept nicht typsicher. Hier kann es vorkommen, daß eine Methode mit spezialisierten Parametern als Methode der Oberklasse eingesetzt wird und nicht mit den allgemeineren Übergabeparametern der Methode der Oberklasse zurechtkommt.

Ein flexibles und typsicheres Gesamtkonzept ist also Contra-Varianz auf den Übergabeparametern und Co-Varianz auf den Rückgabewerten. Dabei bleibt offen, wie Attribute behandelt werden, da sich diese beim Beschreiben wie Übergabeparameter und beim Auslesen wie Rückgabewerte verhalten. Eine mögliche Lösung hierfür ist der Zugriff über Set- und Get-Methoden.

In Eiffel wurde das Konzept der **Co-Varianz** auf Übergabeparametern und Rückgabewerten umgesetzt. Bei diesem Konzept dürfen alle Typen der Features bei einer Vererbung spezialisiert werden. Durch Anwendung von Co-Varianz wird es also möglich Typunsicherheiten zu implementieren, die sich nicht beim Compilieren des Programmes feststellen lassen. Da man aber nach einer Vererbung meist mit spezielleren Parametern arbeiten möchte und in den seltensten Fällen mit allgemeineren, ist das Konzept der Co-Varianz für den Programmierer leichter anzuwenden als das der Contra-Varianz.

In einigen Programmiersprachen, so auch in Eiffel, kann eine Klasse von mehreren Oberklassen erben. Diese **Mehrfachvererbung** kann zu Konflikten führen, wenn mehrere Methoden mit gleichem Namen aufeinanderfallen. In Eiffel gibt es zwei Ansätze zum Umgehen dieser Konflikte. Zum einen kann man die entsprechenden Methoden abstrakt erben, so daß maximal die Methode einer Oberklasse mit konkreter Implementation geerbt wird. Zum anderen kann man die Features umbenennen. Die Umbenennung eines Features bei Mehrfachvererbung von ein und der selben Klasse führt wiederum zu Problemen der polymorphen Anwendung dieses nun duplizierten Features. Hier muß ein Feature für die polymorphe Anwendung selektiert werden.

Die Klassenvererbung bietet also zwei Möglichkeiten, die Wiederverwendung in der Softwareentwicklung zu erhöhen. Zum einen fördern die flexiblen Entwurfsmöglichkeiten durch den Einsatz von Polymorphismus die Wiederverwendbarkeit der Software, zum anderen lassen sich durch Vererbung Implementationen wiederverwenden. Es ist einfach, die Vererbung als Mittel der Wiederverwendung von Implementationen zu nutzen, da sie von den objektorientierten Programmiersprachen direkt unterstützt wird. Dennoch gibt es einige Nachteile. Wenn eine Unterklasse eine Oberklasse erweitert, so befinden sich Teile der physischen Repräsentation der Unterklasse in der Oberklasse. Diese Teile der Oberklasse lassen sich nicht verändern, ohne die Unterklasse zu manipulieren. Es kommt zu einem Aufbruch der Kapselung. Ebenfalls ist es kritisch, wenn es Features in der Oberklasse gibt, die man nicht oder nur mit anderen Parametern gebrauchen kann. In diesem Fall muß man die Oberklasse manipulieren, um die Unterklasse korrekt implementieren zu können. In solchen Fällen ist zu prüfen, ob nicht die Objektkomposition der Klassenvererbung als Mittel zur Wiederverwendung der Implementation vorzuziehen ist.

1.1.2 Komposition

Objektkomposition bedeutet, daß ein Objekt zur Laufzeit eine Referenz auf ein anderes Objekt erhält. Die Komposition läßt sich als ebenso mächtiges Mittel zur Wiederverwendung von Implementationen einsetzen, wie die Vererbung. Diese Technik nennt sich Delegation. Bei der Delegation erhält ein Objekt eine Anfrage und leitet diese an ein Objekt der Klasse, deren Implementation wiederverwendet werden soll, weiter. Dies entspricht bei der Vererbung dem Fall, daß Unterklassen Anfragen an Oberklassen durchlassen. Delegation ist umständlicher zu implementieren, als Vererbung. Für alle Features, die wiederverwendet werden sollen, müssen Methoden implementiert werden, die die Anfragen an diese Features delegieren. Dennoch hat die Delegation gegenüber der Vererbung viele Vorteile.

Da das Delegationsobjekt Instanz unterschiedlicher Klassen sein kann, besteht die Möglichkeit, zur Laufzeit eine andere Implementation auszuwählen.

Eine Klasse kann die Implementation einer anderen Klasse wiederverwenden, ohne die gleiche Schnittstelle repräsentieren zu müssen. Es ist möglich, mit der neuen Klasse nur einen Teil des Funktionsumfangs der wiederverwendeten Klasse zu repräsentieren ohne die Kapselung aufzubrechen. Es läßt sich auch eine völlig neue Schnittstelle definieren. In diesem Falle übersetzen die delegierenden Methoden die Anfragen so, daß sie zu der Schnittstelle der wiederverwendeten Klasse passen.

1.1.3 Generische Klassen

Eine dritte Möglichkeit der Wiederverwendung, die auch Eiffel bereitstellt, ist die Anwendung von generischen Klassen. Bei diesem Konzept müssen bei der Definition der Klasse noch nicht alle Typen spezifiziert sein. Die unspezifizierten Typen werden erst bei der Verwendung der Klasse als Parameter bereitgestellt. Da generische Klassen weder bei Design Patterns noch in dieser Diplomarbeit eine Rolle spielen, wird hier nicht weiter darauf eingegangen.

1.2 Design Patterns

Das Erstellen eines flexiblen wiederverwendbaren Entwurfs erfordert viel Erfahrung. Beim Design unterschiedlicher Applikationen kann man oft ähnlich gelagerte Probleme finden, die sich durch die gleichen Klassenstrukturen lösen lassen. In [1] wurden 23 solcher Strukturen, die Lösungen häufig auftretender Designprobleme darstellen, isoliert. Diese Sammlung nimmt es dem Programmierer teilweise ab, gute Designideen immer wieder selbst finden zu müssen. Er muß nur anhand seiner vorgegebenen Problematik versuchen, das geeignetste Design Pattern herauszufinden.

1.2.1 Beschreibung von Design Patterns

Alle Pattern werden in [1] nach dem gleichen Muster beschrieben.

Mustername und Klassifizierung: Jedes Pattern hat einen repräsentativen Namen, der kurz und präzise sein Wesen wiedergibt. Zusätzlich wird es nach der Aufgabe in Erzeugungsmuster, Strukturmuster und Verhaltensmuster und nach dem Gültigkeitsbereich in klassenbasierte und objektbasierte Muster klassifiziert.
Zweck: Welche Fragestellungen und Probleme behandelt dieses Muster.
Auch bekannt als: Hier werden, wenn vorhanden, andere geläufige Namen dieses Patterns aufgezählt.
Motivation: Das Entwurfsproblem wird an einem Szenario beschrieben.
Anwendbarkeit: Dieser Abschnitt beschreibt, wie man die Situationen erkennt, in denen das Pattern anwendbar ist.
Struktur: Strukturdiagramme auf Basis von OMT und Interaktionsdiagramme beschreiben den Aufbau des Patterns.
Teilnehmer: Es werden die an dem Pattern beteiligten Klassen und deren Zuständigkeiten beschrieben.
Interaktionen: In diesem Abschnitt wird verdeutlicht, wie die Teilnehmer zur Erfüllung der gemeinsamen Aufgabe zusammenarbeiten.
Konsequenzen: Dieser Abschnitt diskutiert die Vor- und Nachteile, die der Einsatz dieses Patterns mit sich bringt.
Implementierung: Hier werden Techniken erklärt und Tips gegeben, wie das Pattern implementiert werden kann. Ebenfalls werden sprachspezifische Aspekte behandelt.
Beispielcode: Im Buch sind Codefragmente in C++ oder Smalltalk vorhanden.
Bekannte Anwendungen: Es werden echte Systeme aufgezählt, in denen das Pattern Anwendung findet.
Verwandte Muster: Dieser Abschnitt diskutiert Unterschiede zu anderen Pattern, aber auch, wie dieses Pattern mit anderen kombinierbar ist.

Der Einsatz von Design Patterns vereinfacht dem Programmierer die Auswahl der geeigneten Wiederverwendungsmechanismen. In Design Patterns wird intensiv von dem Konzept der Komposition Gebrauch gemacht. So wird dem Programmierer dieses zunächst umständlichere, aber flexiblere Konzept näher gebracht. Die Vererbung wird als Mittel der Implementationswiederverwendung nur sehr vorsichtig eingesetzt. Vielmehr wird die Vererbung zur Definition gemeinsamer Schnittstellen und für den Einsatz von Polymorphismus verwendet. Von Mehrfachvererbung wird nur selten Gebrauch gemacht und generische Klassen werden gar nicht verwendet.

1.2.2 Grenzen bei der Implementierung von Design Patterns

Design Patterns ermöglichen einen sehr guten objektorientierten Entwurf. Dennoch zeigen sich gerade im Umgang mit Pattern bei objektorientierten Programmiersprachen Grenzen auf.

Identifikation des Patterns im Quelltext: Der Einsatz von Design Patterns soll die Wartbarkeit der Programme erhöhen. Nun sind Pattern ohne die Beschreibungen in [1] nicht immer leicht zu verstehen. Ist ein Pattern erst einmal in die Klassenstruktur eines Programmes integriert, so ist es nur noch schwer als Einheit identifizierbar. Das flexible Design ist für den Programmierer, der mit der Wartung beauftragt ist, wenig von Nutzen, wenn er das Pattern nicht erkennt und es somit nicht versteht.
Kapselung auf Patternebene: Objektorientierte Programmiersprachen bieten die Möglichkeit, die physische Repräsentation einer Klasse zu kapseln. Lediglich die Schnittstelle ist nach außen sichtbar. Gerade bei einigen Strukturmustern wäre es sinnvoll, die Struktur des Patterns zu verbergen und lediglich eine Schnittstelle nach außen zur Verfügung zu stellen. So könnte man eine höhere Modularität der Programme erreichen. Es ist aber meist nur möglich, jede Klasse für sich zu kapseln. Lediglich in der Sprache Java ab Version 1.1 kann man gekapselte Pattern unter Nutzung des Konzepts der inneren Klassen implementieren.

Verwendung von vorimplementierten Pattern: In objektorientierten Sprachen kann man vorimplementierte Klassenbibliotheken als Mittel der Wiederverwendung nutzen. Diese bewirken Zeitersparnis bei der Implementation und einen einheitlicheren Programmierstil bei unterschiedlichen Programmierern. Will man die Klassenbibliothek nutzen, so verwendet man die Klassen direkt, indem man einfach Objekte instanziiert oder man paßt die Klassen noch an die jeweilige Situation an, indem man von ihnen erbt. Wollte man Pattern aus einer vorimplementierten Patternbibliothek nutzen, so müßte man sie in jedem Fall noch an das zu entwerfende Programm anpassen, da Design Patterns nur sehr abstrakte Codefragmente darstellen. Es ist jedoch keine der drei in Abschnitt 1.1 angegebenen Möglichkeiten zur Wiederverwendung von Klassen mächtig genug, ganze Patterns oder andere Strukturen wiederzuverwenden. Die Abbildungen 1.1 und 1.2 verdeutlichen, warum Klassenvererbung kein geeignetes Mittel zur Wiederverwendung von Klassenstrukturen ist. Die Objektkomposition ist erst recht kein Mittel, da dadurch die Klassenstrukturen vollständig aufgebrochen werden. Mit generischen Klassen lassen sich einfache Klassenstrukturen wiederverwenden. Davon wird in vielen Klassenbibliotheken auch häufig Gebrauch gemacht. Generische Klassen bieten die Möglichkeit, Kompositionsbeziehungen wiederzuverwenden, scheitern aber auch, wenn Vererbungsbeziehungen ins Spiel kommen.

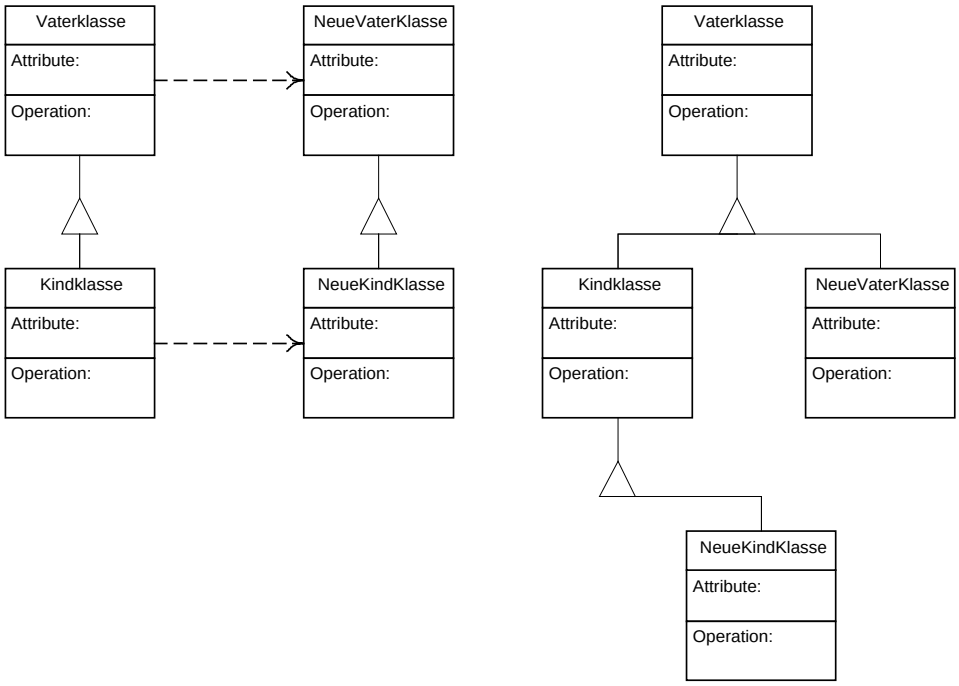

Abbildung 1.1: *Links: Gewünschte Wiederverwendung einer Vererbungsbeziehung*
Rechts: Fehlgeschlagener Versuch der Wiederverwendung durch Klassenvererbung

Die Abbildung 1.1 verdeutlicht, daß es nicht möglich ist, die Vererbung als Mittel der Wiederverwendung von Klassenstrukturen, in denen die Klassen selbst in Vererbungsbeziehungen stehen, zu nutzen. Die Ausgangsklassen `Vaterklasse` und `Kindklasse` stehen in einer Vererbungsbeziehung. Nach der Wiederverwendung und Anpassung der Klassen sollen die entstehenden Klassen `NeueVaterKlasse` und `NeueKindKlasse` ebenfalls in voneinander erben. Der Versuch, die Wiederverwendung durch Vererbung zu realisieren, schlägt fehl. Wenn `NeueVaterKlasse` von `Vaterklasse` und `NeueKindKlasse` von `Kindklasse` erbt, stehen `NeueVaterKlasse` und `NeueKindKlasse` nicht mehr in einer Vererbungsbeziehung. Das Konzept der Polymorphie ist in diesem Fall nicht mehr

anwendbar. Objekte der Klasse `NeueKindKlasse` können nicht Variablen der Klasse `NeueVaterKlasse` zugewiesen werden.

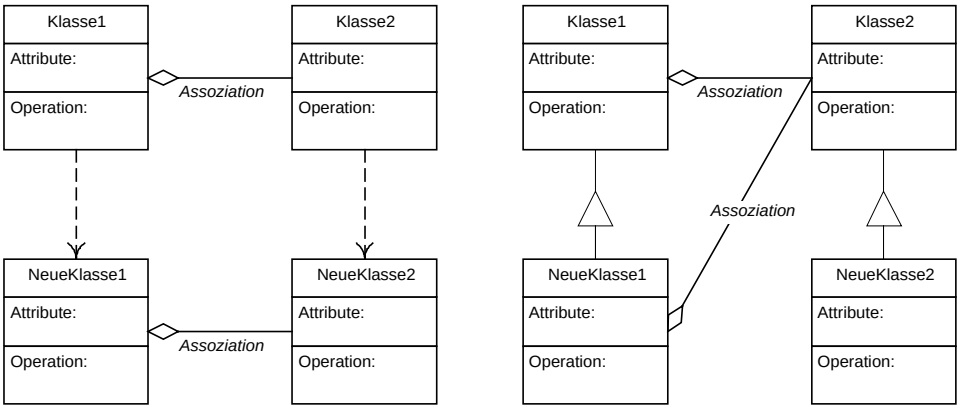

Abbildung 1.2: *Links: Gewünschte Wiederverwendung einer Assoziationsbeziehung*
Rechts: Versuch der Wiederverwendung dieser Struktur durch Klassenvererbung

Die Abbildung 1.2 demonstriert die Wiederverwendung zweier Klassen `Klasse1` und `Klasse2`, die in einer Assoziationsbeziehung stehen. Nach der Wiederverwendung soll `NeueKlasse1` direkt auf `NeueKlasse2` verweisen. Der Versuch, dieses Ziel durch Vererbung zu erreichen, gelingt nur bedingt. Es ist zwar auch nach der Vererbung möglich, daß Instanzen von `NeueKlasse1` auf Instanzen von `NeueKlasse2` verweisen, sie werden aber als Objekte von `Klasse2` gesehen. Der Verweis ist somit unnötig allgemein. Wurde die Schnittstelle von `Klasse2` zu `NeueKlasse2` erweitert, so ist der Zugriff über diesen Verweis auf den neuen Funktionsumfang ohne Typanpassung nicht möglich.

Dieses Problem läßt sich in objektorientierten Sprachen noch mit dem Konzept der generischen Klassen lösen. Dabei wird `Klasse1` mit `Klasse2` parametrisiert und `NeueKlasse1` mit `NeueKlasse2`.

Kapitel 2

Die patternorientierte Sprache *PaL*

Da viele Nachteile der objektorientierten Programmiersprachen den leichten Umgang mit Design Patterns behindern und deren positive Effekte schmälern, liegt die Idee nahe, eine patternorientierte Programmiersprache zu entwickeln. In diesem Kapitel wird ein patternorientiertes Programmiermodell entworfen und dazu die Syntax für die patternorientierte Sprache *PaL* entwickelt.

2.1 Entwicklung eines patternorientierten Programmiermodells

2.1.1 Forderungen an ein patternorientiertes Programmiermodell

Aus den aufgeführten Nachteilen von objektorientierten Programmiersprachen im Umgang mit Design Patterns lassen sich folgende Forderungen an ein patternorientiertes Programmiermodell ableiten.

Identifizierbarkeit des Patterns im Quelltext: Das Pattern und die dazugehörigen Teilnehmerklassen, von nun an als **Komponenten** bezeichnet, sollen auch noch im Quelltext am Namen des Patterns erkennbar sein.

Kapselung auf Patternebene: Ein Pattern soll mit seinen Komponenten eine Einheit bilden. Patterns sollen wie Objekte in objektorientierten Programmiersprachen instanziierbar sein. Eine **Patterninstanz** bildet zur Laufzeit eine konkrete Repräsentation eines Patterns. Die Struktur des Patterns soll vor anderen Patterninstanzen verborgen bleiben.

Übergeordnetes Verhalten: Die Definition eines den Komponenten übergeordneten Verhaltens, z.B. für die Initialisierung des Patterns, soll möglich sein. Dieses übergeordnete Verhalten stellt die Schnittstelle des Patterns zu anderen Pattern dar. Die Attribute und Methoden des Patterns, die das übergeordnete Verhalten implementieren, werden von nun an als **Patternfeatures** bezeichnet.

Verwendbarkeit von vorimplementierten Pattern: Alle Design Patterns sollen unabhängig von konkreten Aufgabenstellungen implementierbar sein, sich aber auf einfache Art und Weise für ein konkretes Problem anwenden lassen, indem sie sich für diese Problemstellung verfeinern lassen. Zur **Verfeinerung** zählen die Vervollständigung des Patterns, Umbenennungen von Komponenten, Komponentenfeatures und Patternfeatures und die Neuimplementation von Komponentenfeatures und Patternfeatures.

Polymorphismus auf Patternebene: Ähnlich, wie bei Objekten von Unter- und Oberklassen, sollen sich Instanzen eines Patterns durch die Instanzen der davon verfeinerten Pattern substituieren lassen. Patterninstanzen sollen sich bezüglich ihres übergeordneten Verhaltens polymorph verhalten.

Kombinierbarkeit von Pattern: Verwandte Pattern sollen leicht miteinander kombinierbar sein. Hierfür soll das Mittel der **Mehrfachverfeinerung** genutzt werden, die vergleichbar zur Mehrfachvererbung in objektorientierten Programmiersprachen sein soll. Zwei Pattern werden kombiniert, indem sie beide zu einem einzigen neuen Pattern verfeinert werden. Eine Komponente eines aus zwei Ausgangspattern kombinierten Patterns soll Komponenten beider Ausgangspattern repräsentieren können. Die Komponente beinhaltet dann die Features beider Ausgangskomponenten. Kommt es zu Namenskonflikten, soll die Implementation eines Features ausgewählt werden können.

2.1.2 Das Patternmodell und seine grafische Repräsentation

Die Struktur eines objektorientierten Programms läßt sich durch ein grafisches Klassenmodell leichter verdeutlichen. Dieser Gedanke läßt sich auf das patternorientierte Programmiermodell übertragen. Ein Pattern wird durch die folgenden Elemente grafisch repräsentiert:

Rahmen: Die Abgrenzung eines Patterns zu anderen wird durch einen Rahmen verdeutlicht.
Name: Der Name des Patterns steht oben innerhalb des Rahmens.
Komponentenstruktur: Unter dem Namen findet die Klassenstruktur des Patterns in UML-Notation Platz.
Patternfeatures: Unter der Komponentenstruktur werden die Features des Patterns aufgelistet.

Jetzt gilt es noch, die Beziehungen zwischen Pattern darzustellen. Dazu zählen, wie in den Forderungen herausgearbeitet, Referenzen auf andere Patterninstanzen, also Patternkomposition, und die Verfeinerung von Pattern.

Patternkomposition: Die Komposition soll funktionieren, wie die Objektkomposition in objektorientierten Sprachen. Daher werden für die Komposition im Patternmodell die gleichen Zeichen verwendet, wie bei objektorientierten Modellen. Es wird auf die UML-Notation zurückgegriffen.

Verfeinerung: Die Verfeinerung von Pattern ist komplizierter, als die Vererbung in Objektorientierten Programmiersprachen. Daher ist auch die Notation der Verfeinerung etwas umfangreicher. Die Verfeinerung wird durch einen gestrichelten Pfeil dargestellt. Den Verfeinerungsbeziehungen können Namen zugeordnet werden. Dieser Name ist vor allem dann relevant, wenn ein Pattern auf mehr als eine Art zu einem neuen Pattern verfeinert wird. Daher kann der Pfeil zur Unterscheidung der Verfeinerungen mit diesem Namen beschriftet werden. Verfeinerungspfeile lassen sich von Pattern zu Pattern, von Patternfeature zu Patternfeature, von Komponente zu Komponente und von Komponentenfeature zu Komponentenfeature einzeichnen. Um die Anzahl der Pfeile auf ein übersichtliches aber aussagekräftiges Maß zu reduzieren, wird empfohlen, nur die Pattern und die sich ändernden Elemente zu verbinden.

Die Abbildungen 2.1 bis 2.3 demonstrieren die grafische Notation von Pattern in Verfeinerungsbeziehungen. Links in der Abbildung 2.1 ist z.B. das Pattern `PCOMPOSITE` sichtbar. Es besitzt die Komponenten `COMPONENT`, `COMPOSITE` und `LEAF` und das Patternattribut `the_component`. Pattern- und Komponentennamen werden in Großbuchstaben dargestellt und Pattern- und Komponentenfeatures in Kleinbuchstaben. Patternnamen beginnen zur besseren Unterscheidung von den Komponentennamen mit einem vorangestellten „P“. Zusammengesetzte Bezeichner werden mit einem Unterstrich getrennt. Aus Gründen der Übersichtlichkeit werden in den Grafiken dieser Arbeit keine Typen bei Attributen und Funktionen angegeben.

In der Abbildung 2.1 wird eine einfache Verfeinerungsbeziehung dargestellt. Daher ist es nicht nötig, der Verfeinerung einen Namen zu geben. Um die Übersicht zu wahren, wurde die Darstellung der Verfeinerung durch Pfeile auf ein Minimum reduziert. In diesem Fall sind das die Umbenennungen von `COMPONENT` auf `GRAPHIC`, von `COMPOSITE` auf `COMPOSED_GRAPHIC` und von `the_component` auf `the_graphic`. In jedem Fall werden die Pattern, die miteinander in einer Vererbungsbeziehung stehen, durch einen Pfeil verbunden.

Die Abbildung 2.2 zeigt eine Mehrfachverfeinerung, in der das Komponentenfeature `operation` zu `draw` und `resize` vervielfältigt wird. Um die Verfeinerungen unterscheiden zu können, bekommen sie einen Namen. Eine einfache und übersichtliche Art der Benennung der Verfeinerung ist die Bezeichnung mit `Ref_` + dem Element, das vervielfältigt wurde. In diesem Fall heißen die Verfeinerungen also

`Ref_draw` und `Ref_resize`. Die Umbenennung von `operation` in der Komponente `GRAPHIC` impliziert die Umbenennung des Features in `LEAF` und `COMPOSED_GRAPHIC`. Zwischen diesen Komponenten werden daher keine weiteren Verfeinerungspfeile eingezeichnet.

In der Abbildung 2.3 findet ebenfalls eine Mehrfachvererbung statt. In diesem Fall wird die Komponente `LEAF` zu `CIRCLE` und `SQUARE` dupliziert. Die Verfeinerungen werden daher mit `Ref_CIRCLE` und `Ref_SQUARE` bezeichnet. Es werden wieder nur die sich ändernden Elemente durch Pfeile verbunden.

Wie schon weiter oben beschrieben, ist die Verfeinerung das Mittel zur Anwendung abstrakter vorimplementierter Design Patterns. Da sich die praktischen Anwendungen in der Anzahl der Komponenten und deren Features unterscheiden, läßt sich ein Pattern oft erst nach mehreren Verfeinerungsschritten praktisch einsetzen. Die Abbildungen 2.2 bis 2.4 beschreiben die Verfeinerungsschritte vom abstrakten Kompositum zum praktisch einsetzbaren Grafik-Kompositum. Diese Schrittfolge läßt sich auf viele Design Patterns übertragen.

Schritt 1: Mit dem ersten Schritt werden alle diejenigen Elemente verfeinert, bei denen vom abstrakten Ausgangspattern zum praktisch einsetzbaren Pattern eine eins-zu-eins-Beziehung besteht. Bei dem Pattern Kompositum werden die Komponenten `COMPONENT` und `COMPOSITE` in `GRAPHIC` und `COMPOSED_GRAPHIC` umbenannt.

Schritt 2: Jetzt muß man sich entscheiden, ob man zuerst die Methode `operation` oder die Komponente `LEAF` vervielfältigt. Diese Entscheidung kann in einigen Fällen die Komplexität der Implementation späterer Erweiterungen beeinflussen. In diesem Beispiel wurde sich für das Duplizieren der Methode entschieden. Hierfür wird das Mittel der Mehrfachverfeinerung angewendet. Das Pattern `PGRAPHIC_COMPOSITE` wird zweimal verfeinert. Die Methode `operation` erhält bei der einen Verfeinerung den neuen Namen `draw` und bei der anderen den neuen Namen `resize`. Für die Komponente `COMPOSED_GRAPHIC` können diese Methoden sofort implementiert werden.

Schritt 3: Um die Komponente `LEAF` zu vervielfachen, wird wieder zweimal dasselbe Pattern verfeinert. Die Komponente `LEAF` wird dabei einmal in `CIRCLE` und einmal nach `SQUARE` umbenannt. Die Methoden `draw` und `resize` dieser beiden Komponenten werden in diesem Verfeinerungschritt implementiert.

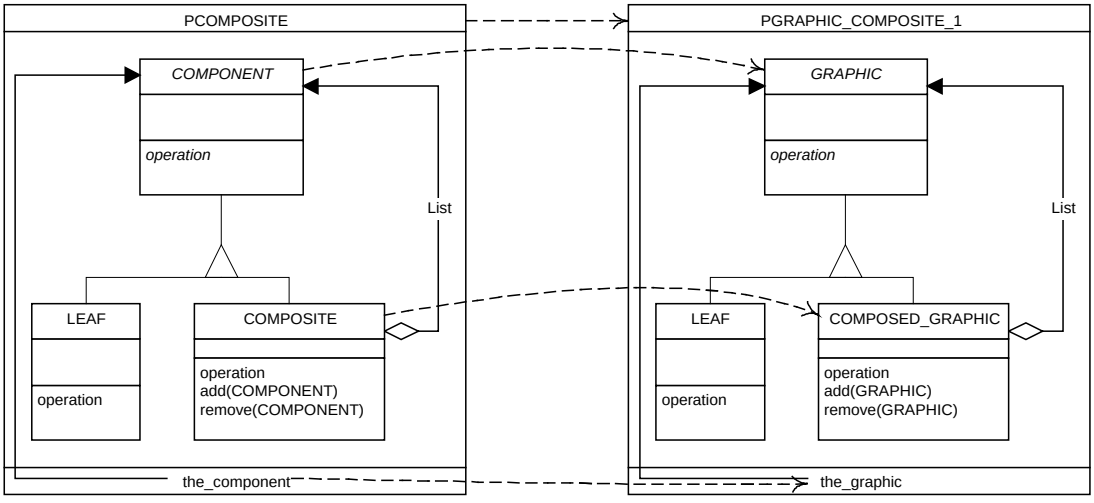

Abbildung 2.1: *Schritt1: Umbenennungen*

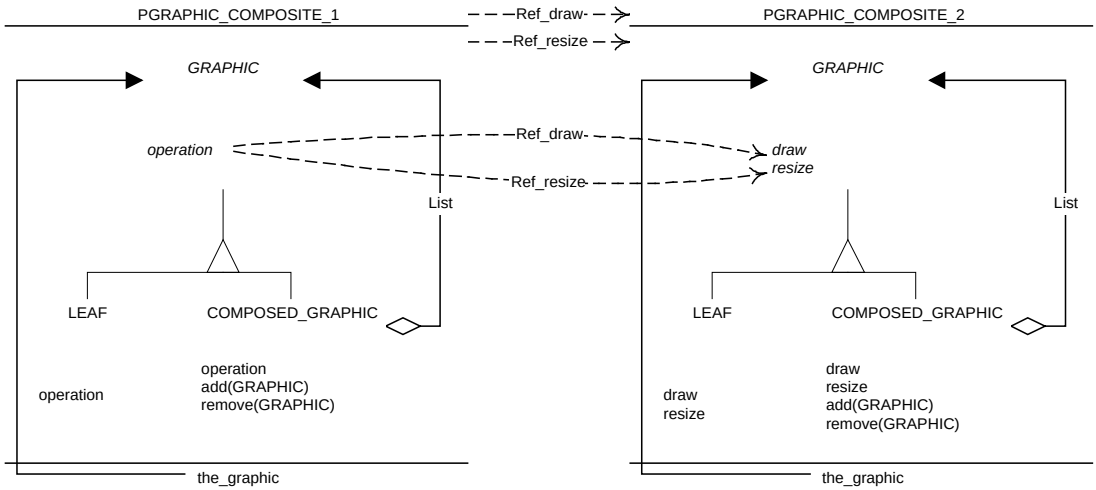

Abbildung 2.2: *Schritt 2: Vervielfachung von operation*

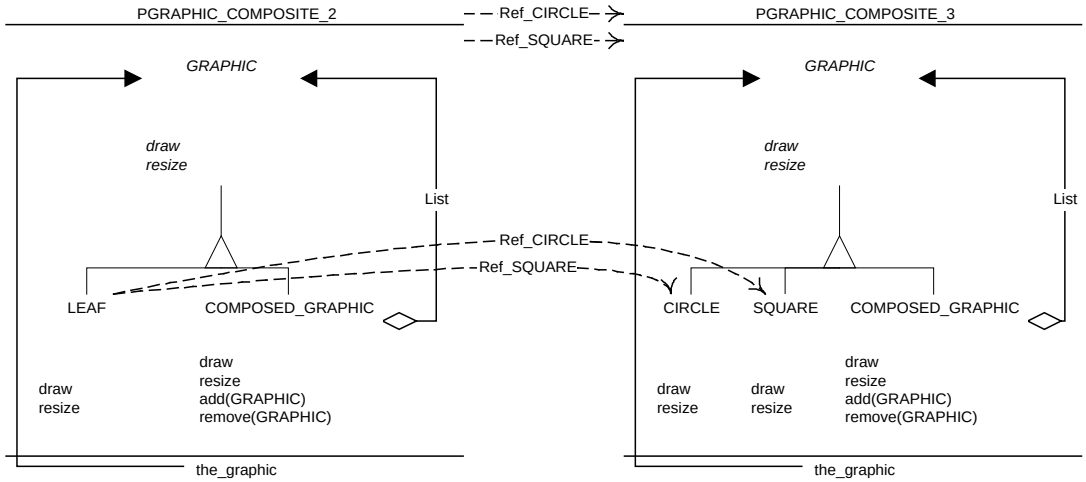

Abbildung 2.3: *Abbildung 2.4: Schritt 3: Vervielfachung von LEAF*

2.2 Die Syntax der patternorientierten Sprache *PaL*

Da noch keine Erfahrungen mit dem patternorientierten Programmiermodell vorliegen, kann der erste Entwurf einer patternorientierten Sprache auch nur ein Prototyp sein. Mit ihm lassen sich aber Erfahrungen über die tatsächliche Einsetzbarkeit sammeln und mögliche oder sogar nötige Verbesserungen finden. Da das patternorientierte Modell auf dem objektorientierten Modell aufbaut, liegt der Gedanke nahe, die Syntax der Sprache *PaL* auf der Basis der Syntax einer objektorientierten Sprache aufzubauen. Hierfür wurde die objektorientierte Sprache Eiffel ausgewählt. Dieser Ansatz hat den Vorteil, daß es schon eine solide und erprobte Basis gibt und man die Überlegungen hauptsächlich auf die neuen Sprachelemente konzentrieren kann. Ein weiterer Vorteil ist, daß sich so auf relativ leichte Art und Weise ein Präcompiler in die Sprache Eiffel schreiben läßt. Zu den Nachteilen zählt, daß man sich durch die Vorgaben von Eiffel zu sehr beeinflussen läßt und den Aufbau der Sprache nicht auf die Anwendung, sondern auf die Übersetzung hin konstruiert. Zudem ist Eiffel für sich schon eine sehr komplexe Sprache mit vielen Konzepten, die bei einer Erweiterung berücksichtigt werden müssen, was eine Erweiterung kompliziert.

2.2.1 Eiffel als Basis der Sprache *PaL*

Auf Eiffel als Basissprache fiel die Entscheidung schon in einer sehr frühen Phase dieses Projektes. Gründe dafür waren die Möglichkeit der Mehrfachvererbung, die Umbenennungsmechanismen für Features und das Konzept der Co-Varianz. Die Möglichkeiten der Umbenennung von Features erwiesen sich im späteren Design der Sprache als nicht mehr relevant, da eine Umbenennung von Komponentenfeatures bei der Vererbung der Komponenten nicht implementiert wurde.

Die Syntax von Eiffel kommt an mehreren Stellen zum Einsatz:

Definition der Komponentenstruktur: Da die Struktur der Teilnehmerklassen objektorientiert ist, kann die Eiffelsyntax nahezu unverändert zum Einsatz kommen. Es sind nur einige Befehle zum Manipulieren der Komponenten in einer Verfeinerung notwendig. Um den Aufwand etwas geringer zu halten, wird auf einige Eigenschaften von Eiffel verzichtet. Dazu zählen hauptsächlich die Zusicherungsmechanismen, wie Vor- und Nachbedingungen, Umbenennungen von Features in einer Vererbungsbeziehung und generische Klassen. Der Einsatz des Patternmodells stellt jedoch ein mächtigeres Mittel als das der generischen Klassen dar.

Definition eines Patterns: Zur Definition eines Patterns wird sich nur grob an der Syntax von Eiffel orientiert, um der Sprache ein einheitliches Bild zu geben. Hier sind zusätzliche Sprachkonstrukte für die Definition einer Verfeinerung notwendig. Die Implementation der Patternfeatures geschieht aber wieder nahezu in Eiffelsyntax.

Definition der Zielsprache: Das Ergebnis der Übersetzung von *PaL* ist reines Eiffel. Da in der Quellsprache die Zusicherungsmechanismen, die Umbenennungsmechanismen und die generische Klassen von Eiffel nicht genutzt werden, kommen sie auch nach der Übersetzung nicht zum Einsatz.

Das Buch [4] diente als Quelle für die Syntaxregeln von Eiffel. Existierende Eiffelcompiler sind flexibler und lassen unterschiedliche Zeichensetzungen (z.B. beim Semikolon) zu. Die Sprache *PaL* akzeptiert nur die Art der Zeichensetzung, wie sie in diesem Buch beschrieben ist.

2.2.2 Die Syntaxregeln von *PaL*

Die vollständigen Syntaxregeln der Sprache *PaL* sind in EBNF-Notation im Anhang zu finden. Hier werden nur die Regeln erläutert, die für das Verständnis der Sprache wichtig sind, und die Regeln, an denen bedeutende Einschränkungen gegenüber Eiffel vorgenommen wurden.

Die Deklaration eines Patterns wurde grob der Deklaration einer Klasse in Eiffel nachempfunden. Die Verfeinerungsklausel `refine` ersetzt dabei das Vererbungskonstrukt `inherit` in Eiffel. Die Klausel `creation` besitzt die gleiche Semantik, wie in Eiffel. Hier werden die Methoden des Patterns aufgezählt, mit denen sich das Pattern instanziieren läßt. Danach werden die Komponenten des Patterns deklariert, was im wesentlichen einer Klassendeklaration unter Eiffel entspricht. Die Klauseln für Patternfeatures entsprechen ebenfalls im wesentlichen den Featureklauseln unter Eiffel. Sie lassen sich aber zusätzlich als `intern` oder `extern` deklarieren. Ohne diese Deklaration gelten die Patternfeatures automatisch als `intern`. Die externen Patternfeatures definieren die Schnittstelle zu anderen Patterninstanzen. Für das Pattern selbst und seine Komponenten sind sowohl die internen Patternfeatures, als auch die externen Patternfeatures sichtbar. Bei `creation` dürfen daher nur externe Patternfeatures aufgezählt werden.

```
<PATTERNDECL> ::=
pattern <PATTERNNAME>
              [ refine { <REFINE> }+ ]
              [ creation <PATTERNFEATURENAME> { , <PATTERNFEATURENAME> }* ]
              { component <COMPONENT>}*
              { [ intern | extern ] feature <PATTERNFEATURE>}*
       end
```

Beispiel:

(Siehe auch Abbildung 2.1)

```
pattern PCOMPOSITE

        component COMPONENT
                ...
        end

        component COMPOSITE
                ...
        end

        component LEAF
                ...
        end

        intern the_component: COMPONENT

end
```

Bei der Verfeinerung von mehreren Pattern kann es zu Namenskonflikten bei Patternfeatures und Komponentenfeatures kommen. Es gibt in *PaL* zwei Möglichkeiten, diese Konflikte aufzulösen. Entweder kann man das entsprechende Feature in dem neuen Pattern neu implementieren oder man selektiert die Implementation aus genau einer Verfeinerung. Eine Selektion kann immer angegeben werden, sie wird nur im Falle eines Namenskonflikts ausgewertet. Man kann die verfeinerten Features einzeln oder gesammelt selektieren.

Im Konstrukt `refine` gibt es zwei Möglichkeiten zum Selektieren von Features. Steht ein `select` vor dem Patternnamen des Patterns, das verfeinert wird, so bedeutet das, daß alle Patternfeatures und Komponentenfeatures dieser Verfeinerung selektiert sind. Die zweite Möglichkeit ist das `select` Konstrukt innerhalb der Verfeinerungsklausel. Hier kann man Patternfeatures oder Komponenten und somit alle zugehörigen Komponentenfeatures selektieren. Eine dritte Möglichkeit zur direkten Selektion von Komponentenfeatures gibt es mit dem Konstrukt `cast`, das weiter unten erläutert wird.

Da später im Quelltext mit dem Konstrukt `cast` noch einmal auf diese Verfeinerung zurückgegriffen werden kann, bekommt sie einen eindeutigen Namen. Dieser wird entweder im refine Konstrukt hinter dem Namen des Patterns, das verfeinert wird, in der Form `<IDENTIFIER>` angegeben, oder er ist gleich dem Patternnamen.

Mit dem `rename` Konstrukt kann man in einer Verfeinerung Komponenten und interne Patternfeatures umbenennen. Auch wenn in Eiffel generell Umbenennungen von Features in einer Vererbungsbeziehung gestattet sind, wurde hier auf die Möglichkeit der Umbenennung von externen Patternfeatures verzichtet, da dies ein deutlich komplexeres Verfeinerungsmodell erfordert.

```
<REFINE> ::=
        [ select ] <PATTERNNAME> [ <PATTERNREF> ] [
                [ rename <RENAME> { , <RENAME> ) }* ]
                [ select <COMPONENT_OR_FEATURENAME> { , <COMPONENT_OR_FEATURENAME> ] }* ]
        end ]

<PATTERNREF> ::=
        '<' <IDENTIFIER> '>'

<RENAME> ::=
        <COMPONENT_OR_FEATURENAME> as <COMPONENT_OR_FEATURENAME>
```

Beispiel:

(Siehe auch Abbildung 2.1)

```
pattern PGRAPHIC_COMPOSITE_1

        refine
                PCOMPOSITE
                        rename
                                COMPONENT as GRAPHIC,
                                COMPOSITE as COMPOSED_GRAPHIC,
                                the_component as the_graphic
                end

end
```

Der Hauptunterschied zwischen Komponenten und Eiffelklassen ist das neue Konstrukt `cast`. Damit lassen sich die Featurenamen in einer Verfeinerung anpassen. Die Konstrukte `inherit`, `creation` und `feature` entsprechen denen in Eiffel, wobei `inherit` stark vereinfacht wurde. Alle anderen Eiffelkonstrukte zur Klassendeklaration wurden zur Vereinfachung weggelassen.

```
<COMPONENT> ::=
        <COMPONENTNAME>
                [ cast <CAST> ]
                [ inherit { <INHERIT> }+ ]
                [ creation <COMPONENTFEATURENAME> { , <COMPONENTFEATURENAME> }* ]
                { feature <COMPONENTFEATURE> }*
        end
```

Das Konstrukt `cast` beinhaltet die optionalen Konstrukte `rename` und `select`. Mit `rename` kann man Komponentenfeatures aus einer ganz bestimmten Verfeinerung umbenennen, indem man den Namen dieser Verfeinerung angibt. Mit dieser Funktionalität kann man Features duplizieren oder Namenskonflikte vermeiden. Gibt man den Namen der Verfeinerung nicht an, so wird das entsprechende Komponentenfeature aus allen Verfeinerungen umbenannt. Mit dem `select` Konstrukt kann man, wie weiter oben schon angesprochen, die Implementation spezieller Features einer Verfeinerung auswählen. Dazu muß der Name des Komponentenfeatures und der Name der Verfeinerung angegeben werden.

```
<CAST> ::=
        [ rename <RENAMECOMPONENTFEATURE> {, <RENAMECOMPONENTFEATURE> }*]
        [ select <CASTSELECT> { , <CASTSELECT> }*]
        end

<RENAMECOMPONENTFEATURE> ::=
        <COMPONENTFEATURENAME> [ from <PATTERNREF> ] as <COMPONENTFEATURENAME>

<CASTSELECT> ::=
        <COMPONENTFEATURENAME> from <PATTERNREF>
```

Beispiel:

(Siehe auch Abbildung 2.2)

```
pattern PGRAPHIC_COMPOSITE_2

        refine
                select PGRAPHIC_COMPOSITE_1 <Ref_draw>
                end

                PGRAPHIC_COMPOSITE_1 <Ref_resize>
                end
```

```
        component GRAPHIC

                cast
                        rename
                                operation from <Ref_draw> as draw,
                                operation from <Ref_resize> as resize
                end

        end

end
```

Die `inherit` Klausel wurde im Vergleich zu Eiffel stark vereinfacht. Ein `rename` innerhalb einer Vererbung wird von *PaL* nicht unterstützt. Daher wird auch das `select` überflüssig. Das Konstrukt `export` wird ebenfalls nicht unterstützt. Das Überschreiben von Features ist in *PaL* durch einfache Neuimplementation möglich. Die Ankündigung durch das `redefine` Konstrukt ist nicht notwendig.

```
<INHERIT> ::=
        <COMPONENTNAME> { ; <COMPONENTNAME> }
```

Die Implementation der Features geschieht ohne Angabe von Klienten, ohne Vor- und Nachbedingungen oder andere Spezialitäten von Eiffel. In diesem Prototypen wurden vorerst folgende Instruktionen realisiert: Instanziierung, Bedingung, Schleife, Aufruf eines Features, Zuweisung und bedingte Zuweisung mit Typüberprüfung.

```
<INSTRUCTION> ::=
        ( <CREATION>
        | <IF>
        | <LOOP>
        | <CALL>
        | <LET>
        | <TRY> )
```

Beim Erzeugen von Komponenten- und Patterninstanzen kann man nicht explizit den zu erzeugenden Typ angeben. Es wird automatisch eine Instanz des Typs der zu beschreibenden Variablen erzeugt.

```
<CREATION> ::=
        !! <WRITABLE>.<LOCALCALL>
```

Der Aufruf eines Patternfeatures ist noch nicht vollständig orthogonal implementiert. Daher ist ein Call noch nicht vom Ergebnis eines beliebigen Ausdrucks aus möglich.

```
<CALL> ::=
        ( <LOCALCALL>
        | <LOCAL>.<LOCALCALL> )

<LOCALCALL> ::=
        <PATTERNORCOMPONENTFEATURE> ['(' EXPRESSION { , EXPRESSION }* ')' ]
```

Die Deklaration einer Variablen oder eines Übergabeparameters mit dem Typ des aktuellen Patterns ist auf zwei Arten möglich. Zum einen kann man als Typ den Namen des aktuellen Patterns angeben. Dieser Typ wird im Falle einer Verfeinerung nicht mitverfeinert. Diese Variante eignet sich z.B. für die Deklaration von Übergabeparametern externer Patternfeatures. Eine zweite Möglichkeit ist die Angabe des Typs `currentpattern`. Bei dieser Variante wird die Typdeklaration im Falle einer Verfeinerung mit spezialisiert. Der Typ ist immer der des aktuellen Patterns. Diese Methode eignet sich hauptsächlich für patterninterne Übergabeparameter. In Eiffel gibt es das vergleichbare Konstrukt `like current`, welches immer den Typ der aktuellen Klasse bedeutet. Dieses Konstrukt gibt es in *PaL* nicht. Für Komponenten wurde das vergleichbare Konstrukt `currentcomponent` angedacht, aber noch nicht vollständig implementiert.

```
<TYPE> ::=
       ( currentpattern | {<IDENTIFIER> )
```

Kapitel 3

Die Werkzeuge für die Realisierung

Um einen Compiler zu schreiben, benötigt man die geeigneten Werkzeuge. Da es sich hier um einen Präcompiler handelt, ist zum einen ein geeignetes Tool zum Spezifizieren und Implementieren von Quelle-Quelle-Übersetzern und zusätzlich ein Eiffelcompiler gefragt. Der Übersetzungsprozeß von *PaL* in die Sprache Eiffel läßt sich grob in drei Arbeitsschritte aufteilen.

Parsen: Der *PaL*-Quelltext wird eingelesen und es wird ein abstrakter Syntaxbaum aufgebaut.
Transformieren: Der abstrakte Syntaxbaum der Sprache *PaL* wird in einen abstrakten Syntaxbaum der Sprache Eiffel transformiert.
Generieren: Aus dem neuen abstrakten Syntaxbaum wird Eiffel-Quelltext generiert.

Das System LDL bietet Formalismen, mit denen man diese Übersetzungsschritte spezifizieren kann. Als Eiffelcompiler wurde SmallEiffel gewählt.

3.1 LDL – Language Development Laboratory

Das System LDL ist in [5] ausführlich dokumentiert. Daher wird hier nur kurz auf die für dieses Projekt wesentlichen Eigenschaften dieses Systems eingegangen.

LDL ist ein Werkzeug zum Entwickeln beweisbar korrekter Prototypinterpreter und Prototypübersetzer. Es bietet unterschiedliche Formalismen zum Erstellen ausführbarer Spezifikationen. Durch die folgenden Stärken erweist sich LDL als hervorragend für dieses Projekt geeignet.

Definition der Sprache: Durch das Mittel der Spezifikation läßt sich die Sprache *PaL* auf präzise Art und Weise vollständig und korrekt definieren.
Definition der Semantik: Die Spezifikation der Transformation von *PaL* zu Eiffel definiert zusammen mit der Semantik der Sprache Eiffel die Semantik von *PaL*.
Ausführbarkeit der Spezifikation: LDL kann diese Spezifikationen interpretieren. Dadurch erhält man die Implementation eines Übersetzers, die sich auf das Wesentliche beschränkt. Die Korrektheit der Spezifikation impliziert die Korrektheit des Übersetzers.

Im Folgenden wird auf die Formalismen von LDL eingegangen, die für die Entwicklung eines Quelle-Quelle-Übersetzers von Bedeutung sind.

GS: Dieser Formalismus ermöglicht die Spezifikation von GSF-Schemata. GSF steht für Grammatiken syntaktischer Funktionen, eine Art attributierter Grammatiken.
Ein GSF-Schema besteht aus einer Menge kontextfreier attributierter Regeln, die mit relationalen Formeln verbunden sein können, welche Beziehungen zwischen den Attributen beschreiben.
In diesem Projekt kommt der GS-Formalismus mit Verfeinerung durch den LG-Formalismus (s.u.) für das Parsen des *PaL*-Quelltextes zum Einsatz. Es wird ein abstrakter *PaL*-Syntaxbaum aufgebaut. Zum Generieren des Eiffel-Quelltextes aus dem Eiffel-Syntaxbaum wird der GS-Formalismus durch den TG-Formalismus (s.u.) verfeinert.

LG: Mit dem LG-Formalismus (Lexical Grammar) können Scanner beschrieben werden. Es besteht die Möglichkeit, Zeichenmengen zusammenzufassen und zu benennen. Weiterhin kann man Morphemklassen durch reguläre Ausdrücke definieren. Solchen Klassen lassen sich Funktionen zuordnen, die eine erkannte Zeichenfolge in Werte eines entsprechenden Datentyps umwandeln. Diese Morphemklassen stellen Terminale in den Sprachen dar, die durch attributierte Grammatiken beschrieben werden. Die Erkennung von Schlüsselwörtern steht beim LG-Formalismus nicht im Mittelpunkt, diese erkennt LDL automatisch. Zusätzlich ist es in diesem Formalismus möglich, Aktionen für die Morphemklassen, wie z.B. das Herausfiltern von Leerzeichen, in die Scannerbeschreibung einzufügen.
Der LG-Formalismus wird in diesem Projekt genutzt, um den Aufbau von Strings, Zahlen und Identifiern zu definieren.

TG: Der TG-Formalismus ist das Gegenteil zum LG-Formalismus. Er dient zur Beschreibung der Erzeugung von Zeichenketten aus Terminalattributen. Dazu kann jedem Terminalattribut eine Umwandlungsfunktion zugeordnet werden.
In diesem Projekt werden Strings, Zahlen und Identifiern Umwandlungsfunktionen für die textuelle Darstellung zugewiesen.

IF: Mit dem IF-Formalismus können die Interfaces der angebundenen Funktionen beschrieben werden.
In dieser Arbeit werden ausschließlich vorimplementierte IF-Module aus der LDL-Bibliothek benutzt.

IR: Mit dem IR-Formalismus können Interferenzregeln zur Beschreibung von Transformationen und Semantik spezifiziert werden. Aus einer Regel läßt sich eine Schlußfolgerung ableiten, wenn alle Prämissen gültig sind. Wie im GS-Formalismus bestehen die Prämissen und Schlußfolgerungen aus parametrisierten Symbolen. Der Interpreter versucht auch hier, die Regeln entsprechend ihrer Reihenfolge in der Spezifikation anzuwenden.
Die gesamte Transformation von dem abstrakten *PaL*-Syntaxbaum in einen abstrakten Eiffel-Syntaxbaum wird durch IR-Regeln definiert.

RFD: Der RFD-Formalismus dient der Definition rekursiver Funktionen. Diese funktionale Sprache dient in LDL der Beschreibung der Semantik im denotationalen und operationalen Stil.
Da der RFD-Formalismus in dieser Arbeit nicht zur Anwendung kommt, soll hier nicht weiter darauf eingegangen werden.

PRA: Die Pragmatik der Spezifikationen wird mit dem PRA-Formalismus definiert. Hier werden alle Spezifikationen zu einem interpretierbaren Programm zusammengesetzt. Dazu werden folgende Konzepte angewendet:
Verfeinerung wird im Sinne des Substitutionstheorems für kontextfreie Grammatiken angewendet. Das bedeutet, daß nicht definierte Symbole in einer Spezifikation durch eine andere Spezifikation definiert sind, die diese Spezifikation verfeinert.
Interpretation bedeutet im Sinne einer Algebra, daß Symbole, z.B. Funktionssymbole in attributierten Grammatiken mit ihren Interpretationen verbunden werden. Unterschiedliche Interpretationen können durch qualifizierte Präfixe eingeteilt werden.
Kopplung heißt, die Spezifikationen lassen sich phasenartig zusammensetzen. Eine Phase kann Ergebnisse vorhergehender Phasen weiterverarbeiten und selbst Zwischenergebnisse für weitere Phasen zur Verfügung stellen.
Umbenennung von Sorten und Symbolen ermöglicht deren Wiederverwendung in unterschiedlichen Kontexten.
Von PRA-Formalismen wird in diesem Projekt mehrfach Gebrauch gemacht. Die Hauptdatei, mit der die Spezifikationen aufgerufen werden, um den Quelltext einzulesen und zu transformieren, ist eine PRA-Spezifikation. Die Fehlerausgabe und die Initialisierung zur Generierung von Eiffel-Quelltext werden ebenfalls durch Nutzung der PRA-Formalismen beschrieben. Dabei werden die Mittel der Verfeinerung, der Interpretation und der Kopplung eingesetzt, nicht aber der Umbenennung.

LDL selbst ist weitestgehend plattformunabhängig. Alle Quelldateien und die Bibliotheken liegen als Textdateien vor. Allerdings muß das Dateisystem lange Dateinamen unterstützen. LDL benötigt zum Laufen einen Prolog-Interpreter. Die hier verwendete Version 3.5.2 von LDL unterstützt SWI-Prolog ab Version 2.7.14. Diese Prologversion ist u.a. für Unix und Win32 verfügbar.

LDL ist frei verfügbar auf der LDL Home Page unter [10].

SWI-Prolog ist ebenfalls frei verfügbar unter [11].

3.2 SmallEiffel

SmallEiffel ist der GNU-Eiffel-Compiler. Er befindet sich zur Zeit noch in der Entwicklung. Ziel ist es, SmallEiffel zu einem vollständigen, schnellen, aber kleinen freien Eiffel-Compiler zu entwickeln. Hier wurde die Version -0.80 verwendet. Diese Version besitzt noch einige Einschränkungen, die aber in diesem Projekt nicht stören, weil der hier erzeugte Quelltext die von diesem Compiler noch nicht unterstützten Eigenschaften von Eiffel nicht ausnutzt.

Um mit dem SmallEiffel-Compiler ein startbares Programm zu erzeugen, compiliert man die Eiffelklasse, von der aus die dynamische Objektstruktur des Programmes aufgebaut wird. Als Parameter für den Compiliervorgang gibt man eine Creation-Methode an, die das erste Objekt initialisiert. Beim Starten des erzeugten Programms mit dem einfachen Namen „a“ bzw. „se“ wird dann automatisch ein Objekt der angegebenen Klasse instanziiert und diese Creation-Methode aufgerufen.

Für dieses Projekt heißt das, daß keine besonderen Anstrengungen getroffen werden müssen, um startbare Programme zu erzeugen. (Bei ISE-Eiffel ist z.B. eine ACE-Datei zum Initialisieren einer Eiffelklasse notwendig.)

SmallEiffel unterstützt viele Betriebssysteme. Es soll nach Angaben der Entwickler auf allen Plattformen laufen, auf denen es den ANSI C - POSIX Compiler gibt. Die Versionen für DOS, Win32 und UNIX wurden auf Kompatibilität zu den hier erzeugten Eiffel-Quelltexten getestet, wobei sich die DOS-Version aufgrund der kurzen Dateinamen für dieses Projekt als weniger geeignet erwies.

Die neuste Version von SmallEiffel ist frei verfügbar unter [12].

Kapitel 4

Spezifikation des Compilers mit LDL

4.1 Plan für die Übersetzung

Wie weiter oben schon kurz angesprochen, setzt sich der Übersetzungsprozeß eines Quelle-Quelle-Übersetzers aus den drei Phasen Parsen, Transformieren und Generieren zusammen. Welche Anforderungen an das Parsen und an das Generieren gestellt werden müssen, steht jedoch erst fest, wenn der Umfang der Transformationen bestimmt ist. Daher wird hier erst einmal ein grober Plan für die Übersetzung festgelegt.

4.1.1 Aufwandsbegrenzung

Das Ziel ist es, die Transformationen so einfach wie möglich zu gestalten. Da sich die Syntax der Sprache *PaL* an Eiffel orientiert, sollen möglichst viele Konstrukte direkt in Eiffel-Quelltext überführt werden. Wenn Manipulationen an Konstrukten notwendig sind, dann sollen diese so einfach wie möglich sein. Konstrukte, die so überführt werden, müssen während der Transformation und Generierung nicht verstanden, sondern nur korrekt in Eiffel-Quelltext übertragen werden. Da bei solchen Transformationen eine Erkennung von Fehlern in der Semantik schwer realisierbar ist, wird die Fehlererkennung in den meisten Fällen dem Eiffel-Compiler überlassen. Es wird sich also bei diesem Prototyp um einen „positiv denkenden“ Compiler handeln, d.h. korrekt implementierte *PaL*-Programme werden korrekt übersetzt, Fehler in der Implementation offenbaren sich nicht immer bei der Übersetzung von *PaL* zu Eiffel, sondern zum Teil erst bei der Übersetzung des generierten Eiffel-Quelltextes.

4.1.2 Übersetzung eines Patterns ohne Verfeinerung

Pattern, die nicht durch Verfeinerung anderer Pattern aufgebaut wurden, werden von nun an als **flache Pattern** bezeichnet. Wenn man von der Instanziierung von Pattern und Komponenten absieht, ist die Abbildung eines flachen Patterns auf Eiffel-Quelltext relativ einfach. Komponenten eines flachen Patterns können fast ohne Änderungen in Eiffelklassen überführt werden. Ein Pattern kann man auch in eine Klasse überführen, die Patternfeatures werden dabei zu Features der Eiffelklasse. Die Bindung zu den Komponenten wird dynamisch zur Laufzeit aufgebaut. Da auf die internen Features von außen nicht zugegriffen werden darf, wird zusätzlich eine abstrakte Oberklasse gebildet. Diese Oberklasse beinhaltet die externen Features in abstrakter Form und bildet die Schnittstelle des Patterns zu anderen Pattern.

Den entstehenden Eiffelklassen müssen eindeutige Namen zugewiesen werden. Klassenbezeichner in Eiffel setzen sich aus Buchstaben, Zahlen und Unterstrichen „_“ zusammen. Das erste Zeichen darf dabei nur ein Buchstabe sein. Es soll in *PaL* die Möglichkeit bestehen, auf vorhandene Eiffelklassen aus Bibliotheken zuzugreifen, d.h. die Bezeichner in *PaL* müssen sich ebenfalls aus Buchstaben, Zahlen und Unterstrichen zusammensetzen.

Da der Umfang der nötigen Transformationen gering gehalten werden soll, erhält die abstrakte Oberklasse, die das externe Verhalten eines Patterns repräsentiert, genau den Namen des Patterns. Das hat hauptsächlich den Vorteil, daß bei der Transformation eines Patterns nicht über den Patternrand hinaus geschaut werden muß. Wird auf ein anderes Pattern verwiesen, ist keine Änderung der Referenz notwendig. Die transformierte Eiffelklasse verweist automatisch auf die Klasse, die das externe Verhalten eines Patterns repräsentiert. Dieses Konzept hat zusätzlich den Vorteil, daß man auch auf Pattern referenzieren kann, die in einem anderen *PaL*-Quelltext stehen, und sogar auf echte Eiffelklassen, um z.B. Eiffel-Bibliotheken zu nutzen.

Da auf jeden Fall mehr Eiffelklassen entstehen, als es Pattern und Komponenten gibt, müssen vom Präcompiler Namen konstruiert werden. Hierbei besteht das Problem, daß die Namensräume für Pattern, Komponenten und Eiffelklassen identisch sind. Wird der Name eines Patterns unverändert als Name für eine Eiffelklasse übernommen, so kann man mit diesem Namen immer auch einen vom Compiler konstruierten Klassennamen nachbilden, was zu einem Namenskonflikt führt.

Das Nachbilden von den konstruierten Klassennamen ließe sich verhindern, indem man den Namensraum für Pattern und Komponenten einschränkt. Eine Möglichkeit wäre hier das Verbieten des Unterstriches bei der Namensbildung. Alle vom Compiler konstruierten Klassennamen könnten, wenn sie einen solchen Unterstrich enthalten, im *PaL*-Quelltext nicht nachgebildet werden. Damit verwehrt man sich aber wieder die Möglichkeit, auf vorhandene Eiffelklassen aus Bibliotheken, die einen Unterstrich im Namen haben, zuzugreifen. Der Unterstrich wird häufig bei der Benennung von Eiffelklassen verwendet, um einen zusammengesetzten Namen lesbarer zu machen.

Der Unterstrich ist zwar als letztes Zeichen im Namen von Eiffelklassen zulässig, jedoch absolut unüblich. Daher ist es ein sinnvoller Kompromiß, den Unterstrich „_“ als letztes Zeichen im Namen von Pattern und Komponenten zu verbieten. Wenn die vom Compiler konstruierten Namen als letztes Zeichen einen Unterstrich bekommen, so lassen sich diese nicht in der *PaL*-Syntax nachbilden.

Die externe Schnittstelle eines Patterns wird, wie weiter oben beschrieben, durch eine abstrakte Klasse repräsentiert, die den unveränderten Namen des Patterns hat.

Das Pattern mit seiner vollständigen Implementation wird durch eine Eiffelklasse implementiert, die von der abstrakten Klasse erbt. Der Name dieser Klasse wird durch Dopplung des Patternnamen und Anhängen eines Unterstriches gebildet.

Die Komponenten eines Pattern werden durch Eiffelklassen implementiert, deren Name sich aus Verkettung von Patternname, Komponentenname und einem Unterstrich bildet.

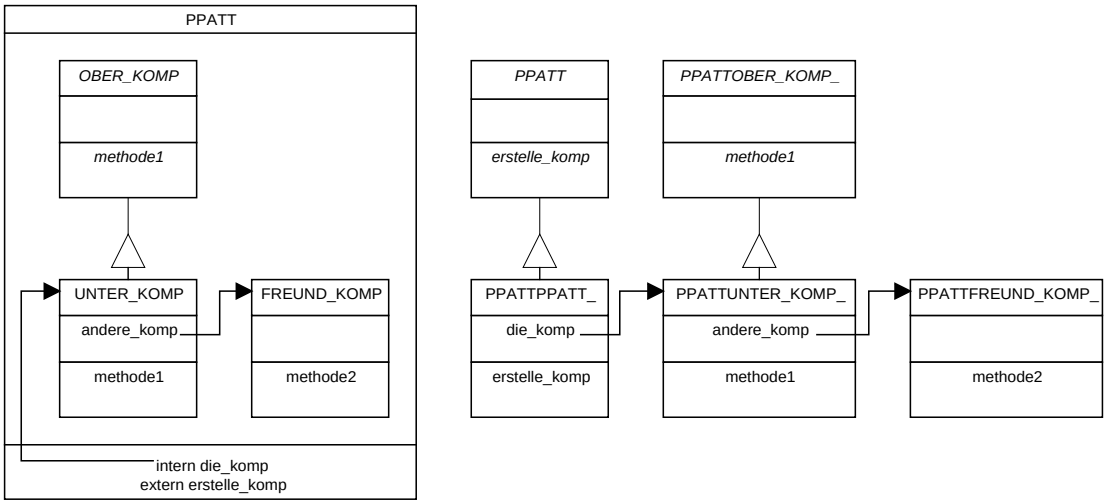

Abbildung 4.1: *Überführung eines Patterns in Eiffelklassen*

Die Abbildung 4.1 demonstriert an einem konstruierten Beispiel die Überführung des flachen Patterns `PPATT` in Eiffelklassen. Die externe Schnittstelle des Patterns wird durch die abstrakte Klasse `PPATT` repräsentiert. Diese Klasse ist die einzige Klasse dieses Patterns, die für die generierten Klassen anderer Patterns sichtbar ist. Sie enthält die externe Methode `erstelle_komp` in abstrakter Form. Tatsächlich werden Instanzen der konkreten Unterklasse `PPATTPPATT_` erzeugt. In dieser Klasse sind die externen und internen Patternfeatures implementiert. Die Komponenten des Patterns werden direkt in Eiffelklassen überführt. Vom Compiler werden aus den Namen des Patterns und der Komponenten die neuen Klassennamen `PPATTOBER_KOMP_`, `PPATTUNTER_KOMP_` und `PPATTFREUND_KOMP_` generiert.

Das ehemalige interne Patternattribut `die_komp` in der Klasse `PPATTPPATT_` verweist nun auf die Klasse `PPATTUNTER_KOMP_`.

4.1.3 Übersetzung eines Patterns mit Verfeinerung

Nachdem die Transformation eines Patterns, das in keiner Verfeinerungsbeziehung zu anderen Pattern steht, beschrieben wurde, soll diese Transformation so erweitert werden, daß sie auch auf Pattern mit Verfeinerungsbeziehungen anwendbar ist. Der Grundgedanke dazu ist, die Verfeinerungsbeschreibung eines Patterns so weit aufzulösen, daß es selbst zu einem flachen Pattern wird.

Es war gefordert, daß sich ein verfeinertes Pattern zu seinen Ausgangspattern bezüglich seiner externen Features polymorph verhält. Daher muß die aus diesem Pattern resultierende abstrakte Eiffelklasse, die das externe Verhalten des verfeinerten Patterns repräsentiert, von den abstrakten Klassen erben, die das externe Verhalten der Ausgangspatterns repräsentieren. Weitere Vererbungsbeziehungen sollen nicht zur Abbildung einer Verfeinerung in Eiffel-Quelltext verwendet werden. Die Eiffelklassen, die Komponenten des verfeinerten Patterns darstellen, stehen in keiner Beziehung zu den Klassen, welche die Komponenten der Ausgangspattern darstellen.

Ein Pattern wird also ebenfalls als flach bezeichnet, wenn alle Verfeinerungen bis auf eine Referenz auf die Ausgangspattern reduziert wurden.

Die Beschreibungsmittel der Verfeinerung sind Umbenennung von Patternfeatures, Komponenten und Komponentenfeatures und Auswahl von Implementationen von Features. Im ersten Schritt der Übersetzung wird nach dieser Verfeinerungsanleitung aus den Ausgangspattern ein völlig neues Pattern zusammengesetzt, so daß die Implementation des neuen Patterns nicht mehr auf mehrere Pattern verteilt ist, sondern nur in diesem Pattern steckt. Dazu wird von jedem verfeinerten Pattern eine temporäre Kopie angelegt. In dieser Kopie werden das Pattern, die Patternfeatures, die Komponenten und die Komponentenfeatures und alle Referenzen darauf entsprechend der Verfeinerung umbenannt. Die Patternfeatures und Komponentenfeatures werden entsprechend der Selektionen markiert. Dann werden die temporären Kopien mit dem neuen Pattern verschmolzen. Ein neu implementiertes Feature hat dabei in jedem Fall Vorrang vor einem Feature aus einem verfeinerten Pattern, sonst haben selektierte Features Vorrang. Treten Namenkonflikte bei Features zwischen den temporären Kopien auf, so wurde das Mittel der Selektion nicht korrekt angewendet. Es handelt sich dann um einen Fehler im *PaL*-Programm. Das Ergebnis dieses Prozesses ist ein flaches Pattern ohne Verfeinerungsklausel mit der gleichen Semantik, wie sie durch die Verfeinerung in der *PaL*-Implementation beschrieben wird.

Das flache Pattern wird, wie weiter oben beschrieben, in Eiffelklassen transformiert. Die abstrakten Eiffelklassen, die die externe Schnittstelle der Oberpatterns repräsentieren, werden entsprechend der Verfeinerungsbeziehungen in Vererbungsbeziehungen gesetzt.

Die Abbildung 4.2 stellt anhand eines konstruierten Beispiels die Übersetzung der Pattern `PREF`, `PERBE` und `PPATT` dar. `PPATT` wird von den flachen Pattern `PREF` und `PERBE` verfeinert. Bei der Verfeinerung von `PREF` werden die Komponenten `KOMP` und `FREUND` in `UNTER_KOMP` und `FREUNDKOMP` umbenannt. Das Attribut `der_freund` der Komponente `KOMP` wird in `andere_komp` umbenannt. Bei der Verfeinerung des Patterns `PERBE` wird die Komponente `VATER` in `OBER_KOMP` umbenannt und die Komponente `KIND` auf `UNTERKOMP` abgebildet. Das Feature `methode` aus `VATER` wird in `OBER_KOMP` auf `methode1` abgebildet. Das interne Patternattribut `das_kind` wird in `die_komp` umbenannt.

Die Übersetzung der flachen Pattern `PREF` und `PERBE` verläuft, wie oben beschrieben. Die Eiffelklassen `PREF` und `PREFPREF_` repräsentieren das Pattern `PREF` und die Klassen `PREFKOMP_` und `PREFFREUND_` stehen für die Komponenten des Patterns. Analog verhält es sich mit dem Pattern `PERBE`.

Wenn im Pattern PPATT die Verfeinerungen von PREF und PERBE aufgelöst werden, entsteht ein flaches Pattern, das im wesentlichen dem in der Abbildung 4.1 entspricht. Folglich wird es ebenso in Eiffelklassen übersetzt. Um den Polymorphismus auf Patternebene zu wahren, wird die Klasse PPATT, die das externe Verhalten des Patterns repräsentiert, von den Klassen PREF und PERBE abgeleitet. Die aus den Komponenten abgeleiteten Klassen PPATTOBER_KOMP_, PPATTUNTERKOMP_ und PPATTFREUND_KOMP_ stehen in keiner Beziehung zu den Klassen der Verfeinerten Pattern.

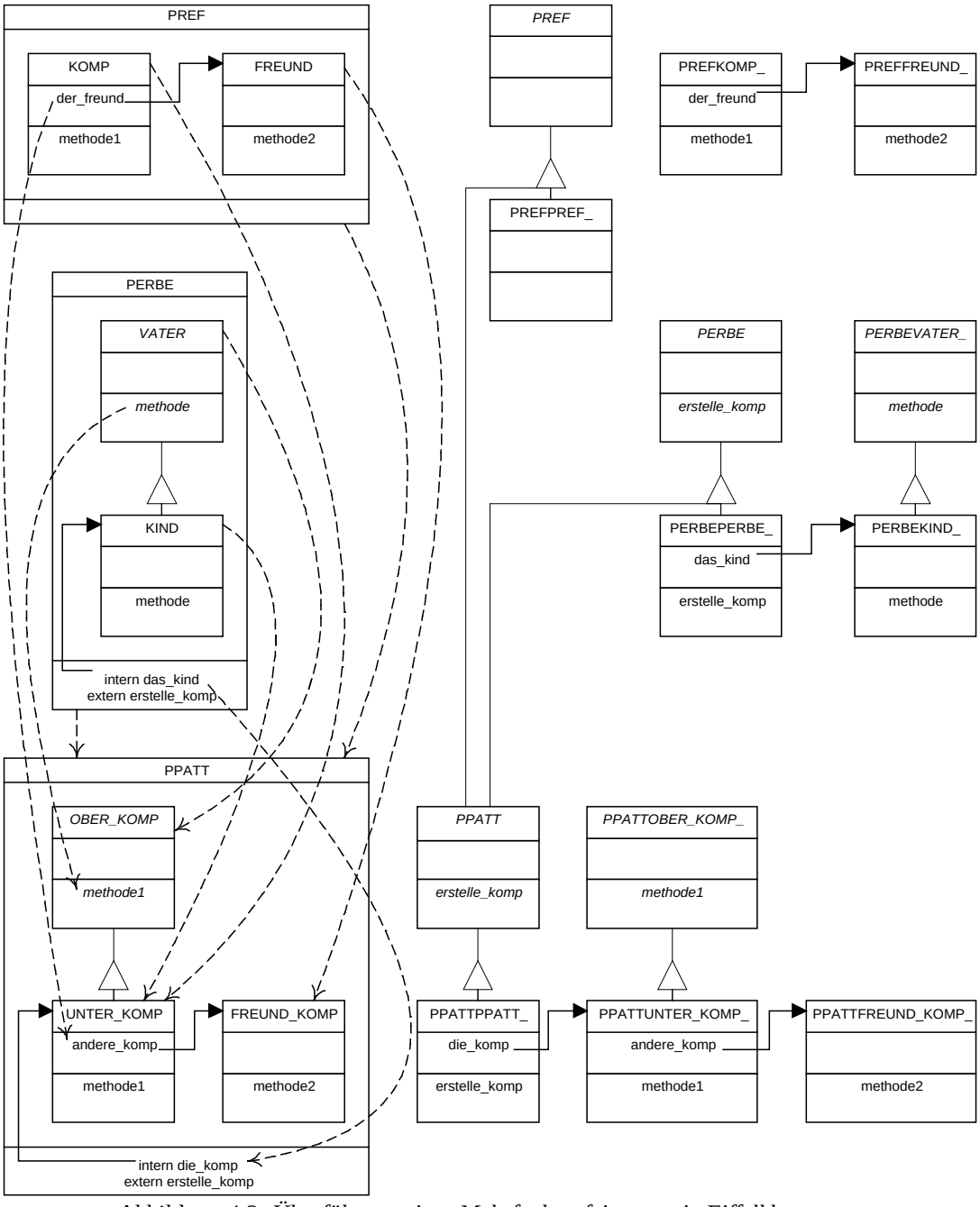

Abbildung 4.2: *Überführung einer Mehrfachverfeinerung in Eiffelklassen*

4.1.4 Aufbau des Übersetzers

Der Aufbau des Compilers läßt sich wie folgt darstellen:

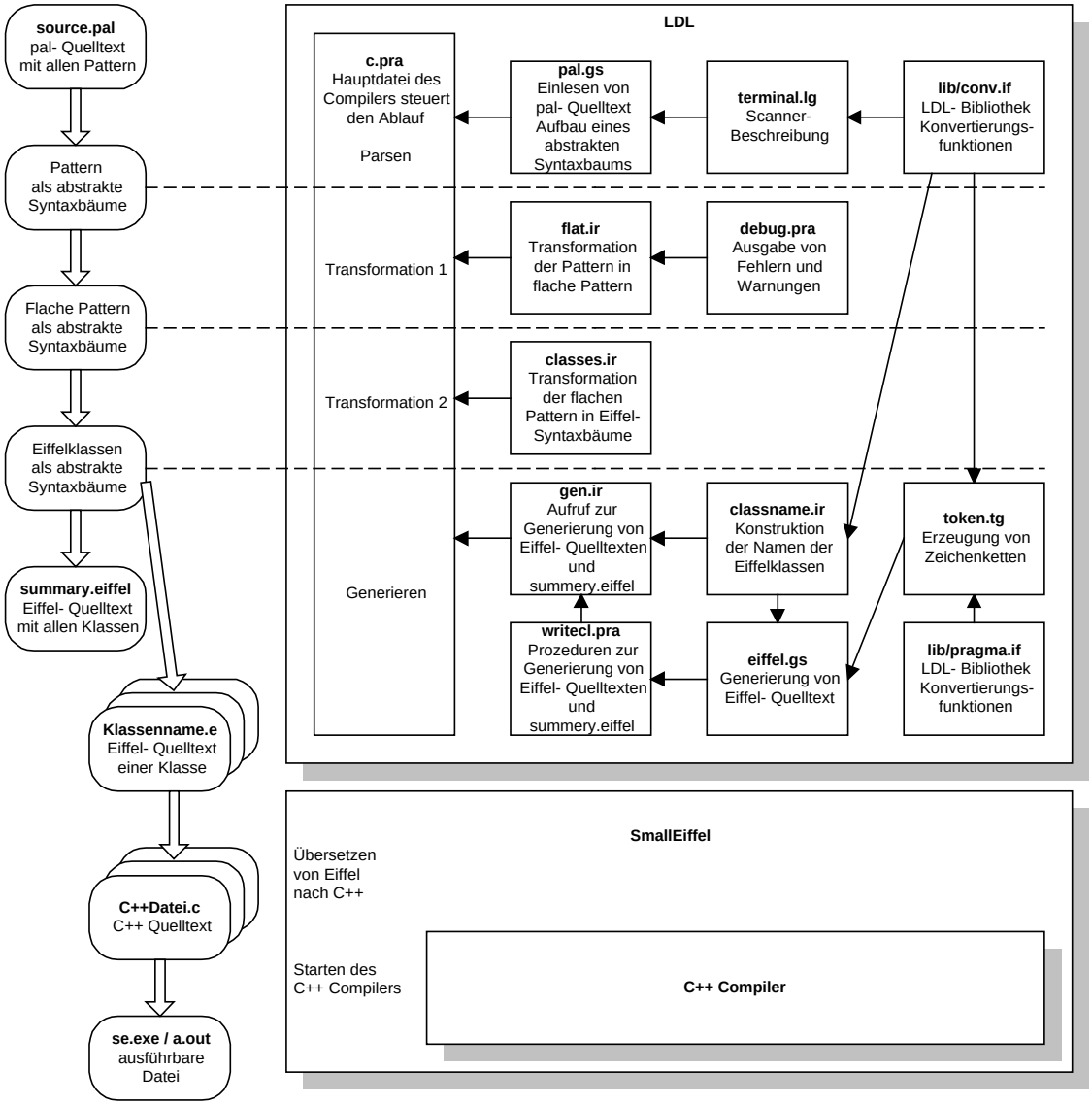

Abbildung 4.3: *Aufbau des PaL-Compilers*

Der Übersetzer muß die folgenden Schritte vollziehen:

LDL:

Parsen: Der *PaL*-Quelltext wird eingelesen und in einen Abstrakten Syntaxbaum umgewandelt.

Transformation 1: Jedes Pattern wird in ein flaches Pattern transformiert, d.h. alle Verfeinerungen werden aufgelöst.

Transformation 2: Die Syntaxbäume der flachen Pattern werden in Syntaxbäume von Eiffelklassen umgewandelt.

Generieren: Die Syntaxbäume der Eiffelklassen werden zu Eiffel-Quelltext umgewandelt. Dabei wird eine Datei pro Klasse mit dem Namen der Eiffelklasse generiert. Auf diese Art benötigt der Eiffel-Compiler die Eiffel-Quelltexte. Zusätzlich wird noch eine Datei generiert, die alle erzeugten Eiffelklassen beinhaltet. Auf diese Weise ist dem Programmierer ein leichter Vergleich mit der *PaL*-Datei möglich.

SmallEiffel:

Eiffel-Compilierung: Der SmallEiffel-Compiler übersetzt alle Eiffel-Klassen in C++ Quelltext.
C++ Compilierung: Anschließend wird von SmallEiffel automatisch ein C++ Compiler gestartet, der eine ausführbare Datei erzeugt.

4.2 Das Parsen

Das Ziel des Parsens ist es, den *PaL*-Quelltext einzulesen, Syntaxfehler zu erkennen und einen für die Transformation geeigneten Syntaxbaum aufzubauen. Wie bei der Beschreibung von LDL schon erwähnt, wird hierfür der GS-Formalismus verwendet.

Die Umwandlung der *PaL*-Syntax, wie sie in der EBNF-Notation spezifiziert wurde, in einen abstrakten Syntaxbaum verläuft im Allgemeinen nach folgendem einfachen Konzept:

Nichtterminale werden zu Knoten.
Schlüsselwörter werden weggelassen.
Morphemklassen-Terminale werden zu Blättern.

Von diesem üblichen Konzept wurde nur beim Parsen von linksrekursiven Ausdrücken, Feature-Calls und Typdeklarationen leicht abgewichen.

Die Identifier unterliegen den im Abschnitt 4.1.2 beschriebenen Namenskonventionen.

Es wird daher hier nur näher auf linksrekursive Ausdrücke, die Sonderbehandlung von Feature-Calls und Typdeklarationen und auf die Erkennung von Morphemklassen eingegangen.

4.2.1 Die Erkennung von Morphemklassen

Mit dem GS-Formalismus werden Schlüsselwörter automatisch als Terminale erkannt. Die Erkennung von Morphemklassen, wie Strings, Identifier und Zahlen muß durch den LG-Formalismus spezifiziert werden. Hierfür wurde das Modul *terminal.lg* erstellt. Wie im Namenskonzept beschrieben wurde, setzen sich Identifier aus Buchstaben, Zahlen und Unterstrichen zusammen. Das erste Zeichen darf nur ein Buchstabe und das letzte Zeichen nur ein Buchstabe oder eine Zahl sein. Es folgt der entsprechende Auszug aus der Spezifikation. Die vollständige Spezifikation *terminal.lg* ist, wie auch alle anderen Spezifikationen, im Anhang zu finden.

```
Sets
 letter         = 'A' .. 'Z' | 'a' .. 'z'.
 digit          = '0' .. '9'.
 but_unquote    = Any - '"'.

Classes
 id         = (letter | (letter (letter | digit | '_')* (letter | digit)))
              : lib/conv chars2identifier.
 string     = '"' ( but_unquote )* '"' : lib/conv charsQuoted2string.
```

4.2.2 Linksrekursive Ausdrücke

Das Top-Down-Prinzip des Parsens hat den Nachteil, daß keine linksrekursiven Ausdrücke erkannt werden können. Das heißt, Regeln der folgenden Art müssen umgestellt werden:

```
expression : expression, ...
```

Binäre Ausdrücke wurden in der Syntaxdefinition auf diese Art spezifiziert. Bei allen anderen Arten von Ausdrücken tritt dieses Problem nicht auf.

```
<EXPRESSION> ::=
       ( <CONSTANT>
       | '(' <EXPRESSION> ')'
       | <UNARY> <EXPRESSION>
       | <CALL>
       | <EXPRESSION> <BINARY> <EXPRESSION>
```

Daher wurden die Ausdrücke `expression` aufgeteilt in simple Ausdrücke `sexpression`, bei denen es keine Probleme mit Linksrekursivität gibt und rekursive Ausdrücke `rexpression`, die nicht linksrekursiv angewendet werden dürfen. Siehe *pal.gs* im Anhang, Regeln [r0350] bis [r0359].

4.2.3 Sonderbehandlung der Feature-Calls

Wie schon in der Syntaxdefinition angesprochen, sind die Möglichkeiten des Aufrufs eines Features etwas eingeschränkt. Folgende drei Varianten sind möglich:

Lokales Feature: Es wird ein Feature des aktuellen Patterns oder der aktuellen Komponente aufgerufen.
Lokale Variable ruft Feature: Ein Feature einer Pattern- oder Komponenteninstanz, auf die die lokale Variable verweist, wird aufgerufen.
Lokales Feature ruft Feature: Ein Feature einer Pattern- oder Komponenteninstanz, welches Ergebnis einer lokalen Funktion ist oder auf die ein lokales Attribut verweist, wird aufgerufen.

Beispiel

```
procedure_name(...);
local_variable.procedure_name(...);
local_attribute.procedure_name(...);
local_function.procedure_name(...)
```

aber nicht
`local_variable.function_name(...).procedure_name(...)`

Diese Arten des Feature-Aufrufs haben den Vorteil, daß man den Typ der Pattern- oder Komponenteninstanz, deren Feature aufgerufen wird, ohne vollständiges Typsystem ausschließlich durch den Quellcode des aktuellen Patterns feststellen kann. Der Typ wird als Attribut oder Funktion des aktuellen Patterns oder Komponente, als lokale Variable oder als Übergabeparameter der aktuellen Methode deklariert.

Der Syntaxbaum soll so aufgebaut werden, daß er für die folgenden Transformationen leicht zugänglich ist. Die einzigen Änderungen, die an der Implementation eines Features notwendig sind, um *PaL* in Eiffel umzuwandeln, sind Umbenennungen von Feature-Calls und bei der Instanziierung von Pattern und Komponenten. Würde der Syntaxbaum entsprechend der Syntaxdefinition aufgebaut werden, so wäre bei der Transformation bei jeder Umbenennung ein kompletter Abstieg in jeden Zweig des Syntaxbaums notwendig. Bei Erweiterungen des Befehlsumfanges der Sprache *PaL*, wären auch immer Ergänzungen der Spezifikation der Transformation notwendig, auch wenn die Feature-Umbenennungen nicht betroffen sind. Daher werden die Feature-Calls aus dem Ast des abstrakten Syntaxbaums, der für die

Implementation eines Features steht, herausgezogen und einer Liste, parallel zu diesem Ast des abstrakten Syntaxbaums, gespeichert. In dieser Liste werden die Namen der in der Implementation aufgerufenen Features festgehalten. Mit jedem Feature-Namen wird, wenn es ein lokales Feature ist, ein `current`, sonst der Name der lokalen Variablen oder des lokalen Features, von dem es aufgerufen wurde, abgespeichert. Zusätzlich wird schon ein Feld reserviert, in das bei der Transformation der Typ des Patterns oder der Komponente abgespeichert wird, dessen Feature aufgerufen wird.

Da der Parser kontextfrei arbeitet, kann er lokale Variablen, lokale Attribute und lokale Funktionen ohne Übergabeparameter nicht unterscheiden. Diese Ausdrücke werden daher alle wie ein Feature-Aufruf geparst. Siehe *pal.gs* im Anhang, Regeln [r0300] bis [r0359].

Beispiel

Das folgende Beispiel zeigt an einem *PaL*-Codefragment die Umwandlung in einen abstrakten Syntaxbaum.

```
pattern PCALL_BEISPIEL

  creation make

  feature make is do end

  feature ein_attribut: PANDERES_PATTERN

  feature eine_funktion(ein_parameter: PANDERES_PATTERN): PCALL_BEISPIEL is
    local
      lokale_variable: CALL_BEISPIEL
    do
      !!lokale_variable.make;
      ein_attribut := ein_attribut.umwandeln(ein_parameter);
      result := lokale_variable
    end

end
```

Der in Abbildung 4.4 gezeigte Auszug aus dem Syntaxbaum repräsentiert den *PaL*-Code des Features `eine_funktion` zwischen `do` und `end`. Ausgangspunkt für die Darstellung der Implementation im abstrakten Syntaxbaum ist der Knoten `instructions`. Er enthält zwei Listen, eine mit Befehlen und eine mit den Datensätzen der Feature-Calls. Die erste Liste enthält drei Knoten mit Befehlen, im `creation`-Knoten wird die Instanziierung festgehalten und in den beiden `let`-Knoten werden die Zuweisungen definiert. An den Blättern dieses Astes des Syntaxbaums steht jeweils ein `loccall`-Knoten. Dieser Knoten steht für einen Identifier im Quelltext, der eine lokale Variable oder ein Feature bezeichnet. Würde hier der Name des Identifiers festgehalten werden, dann wäre die spätere Umbenennung des Identifiers oder die Feststellung des Typs sehr aufwendig.

Aus diesem Grund werden diese Identifier, die alle potentielle Feature-Calls sind, in der zweiten Liste gespeichert. Diese Liste beinhaltet Datensätze, bestehend aus der aufrufenden Variablen, Platz für den Typ dieser Variablen und dem aufgerufenen Identifier.

Die gestrichelten Pfeile zeigen die Zuordnung der extra aufgelisteten Feature-Calls zum Ast des Syntaxbaums, der den rekursiven Aufbau der Funktionsimplementation festhält. Diese Pfeile dienen aber nur der Veranschaulichung, die Zuordnung wird nicht extra im Syntaxbaum festgehalten, sondern ergibt sich aus der Reihenfolge der Listeneinträge. Durch dieses Beispiel wird offensichtlich, daß eine Umbenennung der Features in einer Liste wesentlich leichter ist, als in einem Baum.

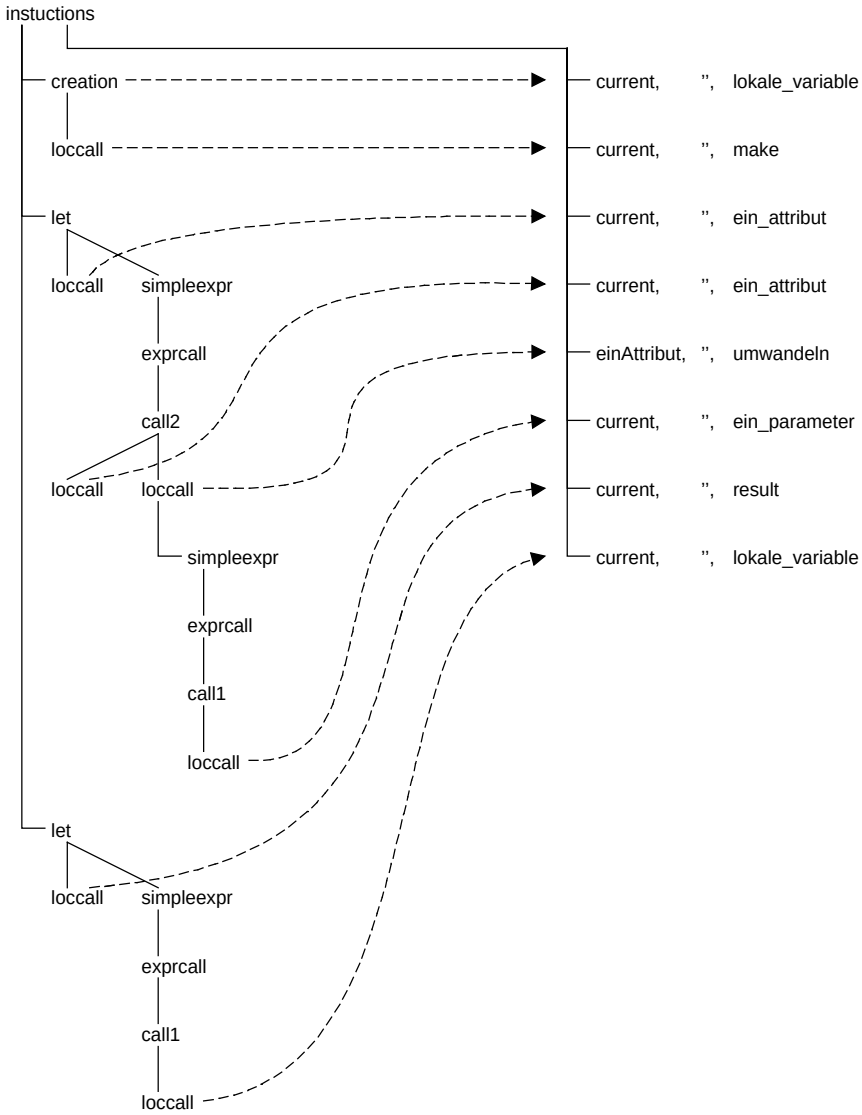

Abbildung 4.4: *Auszug eines abstrakten Syntaxbaums*

4.2.4 Typdeklarationen

Schon beim Parsen der Typdeklarationen wird das Konzept der zusammengesetzten Pattern- und Komponentennamen vorbereitet. Bei jeder Typdeklaration wird zusätzlich zum Typ ein Platzhalter angelegt, in den bei späteren Transformationen der Name des aktuellen Patterns eingetragen wird. Mit diesen beiden Angaben läßt sich dann beim Generieren des Eiffel-Quelltextes die korrekte Typbezeichnung konstruieren. Siehe *pal.gs* im Anhang, Regel [r0240].

```
[r0240] type -> type('', IDtype)
        :
          id -> IDtype.
```

4.3 Die erste Transformation

Das Ziel dieser Transformation ist die Vereinfachung des Syntaxbaums eines Patterns zu einem Syntaxbaum eines flachen Patterns. Bei dieser Transformation können eventuelle Fehler in der Semantik der Verfeinerungsbeschreibung gefunden werden. Die Transformation des Syntaxbaums wird mit dem IR-Formalismus in der Datei *flat.ir* spezifiziert und die Ausgabe der Fehler geschieht durch Prozeduren des PRA-Formalismus in der Datei *debug.pra*.

Ein Pattern wird in ein flaches Pattern umgewandelt, indem die Verfeinerungsschritte des Patterns vollzogen werden. Die Verfeinerung bietet das Mittel der Umbenennung von Komponenten, Patternfeatures und Komponentenfeatures. Bei Einsatz von Mehrfachverfeinerung wird zusätzlich das Mittel der Selektion relevant.

Die Abbildung 4.5 demonstriert grob die Vereinfachung eines Patterns, das in einer Verfeinerungsbeziehung zu zwei bereits abgeflachten Pattern steht. Bei dieser Vorgehensweise kann mit Pattern begonnen werden, die von keinem anderen Pattern verfeinert wurden. Die Umwandlung kann dann induktiv auf die gesamte Patternhierarchie fortgesetzt werden.

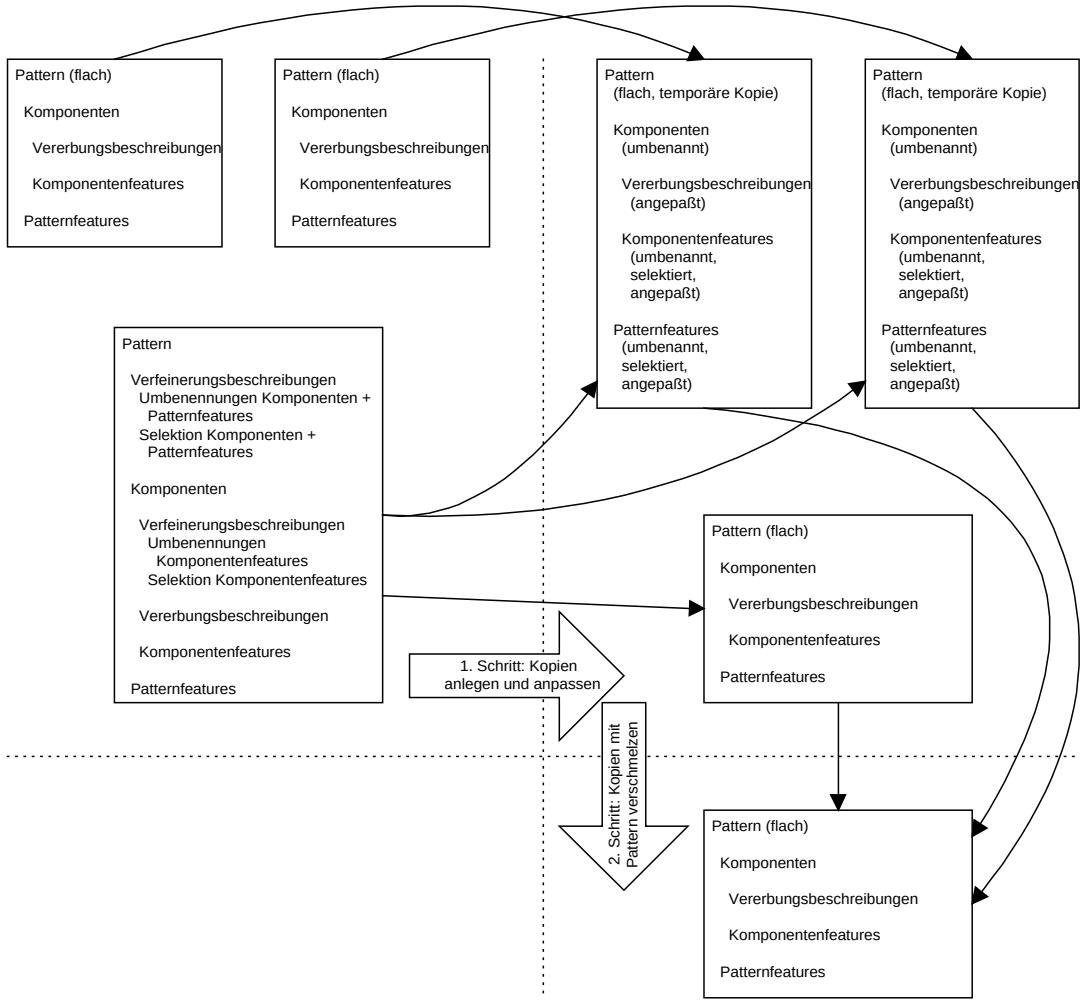

Abbildung 4.5: *Schritte der ersten Transformation*

Die Transformation geschieht in zwei großen Schritten:

1. **Umbenennung und Selektion:** Zuerst werden temporäre Kopien der bereits flachen Ausgangspattern angelegt. Dann werden die Umbenennungen an den Komponenten, Patternfeatures und Komponentenfeatures durchgeführt. Diese Umbenennungen machen zusätzliche Umbenennungen bei entsprechenden Typdeklarationen, Vererbungsbeschreibungen und Feature-Aufrufen erforderlich. Anschließend werden die Patternfeatures und Komponentenfeatures als selektiert oder unselektiert markiert. Aus dem zu vereinfachenden Pattern werden alle Verfeinerungsbeschreibungen entfernt. Es ist nun ein unvollständiges flaches Pattern, dem die Anteile der Oberpattern noch fehlen.
2. **Verschmelzung:** Die temporären Pattern werden mit dem unvollständigen flachen Pattern verschmolzen. Hierbei werden die Selektionsinformationen der Komponentenfeatures und Patternfeatures ausgewertet. Im Konfliktfall wird die Implementation des Features gewählt, das als selektiert markiert ist.

Die Auswahl der Reihenfolge der Pattern für die Transformation geschieht in der Regel [r0010] automatisch. Die Regel `pat2flat` läßt sich nur anwenden, wenn alle Oberpattern des Patterns `PAT` schon in der Menge der flachen Pattern `FLAT*` enthalten sind.

```
[r0010] pat2flat(PAT, FLAT*) -> FLAT,
        list(PAT'* ++ PAT''*, FLAT* ++ [FLAT]) -> FLAT'*
        ------------------------------------------------
        list(PAT'* ++ [PAT] ++ PAT''*, FLAT*) -> FLAT'*
```

Die oben beschriebenen Transformationsschritte werden durch die Relationen `pat2flat` in [r0020] und `refinesinflat` in den Regeln [r0030] und [r0031] gesteuert. Die Relation `pat2flat` extrahiert die Verfeinerungsbeschreibungen aus dem Pattern und stellt sie der folgenden Relation `refinesinflat` als Parameter zur Verfügung. Die Regel [r0030] realisiert die Umbenennung und Selektion von jeweils einer Verfeinerung und speichert diese in temporären Pattern. Die Anwendung der Regel [r0031] führt zur Verschmelzung dieser temporären Pattern.

Im folgenden sind diese Regeln um die Parameter verkürzt dargestellt und kurz erläutert.

```
[r0030]
renamecomp    : Anwendung der Umbenennungen auf die Namen der Komponenten
rencomptyp    : Anwendung der Umbenennungen auf die Vererbungsklauseln, Typdeklarationen
                und Implementationen der Komponentenfeatures
renamepfeat   : Anwendung der Umbenennungen auf die Namen der Patternfeatures
renamepfeattyp: und auf die Typdeklarationen und Implementationen der Patternfeatures
selectpfeat   : Markierung der Patternfeatures mit der Selektion
inhercasts    : Weiterleiten der Umbenennungen von Komponentenfeatures an Unterkomp.
castpfeat     : Umbenennungen von Komponentenfeatures in der Implementationen der
                Patternfeatures
castcomp      : Umbenennungen von Komponentenfeatures in der Implementationen der
                Komponentenfeatures
setselect     : Sammeln geerbter Komponentenfeatures und Selektion der
                Komponentenfeatures
refinesinflat : Rekursiver Aufruf für verbleibende Verfeinerungen
-------------
refinesinflat : Anwendung, wenn noch mindestens eine Verfeinerung existiert

[r0031]
comp2fcomp    : Anlegen einer Liste zur Verwaltung der Selektionen von
                Komponentenfeatures
merge         : Mischen der Patternfeatures, Komponenten und Komponentenfeatures
-------------
refinesinflat : Anwendung, wenn keine Verfeinerung existiert
```

In den folgenden Unterkapiteln wird auf die Realisierung der Mittel der Verfeinerung durch die Transformationsregeln etwas konkreter eingegangen.

4.3.1 Die Umbenennung von Komponenten

Soll in einer Verfeinerung eine Komponente umbenannt werden, so muß diese Umbenennung in der temporären Kopie des Oberpatterns der entsprechenden Verfeinerung durchgeführt werden. Die Relation `renamecomp` in [r1000] und [r1001] realisiert diese Umbenennung. Ergebnis dieser Relation ist die Menge der Komponenten, in der die entsprechenden Komponenten umbenannt wurden, und die Menge aller Umbenennungen, die noch nicht angewandt werden konnten. Die verbliebenen Umbenennungen werden später auf die Patternfeatures angewendet.

Die Umbenennung einer Komponente in der temporären Kopie des Oberpatterns hat zur Folge, daß auch jedes weitere Auftreten des Namens dieser Komponente korrigiert werden muß. Daher müssen in dieser temporären Kopie alle Vererbungen, die Parameter- und Rückgabetypdeklarationen aller Features und die Typdeklarationen lokaler Variablen durchsucht werden.

Anpassung der Vererbungsbeschreibungen

Alle Änderungen, die an Komponenten notwendig sind, werden bei der Anwendung der Relation `rencomptyp` in [r1010] und [r1011] durchgeführt. Dazu zählen die Anpassung der Vererbungen durch `renameinh` in [r1020] und [r1021] und die Anpassung der Komponentenfeatures durch `renamefeattyp` in [r1030] und [r1031]. Der Typ `currentcomponent` wird dabei durch den Namen der aktuellen Komponente ersetzt.

Anpassung der Typen in den Komponentenfeatures

Die Typanpassungen, die an den Komponentenfeatures notwendig sind, gleichen denen, die später auch noch an den Patternfeatures notwendig sind. Die Relation `renfeatbody` wurde daher so ausgelegt, daß sie mit beiden Featuretypen zurecht kommt. Für Komponentenfeatures sind die gleichen Modifikationen notwendig, wie bei internen Patternfeatures. Daher wird auch für Komponentenfeatures der Parameter `'intern'` übergeben. In den Regeln [r1040] bis [r1045] werden die nötigen Anpassungen für die einzelnen Featuretypen Attribut, Prozedur und Funktion spezifiziert. Die meisten Anpassungen sind z.B. bei internen Funktionen notwendig. Hier werden durch `rentypedecls` Umbenennungen an den Übergabeparametern, durch `renametype` am Rückgabetyp und durch `renameimpl` in der Implementation durchgeführt.

Anpassung der Typen in der Implementation der Features

Bei der Anpassung der Implementation in [r1060] sind Korrekturen an Typdeklarationen notwendig. Die Typen der lokalen Variablen werden wieder durch die Relation `rentypedecls` in [r1050] und [r1051] korrigiert. Es müssen aber auch die festgestellten Typen für die Feature-Aufrufe in der Liste `CRITFEAT*` durch die Relation `renamesource` in [r1070] und [r1071] umbenannt werden. Die Ermittlung der Typen für einen Feature-Aufruf wird weiter unten beschrieben. Ein Listeneintrag in der Liste `CRITFEAT*` hat die Form <Variablenname, Variablentyp, Featuretyp>, siehe auch Abbildung 4.4. Die Relation `rencrittype` in [r1080] und [r1081] realisiert die Umbenennung der Typnamen. Die Relation `rencritfeat` ist nur für die Umbenennung von Patternfeatures relevant.

Anpassung der Typen in den Patternfeatures

Nach dem alle Typen in den Komponenten angepaßt wurden, müssen auch alle Typenbezeichnungen in den Patternfeatures korrigiert werden. Das geschieht durch die Relation `renamepfeattyp` in [r1120] und [r1121]. Diese Relation benutzt die schon beschriebene Relation `renfeatbody` in den Regeln [r1040] bis [r1045].

Die Sprache *PaL* erlaubt keine Parameter in externen Patternfeatures die vom Typ einer Komponente sind. Daher führen alle auf solche Typen passenden Umbenennungen durch die Relationen `renametypeerror` und `rentypedeclserror` in den Regeln [1102] und [1103] bzw. [r1052] und [r1053] zu Fehlermeldungen.

4.3.2 Die Umbenennung der Patternfeatures

Die Patternfeatures werden durch die Relation `renamepfeat` in [r1110] und [r1111] umbenannt. Eingabe für diese Relation sind die Umbenennungen, die auf die Komponenten nicht angewendet werden konnten. Bleiben selbst nach der Anwendung auf die internen Patternfeatures noch Umbenennungen übrig, so kommt es in [r1112] zu einer Fehlerausgabe.

Die Umbenennung von internen Patternfeatures hat zur Folge, daß in dieser temporären Kopie auch jeder Verweis auf das Patternfeature korrigiert werden muß. Alle entsprechenden Feature-Aufrufe in den Implementationen der Pattern- und Komponentenfeatures müssen angepaßt werden. Da nur externe Patternfeatures Creation-Features sein sollen, ist keine Umbenennung in der Creation-Klausel notwendig.

Anpassung der Implementation der Features

Die Anpassung der Verweise auf die Patternfeatures wird zusammen mit der Anpassung der Verweise auf die Komponentenfeatures durchgeführt. Der Abstieg im Syntaxbaum zur Implementation der Komponentenfeatures beginnt ebenfalls mit der Relation `rencomptyp` und zur Implementation der Patternfeatures mit der Relation `renamepfeattyp`. Handelt es sich bei dem anzupassenden Komponenten- oder Patternfeature um eine Prozedur oder Funktion, so wird in beiden Fällen, wie schon bei der Umbenennung der Komponenten beschrieben, die Relation `renamesource` aufgerufen.

Die Relation `renamesource` in [r1070] und [r1071] benutzt die Relation `rencritfeat` in [r1100] und [r1101] um die Feature-Aufrufe von umbenannten internen Patternfeatures anzupassen. Die Patternfeatures werden an dem Typ `'currentpattern'` in der Liste `CRITFEAT*` erkannt.

4.3.3 Die Umbenennung von Komponentenfeatures

Die Umbenennung eines Features einer Komponente erfordert nicht nur die Anpassung der Implementationen der Pattern- und Komponentenfeatures und evtl. der Creation-Klausel, sondern auch die gleiche Umbenennung bei allen davon erbenden Komponenten. Die Umbenennung eines Komponentenfeatures muß also vom Programmierer immer beim ersten Auftreten in der allgemeinsten Komponente in der Komponentenhierarchie durchgeführt werden. Durch das Weiterreichen der Umbenennungen an erbende Komponenten kann sichergestellt werden, daß ein Feature auch dann einer Komponente zugeordnet werden kann, wenn es geerbt ist und nicht in der Implementation der Komponente auftaucht.

Umbenennung von Features in einer Komponente und allen erbenden Komponenten

Das Weiterleiten wird durch die Relationen `inhercasts` und `inhercast` in den Regeln [r1140] bis [r1144] realisiert. Dort werden die Verfeinerungsbeschreibungen aus den Komponenten des neuen Patterns herausgelöst und in die Komponentenhierarchie der temporären Patternkopie, die keine `CAST`-Einträge mehr enthält, einsortiert. Dazu werden die Komponenten entsprechend ihrer Hierarchie in der temporären Patternkopie bearbeitet. Eine Komponente erhält alle `CASTREN*` seiner Oberkomponenten und, wenn vorhanden die komplette `CAST`-Beschreibung aus der gleichnamigen Komponente des aktuell bearbeiteten Patterns. Informationen über Vererbung, Creations und Features sind von jetzt an für das `CAST` bedeutungslos und werden weggelassen.

Diese auf die `CAST`-Beschreibung reduzierten Komponenten dienen als Grundlage für die Umbenennung der Komponentenfeatures. Mit der Relation `castcomp` in [r1160] bis [r1162] werden durch Anwendung der Relation `renamecast` in [r1170] bis [r1173] sowohl die Komponentenfeatures umbenannt, als auch die Implementationen der Komponentenfeatures korrigiert. Eine Fehlerausgabe bei unpassenden Umbenennungen ist hier nicht möglich, da die geerbten Umbenennungen nicht immer anwendbar sind. Die Creations der Komponenten werden mit der Relation `rencreate` in [r1210] bis [r1212] korrigiert.

Anpassung der Implementation der Komponentenfeatures

Mit der Relation `castfeatbody` in [r1180] bis [r1182] werden die Featurearten unterschieden und bei Prozeduren und Funktionen wird die Relation `castsimpl` in [r1190] und [r1191] angewendet. Diese Relation ruft mit den Umbenennungen aus nur einer Komponente die Relation `castimpl` in [r1192] und [r1193] auf.

In der Relation `castimpl` wird für jeden Listeneintrag aus `CRITFEAT*` anhand des Typs der aktuellen Komponente überprüft, ob eine Umbenennung in Frage kommt. Wenn das der Fall ist, wird die Relation `castcritfeat` in [r1200] bis [r1202] angewendet. Dort findet eine Umbenennung statt, wenn der Name der Verfeinerung übereinstimmt oder nicht angegeben wurde und wenn eine Umbenennung auf den Featurenamen paßt.

Anpassung der Implementation der Patternfeatures

Wie die Implementationen der Komponentenfeatures, müssen auch die Implementationen der Patternfeatures angepaßt werden. Das geschieht durch die Relation `castpfeat` in [r1150] und [r1151]. Diese Relation nutzt die schon beschriebene Relation `castfeatbody`.

4.3.4 Die Selektion von Patternfeatures

Die Selektion der Patternfeatures wird durch die Relation `selectpfeat` in den Regeln [r1130] bis [r1133] realisiert. Eingabeparameter für diese Relation sind die evtl. vorhandene Selektion vor der `refine`-Klausel, die Namen der selektierten Patternfeatures und Komponenten in der `select`-Klausel und die Patternfeatures. Rückgabewerte dieser Relation sind die mit `'sel'` oder `'unsel'` als selektiert oder nicht selektiert markierten Patternfeatures und die verbliebenen Selektionen, die alle auf Komponenten anwendbar sein müßten.

Die Selektion der Patternfeatures durch die `select`-Klausel wird vorrangig vor der Selektion einer ganzen Verfeinerung ausgewertet. So wird sichergestellt, daß nach der Anwendung dieser Relation in der `select`-Klausel keine Namen von Patternfeatures mehr stehen.

4.3.5 Die Selektion von Komponentenfeatures

Die Selektion von Komponentenfeatures ist komplizierter zu realisieren als die der Patternfeatures. Zum einen ist sie auf drei Ebenen möglich, auf Verfeinerungsebene, auf Komponentenebene und auf Komponentenfeatureebene. Zum anderen soll es möglich sein, in einer Komponente geerbte, aber nicht neu implementierten Komponentenfeatures zu selektieren. Diese geerbten Komponentenfeatures sollen dann Vorrang gegenüber den in dieser Komponente implementierten Komponentenfeatures aus einer anderen Verfeinerung haben.

Die Relation `setselect` in [r1220] bis [1223] realisiert die Selektion der Komponentenfeatures. Eingabewerte sind die Selektion vor der `refine`-Klausel, die Namen der selektierten Komponenten, die bei der Selektion der Patternfeatures übriggeblieben sind, die auf die `CAST`-Beschreibung reduzierten Komponenten, die zu bearbeitenden Komponenten und die Bezeichnung der Verfeinerung. Das Ergebnis ist eine Menge von Komponenten, die um eine Liste mit den Namen aller eigenen oder geerbten Features, die als selektiert oder nicht selektiert markiert sind, erweitert wurden.

Test auf Selektion der Komponente

Durch Anwendung der Relation `testselect` in [r1230] bis [r1232] wird mit Hilfe der verbliebenen Selektionsliste und der Selektion der Verfeinerung festgestellt, ob die gesamte Komponente selektiert ist. Der Name der Komponente taucht nach der Selektion nicht mehr in der Selektionsliste auf. Sind nach Anwendung der Relation `setselect` auf alle Komponenten noch Namen in der Selektionsliste, so führt das zu einer Fehlerausgabe.

Ermittlung der vollständigen Featureliste, inkl. geerbte Komponentenfeatures

Durch die Relation `inhinfullc` in [r1240] und [r1241] wird für die Komponenten eine vollständige Liste mit all ihren Features und deren Selektionszustand angelegt. Dafür wird für die Komponenten, oben in der Komponentenhierarchie beginnend, die Relation `inherfullf` in [r1260] und [r1261] angewendet.

Durch diese Relation gesteuert, wird durch mehrfache Anwendung der Relation `inher1feat` in [r1262] und [r1263] jedes Feature aus den bereits angelegten Featurelisten der Oberkomponenten importiert. Die Selektionsmarkierungen werden dabei nicht übernommen.

Nachdem die Features aller Oberkomponenten in die Liste eingetragen wurden, werden durch die Relation `feat2fullf` in [r1250] bis [r1252] auch noch die Namen der direkt in der Komponente implementierten Features hinzugefügt.

Markierung der Selektion der Komponentenfeatures

Anschließend werden mit der Relation `setfeatsel` in [r1270] bis [r1272] in der nun vollständigen Liste der Komponentenfeatures die selektierten Features markiert. Diese Relation markiert durch Anwendung der Relation `set1featsel` in [r1280] bis [r1282] die Namen der Komponentenfeatures in der Featureliste mit `'sel'` oder `'unsel'` als selektiert oder nicht selektiert. Bleiben dabei Selektionen übrig, die auf diese Verfeinerung passen, so erscheint eine Fehlermeldung.

4.3.6 Das Zusammensetzen des Patterns

Nach dem alle temporären Patternkopien durch die Regel `refinesinflat` in [r0030] fertiggestellt wurden, werden sie in [r0031] mit der neuen Patternbeschreibung zusammengefügt. Dafür wird, um ein einheitliches Format zu schaffen, wie in den temporären Kopien in jeder neuen Komponente eine Liste mit den Namen der Komponentenfeatures zur Auswertung der Selektionen angelegt. Anschließend werden nacheinander die temporären Kopien hinzugemischt.

Vorbereitung der neuen Komponenten

Die Relation `comp2fcomp` in [r2000] und [r2001] legt in den neuen Komponenten eine leere Liste für Featurenamen mit Selektionsmarkierung an. Diese Liste ist in erster Linie dazu da, daß die Parameter der Relation `merge` gleiche Typen haben. Die Tatsache, daß ein Komponentenfeature in der Komponente implementiert ist, aber nicht in der Featureliste auftaucht, läßt sich später als Information dafür verwenden, daß das Komponentenfeature neu implementiert ist und nicht aus einer Verfeinerung stammt.

Mischen der Pattern

Mit der Relation `merge` in [r2010] werden durch mehrmalige Anwendung der Relation `mix` in [r2020] dem neuen Pattern nacheinander die temporären Kopien der Oberpattern hinzugemischt. Die Verfeinerungsbeschreibungen sind dabei nicht mehr relevant. Es werden aber die Namen der Oberpattern in einer Liste `INHER*` festgehalten, die es später ermöglichen soll, daß die Eiffelklassen, die Abbilder der Pattern werden, in eine Vererbungsbeziehung gesetzt werden können.

Durch die Relation `mixcomp` werden die Komponenten der beiden Pattern gemischt. Dieser Vorgang wird im nächsten Unterkapitel beschrieben.

Mischen der Patternfeatures

Die Relation `mixpfeat` in [r2160] bis [r2166] mischt die Patternfeatures beider Pattern. Dazu wird mit der Relation `extint` in [r2170] und [r2171] festgestellt, ob das Patternfeature extern oder intern sein wird. Ist mindestens eine Implementation des Patternfeatures als extern deklariert, so ist es extern, sonst ist es intern.

Das Mischen der Patternfeatures geschieht nach folgender Vorgehensweise.

Wird ein Patternfeature in diesem Pattern neu implementiert, hat es grundsätzlich Vorrang vor Patternfeatures die aus den temporären Kopien der Oberpattern stammen.

Wird ein Patternfeature nicht neu implementiert so werden zunächst auch unselektierte Patternfeatures hinzugemischt und mit `'unsel'` markiert.

Soll einem mit `'unsel'` markierten Feature ein weiteres unselektiertes Patternfeature hinzugemischt werden, so wird dieses mit `'unsel2'` markiert. Das bedeutet, wenn das Feature nicht noch durch ein selektiertes Patternfeature überschrieben wird, liegt ein Namenskonflikt vor.

Selektierte Patternfeatures überschreiben unselektierte Patternfeatures aus anderen Verfeinerungen. Sie werden mit `'sel'` markiert. Werden zwei selektierte Patternfeatures gemischt, dann wird ein Fehler ausgegeben.

Nachbearbeitung des gemischten Patterns

Im zweiten Teil der Relation `merge` in [r2011] wird das fertiggemischte flache Pattern noch nachbearbeitet. Es wird überprüft, ob es Namenkonflikte bei Pattern- und Komponentenfeatures gab. Aus den Komponenten werden die Selektionslisten wieder entfernt. An dieser Stelle werden auch den Feature-Aufrufen in den Implementationen der Pattern- und Komponentenfeatures Typen zugeordnet, was im Unterkapitel 4.3.8 näher beschrieben ist.

Die Überprüfung auf Namenskonflikte bei Patternfeatures erfolgt durch die Relation `testsel` in [r2110] und [r2111]. Dort wird kontrolliert, ob kein Patternfeature nach dem Mischen mit `'unsel2'` markiert ist, sonst erfolgt eine Fehlerausgabe.

Bei den Komponentenfeatures erfolgt die Überprüfung auf Namenskonflikte über die Relation `fcomp2comp` in [r2090] und [r2091] durch die Relation `testunsel` in [r2100] und [r2101].

4.3.7 Die Zusammensetzung der Komponenten

Die Relation `mixcomp` in [2030] bis [2031] beschreibt das Mischen der Komponenten aus den Verfeinerungen zu den neuimplementierten Komponenten. Dazu gehört das Mischen der Creations, das Mischen der Vererbungen und das Zusammenstellen der Komponentenfeatures. Die Vererbungen werden zu diesem Zweck einfach summiert.

Mischen der Creations

Die Creations werden durch die Relation `composeids` in [r2040] und [r2041] so zusammengestellt, daß jedes Feature maximal einmal aufgezählt wird.

Mischen der Komponentenfeatures

Das Mischen der Komponentenfeatures geschieht durch die Relation `mixfeat` in [r2050] bis [r2059] nach ähnlichen Regeln wie das Mischen der Patternfeatures. Neu ist hier nur, daß auch geerbte, nicht in dieser Komponente implementierte Features selektiert sein können. Dieser Fall wird durch die Regeln [r2056] bis [r2058] abgedeckt.

4.3.8 Zuordnung der Typen zu den Feature-Calls

Damit die Umbenennung der Pattern- und Komponentenfeatures in den Implementationen der Funktionen und Prozeduren der Pattern und Komponenten funktioniert, müssen die Typen der Pattern und Komponenten festgestellt werden, deren Features aufgerufen werden. Das geschieht in der Relation `merge` in [r2011]. Pattern- und Komponentenfeatures können nur von lokal deklarierten Variablen oder von Attributen oder den Rückgabewerten von Funktionen des aktuellen Patterns bzw. der aktuellen Komponente aufgerufen werden. Da die Namen der Variablen, von denen aus die Features aufgerufen werden, im Syntaxbaum gespeichert sind, müssen nur noch die Typdeklarationen, die auf diese Variablen zutreffen können und wo eine Umbenennung des aufgerufenen Features möglich ist, zusammengestellt und den Feature-Aufrufen mit den abgespeicherten Variablen zugeordnet werden.

Zusammenstellung der Typdeklarationen des Patterns

Für die Patternfeatures beginnt die Zusammenstellung der relevanten Typdeklarationen mit der Anwendung der Relation `pfeats2tdecl` in [r2150] und [r2151]. Dort werden durch die Regel

`feat21tdecl` in [r2145] die Attribute mit ihren Typen und in [r2146] die Funktionen mit ihren Rückgabetypen des aktuellen Patterns zusammengestellt.

Nach der Ermittlung der Typen der Patternfeatures wird mit diesen Typdeklarationen als Parameter die Relation `typepfeats` in [r2070] und [r2071] aufgerufen. Dabei wird die Deklaration der Variable `current` mit dem Typ `currentpattern` hinzugefügt. Diese Relation ruft für jedes Patternfeature die Relation `type1feat` in [r2074] bis [r2076] auf.

Die Relation `type1feat` ist ebenfalls auf Komponentenfeatures anwendbar. Bei der Anwendung auf Attribute geschieht nichts. Bei Prozeduren und Funktionen werden zu den bereits übergebenen Typdeklarationen die Deklarationen der Übergabeparameter und die der lokalen Variablen hinzugefügt und dann die Relation `vari2type` aufgerufen. Bei Funktionen wird zusätzlich die Ergebnisvariable `result` mit dem Ergebnistyp der Funktion deklariert.

Zuordnung der Typen zu den Variablen

Durch die Anwendung der Relation `vari2type` in [r2080] bis [r2082] werden den Variablen, von denen aus Features aufgerufen werden, die entsprechenden Typen zugeordnet. Ist für eine Variable der Typ nicht bekannt, dann muß es sich um ein anderes Pattern handeln, für dessen externe Features sprachbedingt so und so keine Umbenennung in Frage kommt. Für jeden Feature-Aufruf, für den eine Umbenennung möglich ist, steht nach Anwendung dieser Relation der Typ fest.

Zusammenstellung der Typdeklarationen der Komponenten

Für Komponentenfeatures verläuft die Ermittlung der Typen der Feature-Aufrufe nach dem gleichen Prinzip und größtenteils auch nach den gleichen Regeln. Startpunkt für die einzelnen Komponenten ist die Relation `fcomp2comp` in [r2090] und [r2091]. Ein Unterschied zu den Patternfeatures ist, daß in der Zusammenstellung der Typdeklarationen auch geerbte Features beachtet werden müssen.

Durch die Relation `inherftypes` in [r2120] und [r2121] werden die schon ermittelten Typdeklarationen der Oberkomponenten zusammengestellt. Durch dieses Verfahren sortieren sich die Komponenten automatisch entsprechend ihrer Hierarchie. Treten bei Mehrfachvererbung gleichnamige Features mit unterschiedlichen Typen auf, so wird durch die Relation `mixtypedecl` in [r2130] bis [r2132] eine Warnung vorbereitet, da hier einfach der zuerst gefundene Typ genommen wird.

Zu diesen geerbten Typdeklarationen werden durch die Relation `feats2tdecl` in [r2140] bis [r2143] die Deklarationen der Features in dieser Komponente hinzugemischt. Dafür wird wieder die schon beschriebene Relation `feat21tdecl` verwendet. Sollten dadurch nicht alle mit einer Warnung markierten geerbten Typdeklarationen überschrieben worden sein, so wird diese Warnung ausgegeben.

Dann wird, äquivalent zu den Patternfeatures, die Relation `typefeats` in [r2072] und [r2073] aufgerufen. Parameter dieser Relation sind die vorher ermittelten Typdeklarationen, denen noch der Bezeichner `current` mit dem Typ `currentcomponent` hinzugefügt wird. Hier wird die schon bei den Patternfeatures verwendete Relation `type1feat` in [r2074] bis [r2076] aufgerufen, die zu dem Ergebnis führt, daß auch alle Feature-Calls in der Implementation der Komponentenfeatures mit einem Typ verbunden sind.

4.4 Die zweite Transformation

Das Ziel dieser Transformation ist es, den abstrakten Syntaxbaum eines vereinfachten flachen Patterns in Eiffelklassen zu übersetzen. Wie weiter oben beschrieben, wird das Pattern selbst in eine abstrakte und eine konkrete Eiffelklasse transformiert und die zugehörigen Komponenten in jeweils eine Eiffelklasse. Bei dieser Transformation soll auch die Generierung der Namen der Eiffelklassen vorbereitet werden, da nach dieser Transformation der Bezug zu den Pattern verloren geht. Die zweite Transformation wird durch die Datei *classes.ir* beschrieben. Die Umsetzung der Ziele wird in den folgenden Unterkapiteln beschrieben.

4.4.1 Die Abbildung der externen Schnittstelle des Patterns

Durch die Relation `flat2classes` in [r1000] wird die Transformation eines Patterns in Eiffelklassen angestoßen. Zur Abbildung der externen Schnittstelle des Patterns auf eine abstrakte Eiffelklasse wird die Relation `flat2extclass` in [r1100] verwendet. Die Eiffelklasse erhält den gleichen Namen, wie das Pattern.

Vererbung zu Oberpattern

Dort wird die Relation `inh2clinh` in [r1110] und [r1111] aufgerufen, die dazu dient, für die Vererbungsklauseln die Realisierung des Namenskonzeptes vorzubereiten. Im Eiffel-Syntaxbaum stehen für jedes `inherit` zwei Bezeichner, aus denen der Name der Oberklasse zusammengesetzt wird. Da die Eiffelklassen, die die externen Schnittstellen repräsentieren, genauso heißen, wie die Pattern, müssen diese speziellen Vererbungsklauseln um einen leeren Identifier erweitert werden.

Umwandlung der Features in abstrakte Features

Die Relation `pfeat2deffeat` in [r1120] bis [r1122] bildet aus jedem externen Patternfeature ein abstraktes Eiffelfeature. Dazu wird bei externen Patternfeatures die Relation `feat2deffeat` in [r1130] bis [r1132] aufgerufen. Bei Attributen findet dabei keine Veränderung statt, bei Prozeduren und Funktionen werden die Implementationen durch ein `deferred` ersetzt.

4.4.2 Die Abbildung des vollständigen Patterns

Durch die Relation `flat2intclass` in [1200] wird das Pattern mit allen Patternfeatures in eine Eiffelklasse umgewandelt. Dazu werden die Komponenten weggelassen und die Patternfeatures leicht angepaßt. Es wird ein neues Vererbungskonstrukt angelegt, das beinhaltet, daß diese Klasse von der abstrakten Oberklasse erbt. Der Name der Eiffelklasse setzt sich aus zwei Bezeichnern zusammen, die beide mit dem Namen des Patterns belegt sind. Daraus wird beim Generieren von Eiffel-Quelltext der Klassenname in der Form „`PatternnamePatternname_`“ zusammengesetzt.

Die Relation `pfeat2feat` in [r1200] und [1210] macht aus Patternfeatures Eiffelfeatures. Dazu werden die Intern-Extern-Markierung und die Selektionsmarkierung weggelassen.

4.4.3 Die Abbildung der Komponenten

Komponenten werden durch die Relation `comps2classes` in [r1300] und [r1301] umgewandelt. Dazu wird für jede Komponente die Relation `comp2class` in [r1310] aufgerufen.

In dieser Relation wird der Name der Komponente um den Namen des Patterns ergänzt. Aus diesen beiden Bezeichnern wird beim Generieren von Eiffel-Quelltext der Name der Eiffelklasse in der Form „`PatternnameKomponentenname_`“ zusammengesetzt.

Zusätzlich werden durch die schon kurz beschriebene Relation `inh2clinh` in [r1110] und [r1111] die Vererbungsklauseln auf die Namenserweiterung vorbereitet. Vor jeden Namen einer Oberkomponente wird noch ein Bezeichner mit dem Namen des Patterns gesetzt. Beim Generieren wird dann wieder der vollständige Name der Oberklasse zusammengesetzt.

4.4.4 Die Anpassung der Typen an die Klassennamen

Da die meisten Klassen einen neuen Namen bekommen haben, müssen die Typen an diese neuen Namen angepaßt werden, so wie es bei den Vererbungen schon geschehen ist. Bei den Klassen, die die externe Schnittstelle des Patterns repräsentieren, ist keine Anpassung mehr nötig, da keine Implementationen vorhanden sind und die Parameter der Features nur vom Typ eines extern betrachteten Patterns sein können.

Die Relation `expandtypes` in [r2000] und [r2001] ruft für alle Klassen, die internes Verhalten des Patterns repräsentieren, für jedes Feature die Relation `expfeatures` in [r2010] und [r2011] auf, die wiederum die Relation `expfeatbody` in [r2020] bis [r2022] aufruft.

Die Relation `expfeatbody` unterscheidet nach Attributen, Prozeduren und Funktionen. Der komplexeste Fall sind die Funktionen. Hier müssen der Rückgabetyp, die Deklarationen der Übergabeparameter, die Deklarationen der lokalen Variablen und auch die Typen in den Listen der Feature-Calls in der Implementation erweitert werden.

Anpassung der Typdeklarationen

Für die Deklarationen der Übergabeparameter und der lokalen Variablen wird die Relation `exptypedecl` in [r2030] und [r2031] aufgerufen. Diese Relation benutzt für jeden einzelnen Typ die Relation `exp1type`, die auch für die Namenserweiterung des Rückgabetyps aufgerufen wird.

In der Relation `exp1type` in [r2040] bis [r2042] werden nur die Typen angepaßt, die zu diesem Pattern gehören. Typen von anderen Pattern müssen nicht erweitert werden, weil bei diesen Pattern nur auf die externe Schnittstelle zugegriffen werden soll und die dafür zuständige Klasse den gleichen Namen hat, wie das Pattern. Es werden also nur die Typen angepaßt, zu denen sich eine aus diesem Pattern abgeleitete Klasse finden läßt. Wird eine passende Klasse gefunden, so wird der Typ der Klassenbezeichnung angepaßt. An dieser Stelle wird auch erst der Typ `currentpattern` aufgelöst. Dazu wird nach der Klasse gesucht, die das vollständige Verhalten des Patterns repräsentiert. Sie wird an den identischen Bezeichnern erkannt, aus denen sich der Name der Klasse zusammensetzt. Diese Namen werden anstelle des Typs `currentpattern` eingetragen. Alle nicht erkannten Typen bleiben unverändert.

Anpassung der Creation-Typen

Die Relation `expcritfeats` in [r2050] und [r2051] ruft für jeden einzelnen Feature-Aufruf die Relation `exp1critfeat` in [r2060] bis [r2062] auf, um den zugehörigen Typ anzupassen. Dieser ist

noch für den Aufruf von `creation`-Features wichtig, damit zur Laufzeit die richtigen Klassen instanziiert werden. Dabei wird genauso vorgegangen, wie in der Relation `exp1type`.

4.4.5 Die Berechnung der Redefines

Die Sprache *PaL* nutzt im Gegensatz zu Eiffel in der Vererbungsklausel nicht das Konstrukt `redefine`. Eiffel benötigt dieses Konstrukt immer dann, wenn ein geerbtes Feature neu implementiert wird. In *PaL* können durch das Konzept der Mehrfachverfeinerung Fälle auftreten, in denen die Implementation der `redefine`-Klausel für den Programmierer nicht intuitiv ist. Daher wurde dieses Konstrukt nicht in die Sprache *PaL* aufgenommen. Um gültigen Eiffel-Quelltext zu erhalten, wird es an dieser Stelle durch die Relation `redefines` berechnet.

Ermittlung aller Features der Klasse

Durch die Relation `redefines` in [r3000] und [r3001] werden die Features jeder Klasse in der Liste `CLASSFEATS*` gesammelt. An dieser Stelle werden durch die Relation `mininher` in [r3050] und [3051] doppelt auftretende Vererbungsbeziehungen zur selben Oberklasse entfernt. Anhand der Liste `CLASSFEATS*` und mit Hilfe der vereinfachten Vererbungsklausel werden durch die Relation `testinher` in [r3010] und [3011] alle Features der Klasse ermittelt und die Vererbungsklauseln um das `redefine`-Konstrukt ergänzt.

Ermittlung der redefinierten Features

Dazu wird für jede Vererbungsbeziehung die Relation `comparefeats` in [r3030] bis [r3032] aufgerufen, welche die Features ermittelt, die von der jeweiligen Oberklasse geerbt werden und auch in dieser Klasse neu definiert werden.

Durch die Relation `composeids` in [r3040] und [r3041] werden den, in der Relation `idoffeats` in [r3020] und [r3021] ermittelten, klasseneigenen Featurenamen die Namen der Features der Oberklassen hinzugemischt, ohne daß diese doppelt in der Liste auftreten.

4.5 Das Generieren

In der letzten Phase der Übersetzung sollen Dateien mit Eiffel-Quelltexten generiert werden. Es soll für jede Klasse eine Datei mit dem Namen der Klasse erzeugt werden und eine Datei mit dem Quelltext aller Klassen. In dieser Phase werden aus den zweiteiligen Klassennamen vollständige Klassennamen zusammengesetzt. Die Umsetzung dieser unterschiedlich gearteten Ziele wird durch die Dateien *gen.ir*, *writecl.pra*, *eiffel.gs*, *classname.ir* und *token.tg* spezifiziert.

4.5.1 Die Steuerung der Quelltextgenerierung

Die Datei *gen.ir* übernimmt die Steuerung der Quelltextgenerierung.

Die Generierung der Dateien, welche die einzelnen Eiffelklassen enthalten, wird in der Relation `main` in [r0000] durch Aufruf der Relation `writeclasses` in [r0010] und [r0011] angestoßen. Ebenfalls von dieser Relation wird die Generierung der Zusammenfassung in der Datei *summery.eiffel* gestartet.

Die Relation `writeclasses` benutzt für die Erzeugung des Dateinamens aus den beiden Bezeichnern die Relation `names2file`. Diese Relation wird in der Datei *classname.ir* spezifiziert. Im Unterkapitel 4.5.2 wird darauf eingegangen.

Für die Generierung von Eiffel-Quelltext werden die Prozeduren `writeeclass` und `writesummary` der Spezifikation in der Datei *writecl.pra* verwendet. Dort wird das Schreiben der Dateien unter der Verwendung der Spezifikation *eiffel.gs* gestartet.

4.5.2 Die Erzeugung der Klassennamen

In der Datei *classname.ir* werden die Namen der Klassen konstruiert. Dazu werden die beiden Bezeichner, die bis hier die Klasse im abstrakten Syntaxbaum identifiziert haben miteinander verknüpft.

Der erste Bezeichner ist, wenn er belegt ist, der Name des Patterns. Der zweite Bezeichner ist der Name der Komponente oder des Patterns.

Ist der erste Bezeichner nicht belegt, so handelt es sich um die Klasse mit der externen Schnittstelle des Patterns. In diesem Fall ist der Name der Klasse gleich dem Namen des Patterns, der im zweiten Bezeichner festgehalten ist.

Steht im ersten Bezeichner der Name des Patterns, so werden die beiden Bezeichner verknüpft und ein Unterstrich „_“ hinten angehängt.

Die Verknüpfung wird in der Relation `concatnames` in [r0010] und [r0011] beschrieben.

Der Finder des SmallEiffel-Compilers, der zu den Klassennamen die entsprechenden Eiffeldateien sucht, beginnt die Suche nach solchen Dateien, die den Namen der Klasse in Kleinbuchstaben besitzen. Daher wurde zum Erzeugen von Dateinamen die Relation `names2file` in [r0000] angepaßt. Dort werden alle Buchstaben im konstruierten Bezeichner in Kleinbuchstaben umgewandelt.

Für die Erzeugung von Klassen wurde die Konstruktion des Klassennamens durch die Relation `concatcreation` in [r0020] und [r0021] so abgewandelt, daß in keinem Fall die abstrakten Klassen mit der externen Schnittstelle des Patterns erzeugt werden, sondern immer die konkret implementierten Unterklassen.

4.5.3 Das Schreiben des Eiffel-Quelltextes

Das Generieren des Eiffel-Quelltextes in eine Datei wird durch die Datei *eiffel.gs* spezifiziert.

Das Schreiben einer Klasse entspricht in etwa dem umgekehrten Prozeß zum Parsen einer Komponente. Nur die Klassennamen und Typen müssen noch mit der Relation `concatnames` in der Datei *classname.ir* zusammengesetzt werden.

Da sich die Struktur der Features nicht geändert hat, lassen sich zwischen den Regeln beim Parsen der Pattern- und Komponentenfeatures und beim Generieren der Eiffelfeatures direkte Entsprechungen finden. Beim Schreiben der Implementation der Features werden die aufgebauten Listen mit Feature-Aufrufen genauso abgebaut und in den Quelltext eingepaßt, wie sie aufgebaut wurden.

Der einzige Unterschied ist das Erzeugen einer Klasse. Durch die Relation `instr` in [r0322] wird explizit die Klasse angegeben, von der ein Objekt erzeugt werden soll. Dazu wird die Relation `concatcreation` in der Datei *classname.ir* aufgerufen. Auf diese Weise wird verhindert, daß die abstrakten Klassen mit der externen Schnittstelle des Patterns erzeugt werden. An deren Stelle werden Objekte der spezielleren Unterklassen instanziiert.

Kapitel 5

Einstieg in die Programmierung mit *PaL*

Programmierern mit guten Kenntnissen in objektorientierter Programmierung sollte der Einstieg in die Sprache *PaL* nicht schwerfallen. Man kann auf zwei Arten mit *PaL* „fast normal“ objektorientiert programmieren. Die eine Variante ist ein Programm aus nur einem Pattern, in dem man die Komponenten wie Klassen in der objektorientierten Programmierung verwendet, in der anderen Variante nutzt man nur Pattern ohne Komponenten wie man Klassen in der objektorientierten Programmierung nutzt. Wirklich sinnvoll wird der Einsatz der Sprache *PaL* aber erst, wenn man Strukturen wie Design Patterns wiederverwenden und kombinieren will.

Anhand eines Beispiels soll demonstriert werden, wie man Design Patterns implementiert und kombiniert. Es sollen die verwandten Design Patterns Kompositum und Besucher miteinander verknüpft und angewendet werden. Der vollständige *PaL*-Quelltext zu diesem Beispiel liegt in gedruckter Form im Anhang und in der Datei *source.pal* vor.

5.1 Das Design Pattern Kompositum

Zunächst soll das Design Pattern Kompositum in einer abstrakten Form so implementiert werden, daß es vielseitig wiederverwendbar ist.

Das Kompositum ist ein Strukturmuster. Es fügt Objekte zu Baumstrukturen zusammen, um sie als Teil-Ganzes-Hierarchien zu repräsentieren. Das Design Pattern Kompositum ermöglicht es Klienten, einzelne Objekte sowie Kompositionen von Objekten einheitlich zu behandeln.

Das Design Pattern Kompositum besteht aus den Komponenten `COMPONENT`, `COMPOSITE` und `LEAF`. Die Komponenten `COMPOSITE` und `LEAF` erben von `COMPONENT`. Die Komponente `COMPOSITE` ist im wesentlichen eine Liste, die Objekte vom Typ `COMPONENT` verwaltet.

Da man Listen oft gebrauchen kann, soll zunächst das primitive Pattern einer Liste implementiert werden. Es handelt sich hier nicht um ein Design Pattern aus [1].

Wenn man von dem umrahmenden Pattern absieht, gleicht die Implementation einer sehr primitiven, einfach verketteten Liste in *PaL* der Implementation in Eiffel. Neu ist nur die Patternmethode `make`, die eine `LIST`-Komponente instanziiert und das Patternattribut `the_list`, das auf diese Instanz verweist.

```
pattern PLIST

  creation make

  component LIST
    creation make
    feature make is ...
    feature add(an_item: ITEM) is ...
    feature delete is ...
    feature get: ITEM is ...
    feature rewind is ...
    feature next_item is ...
    feature prior(an_item : ITEM): ITEM is ...
    feature is_not_valid: Boolean is ...
    feature is_empty: Boolean is ...
    feature first: ITEM
    feature cursor: ITEM
  end -- component LIST
```

```
  component ITEM
    creation
     make
    feature make is ...
    feature next: ITEM
    feature set_next(an_item: ITEM) is ...
  end -- component ITEM

  extern feature make is
    do
      !!the_list.make
    end -- make

  intern feature the_list: LIST

end -- pattern PLIST
```

Das Design Pattern Kompositum wird vom List-Pattern verfeinert. Dadurch sind die Komponenten `COMPONENT` und `COMPOSITE` schon fast fertig. Es müssen zum vollständigen Pattern nur noch die Komponente `LEAF`, die Vererbungsbeziehungen und die Funktionalität des Kompositums, repräsentiert durch die Methode `operation`, hinzugefügt werden. Zur leichteren Wiederverwendung wird das Iterieren der Kindobjekte zum Weiterleiten von `operation` nicht in der Methode `operation` im Kompositum implementiert, sondern in der Methode `operation_items`, die von `operation` aufgerufen wird. Da die Methode `operation` meist Parameter benötigt, wird zusätzlich eine abstrakte Komponente `PARAM` angelegt, die bei einer konkreten Anwendung die erforderlichen Parameter beinhalten kann.

```
pattern PCOMPOSITE

  refine
    PLIST
      rename
        LIST as COMPOSITE,
        ITEM as COMPONENT,
        the_list as the_compos
    end

  creation make

  component COMPONENT
    feature operation(a_parameter: PARAM) is
      deferred
      end
  end -- component COMPONENT

  component LEAF
    inherit COMPONENT
    creation make
  end -- component LEAF

  component COMPOSITE
    inherit COMPONENT
    feature operation(a_parameter: PARAM) is
      do
        operation_items(a_parameter)
      end
    feature operation_items(a_parameter: PARAM) is
      do
        from
          rewind
        until
          is_not_valid
        loop
          cursor.operation(a_parameter);
          next_item
```

```
        end
      end -- operation_items
  end -- component COMPOSITE

  component PARAM
  end -- component PARAM

  extern feature make is
    do
      !!the_compos.make
    end -- make

end -- pattern PCOMPOSITE
```

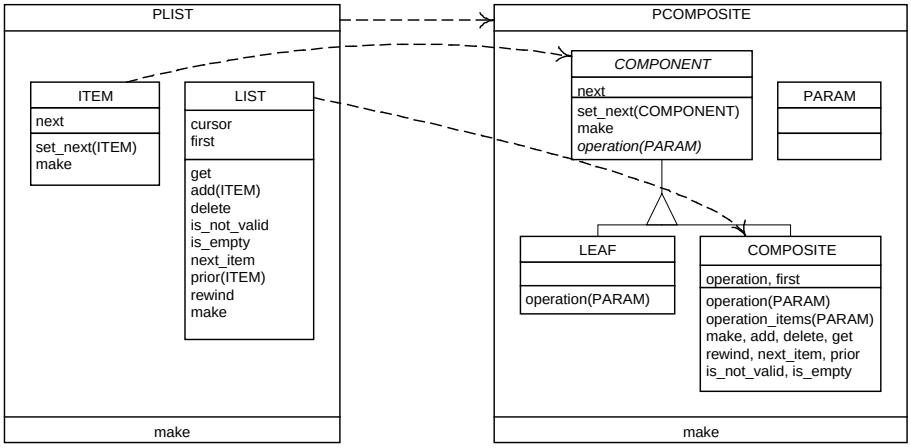

Abbildung 5.1: *Zusammensetzung des Patterns Kompositum*

Dieses kleine Beispiel demonstriert schon, wie leicht sich so häufig verwendete Strukturen, wie Listen unabhängig vom Programmkontext vorimplementieren und dann wiederverwenden lassen.

Die meisten Design Patterns haben mehrere Ansatzpunkte, an denen sie für den konkreten Einsatz erweitert werden können. Beim Pattern Kompositum kann man zum einen mehr Elemente hinzufügen, indem man die Komponente `LEAF` vervielfältigt, andererseits kann man auch mehr Funktionalität einbringen, indem die Methode `operation` vervielfältigt wird. Es ist empfehlenswert, bei der praktischen Anwendung eines Patterns für jede Art der Erweiterung einen eigenen Verfeinerungsschritt durchzuführen. (Siehe dazu auch Unterkapitel 2.1.2)

5.2 Das Design Pattern Besucher

In diesem Beispiel soll zusätzlich das Design Pattern Besucher Anwendung finden. Es wird daher zunächst abstrakt, wie in [1] implementiert.

Das Pattern Besucher ist ein Verhaltensmuster. Es kapselt eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Dieses Muster ermöglicht es, eine neue Operation zu definieren, ohne die Klassen der von dieser Operation bearbeiteten Elemente zu verändern.

Betrachtet man das Pattern Besucher abstrakt, so besteht es aus vier Arten von Komponenten. Jede Operation, die auf einer Menge von Komponenten ausführbar sein soll, wird in einer Komponente `CONCRETE_VISITOR`, die von `VISITOR` erbt, implementiert. Die Komponente `VISITOR` definiert eine einheitliche Schnittstelle. Die zu bearbeitenden Elemente `CONCRETE_ELEMENT` erben die Schnittstelle von der Komponente `ELEMENT`. (Siehe Abbildung 5.2 oben rechts)

Das Besuchermuster hat ebenfalls zwei Ansatzpunkte zur Erweiterung. Die Anzahl der Operationen läßt sich erweitern, in dem man Komponenten, wie `CONCRETE_VISITOR`, hinzufügt. Die Anzahl der zu bearbeitenden Elemente ist durch das Hinzufügen von Komponenten wie `CONCRETE_ELEMENT` erweiterbar.

Ein Besucher hat für jede Art von Elementen eine Methode `visit_concrete_element`, die von dem Element in der Implementation der eigenen Methode `accept` aufgerufen wird. So benötigt ein Element nicht für jede Operation eine neue Methode, sondern die Operation wird durch den entsprechenden Besucher als Parameter an die Methode `accept` übergeben.

Die Implementation des Design Patterns Besucher unterscheidet sich, abgesehen vom umrahmenden Pattern-Konstrukt, nicht von der objektorientierten Programmierung.

```
pattern PVISITOR

  component VISITOR
    feature visit_concrete_element(anelement: CONCRETE_ELEMENT) is
      deferred
      end -- visit_concrete_element
    feature make is
      deferred
      end -- make
  end -- component VISITOR

  component CONCRETE_VISITOR
    inherit VISITOR
    creation
      make
    feature make is
      do
      end -- make
  end -- component CONCRETE_VISITOR

  component ELEMENT
    feature accept(avisitor: VISITOR) is
      deferred
      end -- accept
  end -- component ELEMENT

  component CONCRETE_ELEMENT
    inherit ELEMENT
    feature accept(avisitor: VISITOR) is
      do
        avisitor.visit_concrete_element(current)
      end -- accept
  end -- component CONCRETE_ELEMENT

end -- pattern PVISITOR
```

5.3 Die Kombination von Kompositum und Besucher

In der kleinen Beispielanwendung ist es nun geplant, die Objektstruktur aus Instanzen von `COMPOSITE` und `LEAF` mit den beiden unterschiedlichen konkreten Besuchern `FRAME_VISITOR` und `CIRCLE_VISITOR` zu besuchen. Dafür muß das Pattern Kompositum nicht erweitert werden, das Pattern Besucher jedoch an zwei Stellen. Zum einen gibt es nun die zwei konkreten Elemente `LEAF` und `KOMPOSITE` und die zwei konkreten Besucher `FRAME_VISITOR` und `CIRCLE_VISITOR`. Eine solche Erweiterung in zwei Dimensionen läßt sich am übersichtlichsten in zwei Verfeinerungsschritten realisieren.

Im ersten Schritt wird die Komponente `ELEMENT` aus `PVISITOR` auf die Komponente `COMPONENT` aus `PCOMPOSITE` abgebildet. Die Komponente `CONCRETE_ELEMENT` aus `PVISITOR` wird dupliziert und auf die Komponenten `COMPOSITE` und `LEAF` aus `PCOMPOSITE` abgebildet. Das geschieht durch

Mehrfachverfeinerung vom Pattern PVISITOR. Die beiden Verfeinerungen erhalten die Namen <Ref_COMPOSITE> und <Ref_LEAF>. Da die Methode visit_concrete_element einen direkten Bezug zu der Komponente CONCRETE_ELEMENT hat, muß sie ebenfalls zu visit_composite und visit_leaf dupliziert werden. Das geschieht im cast-Konstrukt der Komponente VISITOR. Die Typen der Übergabeparameter werden immer automatisch angepaßt. Die Methode operation in den Komponenten des Patterns PCOMPOSITE wird auf die Methode accept aus dem Pattern PVISITOR abgebildet. Da die Methode accept mit Parametern vom Typ VISITOR arbeitet, wird die Komponente PARAM aus dem Pattern PCOMPOSITE in VISITOR umbenannt. Zwischen den beiden Verfeinerungen <Ref_COMPOSITE> und <Ref_LEAF> treten Namenskonflikte auf, weil der größte Teil der Features in beiden Verfeinerungen gleich heißt. Daher wird die Verfeinerung <Ref_COMPOSITE> selektiert.

In der Komponente COMPOSITE wird noch die Methode operation_items in accept_items umbenannt, die dann einen beliebigen Besucher an alle Listeneinträge weiterreichen kann.

```
pattern PCOMPOSITEVISITOR

  refine
    select PVISITOR <Ref_COMPOSITE>
      rename
        ELEMENT as COMPONENT,
        CONCRETE_ELEMENT as COMPOSITE
    end
    PVISITOR <Ref_LEAF>
      rename
        ELEMENT as COMPONENT,
        CONCRETE_ELEMENT as LEAF
    end
    PCOMPOSITE
      rename
        PARAM as VISITOR
    end

  component VISITOR
    cast
      rename
        visit_concrete_element from <Ref_COMPOSITE> as visit_composite,
        visit_concrete_element from <Ref_LEAF> as visit_leaf
    end
  end -- component VISITOR

  component COMPONENT
    cast
      rename
        operation as accept
    end
  end -- component COMPONENT


  component COMPOSITE
    cast
      rename
        operation_items as accept_items
    end
  end -- component COMPOSITE

end -- pattern PCOMPOSITEVISITOR
```

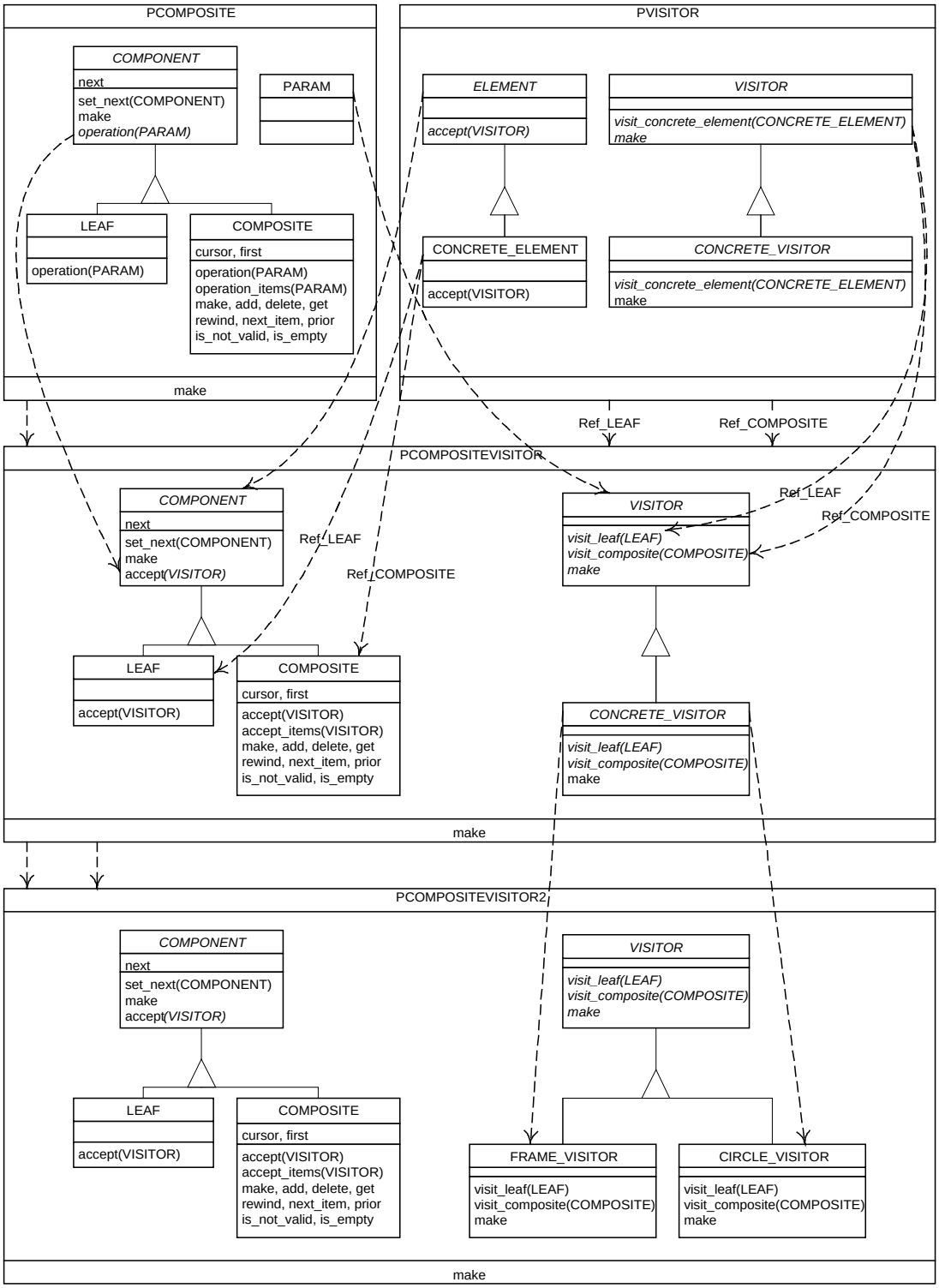

Abbildung 5.2: *Kombination der Pattern Kompositum und Besucher*

Im zweiten Schritt werden zwei konkrete Besucher mit echter Funktionalität implementiert. Dazu wird die Komponente `CONCRETE_VISITOR` zu `FRAME_VISITOR` und `CIRCLE_VISITOR` dupliziert. In diesem Beispiel werden die Methoden dieser Komponenten so implementiert, daß sie einfache Textausgaben erzeugen. Die Komponente `FRAME_VISITOR` erzeugt Ausgaben, in denen eckige Klammern vorkommen und die Komponente `CIRCLE_VISITOR` erzeugt Ausgaben, die runde Klammern enthalten.

In der Methode `make` des Patterns wird eine Baumstruktur erzeugt. Anschließend werden die Besucher `FRAME_VISITOR` und `CIRCLE_VISITOR` auf diese Struktur angewendet.

```
pattern PCOMPOSITEVISITOR2

  refine
    select PCOMPOSITEVISITOR
      rename
        CONCRETE_VISITOR as FRAME_VISITOR
    end
    PCOMPOSITEVISITOR
      rename
        CONCRETE_VISITOR as CIRCLE_VISITOR
    end

  creation make

  component FRAME_VISITOR
    feature visit_composite(acomposite: COMPOSITE) is
      do
        io.put_string("composite[ ");
        acomposite.accept_items(current);
        io.put_string("] ")
      end -- visit_composite
    feature visit_leaf(aleaf: LEAF) is
      do
        io.put_string("[leaf] ")
      end -- visit_leaf

  end -- component FRAME_VISITOR

  component CIRCLE_VISITOR
    feature visit_composite(acomposite: COMPOSITE) is ...
    feature visit_leaf(aleaf: LEAF) is ...
  end -- component CIRCLE_VISITOR

  extern feature make is ...

end -- pattern PCOMPOSITEVISITOR2
```

Die Datei *source.pal* wird mit Hilfe von LDL durch den folgenden Befehl übersetzt:

```
run(PATH/c).
```

Dabei bedeutet `PATH` den Pfad zur Datei *c.pra*. Bei erfolgreicher Übersetzung werden die Eiffeldateien und die Datei *summery.eiffel* erzeugt.

Anschließend muß mit dem SmallEiffel-Compiler die Eiffelklasse übersetzt werden, welche die Creation-Methode enthält, die zur Initialisierung ausgeführt werden soll. Der Compiler wird mit folgendem Befehl gestartet:

```
compile PatternnamePatternname_.e Methodenname
```

In diesem Beispiel heißt das:

```
compile pcompositevisitor2pcompositevisitor2_.e make
```

Dieser Aufruf läßt sich durch Skript-Dateien vereinfachen. In einer DOS-Umgebung z.B. durch eine Datei *c.bat* mit dem Inhalt:

```
compile %1%1_.e %2
```

Auf diese Weise muß der Nutzer nicht die Namenskonvertierungen des *PaL*-Compilers kennen. Er schreibt einfach:

```
c Patternname Methodenname
```

Wenn es wie in diesem Fall nur eine `make`-Methode gibt, ist die Angabe der Methode optional.

Bei erfolgreicher Übersetzung wird abhängig von Betriebssystem und C++ Compiler eine ausführbare Datei *se.exe*, *a.exe* oder *a.out* erzeugt. Das Programm erzeugt dann folgende Ausgabe:

```
composite[ [leaf] [leaf] composite[ [leaf] [leaf] [leaf] ] ]
composite( (leaf) (leaf) composite( (leaf) (leaf) (leaf) ) )
```

Werden beim Einlesen des PaL-Quelltextes vom LDL-Parser Syntaxfehler festgestellt, so wird die Übersetzung angehalten. LDL zeigt immer das zuletzt eingelesene Terminal an. Wenn das Parsen scheitert, ist dieses Terminal meist die Fehlerquelle.

Kapitel 6

Resümee und Ausblick

6.1 Resümee

Mit der Sprache *PaL* ist es gelungen, die eingangs gestellten Forderungen an ein patternorientiertes Programmiermodell umzusetzen. Mit Hilfe des entwickelten Prototyps eines *PaL*-Compilers lassen sich jetzt Erfahrungen mit dieser Sprache sammeln. In den folgenden Unterkapiteln sind erste Erfahrungen mit der Sprache und dem *PaL*-Compiler zusammengetragen.

6.1.1 Vor- und Nachteile der Sprache *PaL*

Die Sprache *PaL* erleichtert eindeutig den Umgang mit Design Patterns und anderen Klassenstrukturen. Sie ermöglicht erst deren Wiederverwendbarkeit, womit sich die Sprache *PaL* hauptsächlich von den objektorientierten Sprachen abhebt. Während in objektorientierten Programmiersprachen ein Design Pattern für jeden Einsatz neu implementiert werden muß, kann in der Sprache *PaL* ein vorimplementiertes abstraktes Design Pattern mit wenigen Zeilen Quellcode auf den jeweiligen Einsatz angepaßt werden. Die eingesetzten Pattern sind auch in der fertigen Implementation noch an ihren Namen erkennbar.

Diese offensichtlichen Vorteile erkauft man sich aber mit einem erheblichen Nachteil. Das schon aus objektorientierten Programmiersprachen bekannte Problem des verteilten Codes wird noch weiter verstärkt. Die Implementation eines Patterns ist auf viele Oberpattern verteilt. Es ist noch schwieriger als bei objektorientierten Programmiersprachen, den Überblick über die vollständige Implementation eines Patterns zu behalten. Für größere Projekte wäre ein Werkzeug für die grafische Unterstützung des Entwurfs und der Implementation erforderlich. Eine weitere denkbare Erleichterung wäre eine Information über den vollständigen Zustand eines Patterns.

Der Sprachentwurf hat sich bei der Implementation von Beispielen im allgemeinen als praktisch erwiesen. Die Einbindung des `cast`-Konstrukts in die Komponenten ist diskussionswürdig. Die Umbenennung von Komponentenfeatures hätte auch in die `refine`-Klausel eingebunden werden können. Vorteil wäre ein Wegfall der Referenzen auf die Verfeinerung, Nachteil wäre in vielen Fällen ein erhöhter Schreibaufwand.

Die erzeugten Eiffelklassen sind relativ gut lesbar und lassen sich mit von Hand implementierten Eiffelklassen kombinieren. Dadurch besteht die Möglichkeit, nur die eingesetzten Design Patterns in der Sprache *PaL* zu definieren. Diese können nach Eiffel übersetzt werden und in ein in Eiffel implementiertes Programm eingebunden werden.

6.1.2 Schwächen des Compilers

Einleitend zu diesem Abschnitt muß gesagt werden, daß bei der Übersetzung von korrekt implementierten *PaL*-Programmen keine Fehler bekannt sind. Der Compiler wurde anhand zahlreicher Design Patterns und zum Testen konstruierter Pattern in komplexen Verfeinerungsbeziehungen überprüft.

Die Schwächen des Compilers offenbaren sich erst, wenn es darum geht, Fehler in *PaL*-Programmen zu finden. Der *PaL*-Compiler hat kein eigenes Typsystem. Das hat zur Folge, daß die meisten Fehler nicht durch den *PaL*-Compiler erkannt werden und fehlerhafte Eiffeldateien erzeugt werden. Die Fehler werden dann erst durch den SmallEiffel-Compiler entdeckt. Der Programmierer muß anhand der Fehler in den Eiffeldateien die Fehler im *PaL*-Quelltext finden. Das ist zwar durch die gute Lesbarkeit der Eiffeldateien mit etwas Übung möglich, aber der Programmierer muß sich mit der Übersetzung von *PaL* zu Eiffel auskennen.

Bei den Fehlern, die der *PaL*-Compiler erkennt, erscheinen zwar Fehlerausgaben, aber die sind nur grob formuliert und verweisen nicht auf die entsprechende Stelle im Quelltext.

Der *PaL*-Compiler ist zur Zeit nur so definiert, daß er den gesamten Quelltext aus der Datei *source.pal* einliest. Wünschenswert wäre, wenn jedes Pattern aus einer eigenen Datei gelesen werden würde und auch bereits vorcompilierte Pattern nicht noch einmal übersetzt werden müßten. Zur Zeit ist bei der Wiederverwendung eines Patterns noch ein Copy-Paste in den aktuellen Quelltext notwendig.

6.2 Ausblick

Dieses Projekt bietet vielfältige Möglichkeiten zur Fortführung.

In erster Linie müssen die aufgezeigten Schwächen des *PaL*-Compilers ausgebessert werden. Die Entwicklung des Compilers könnte dahin gehen, daß ein vollständiges Typsystem implementiert wird und der Compiler alle Fehler in einem *PaL*-Quelltext erkennen und in der Fehlerausgabe die genaue Stelle angeben kann.

Ein weiterer Ansatz zur Verbesserung des Compilers wäre die Möglichkeit, Pattern separat übersetzen zu können. Der Compiler könnte neben den Eiffelklassen zu einem Pattern auch wieder *PaL*-Quelltext von der abgeflachten Variante des Patterns erzeugen. Dieser Ansatz hätte zwei entscheidende Vorteile. Zum einen muß der Compiler das Pattern nicht bei jeder Anwendung neu flach machen, was den größten Teil der Übersetzung einnimmt. Zum anderen kann sich der Programmierer durch diese Dateien einen genauen Überblick über die Struktur eines Pattern verschaffen und muß sie sich nicht selbst aus der Verfeinerungshierarchie zusammensuchen. Auf diese Weise wäre der größte Nachteil der Sprache *PaL* aufgehoben.

Wie schon weiter oben angesprochen, wäre auch ein Werkzeug zur grafischen Unterstützung denkbar. Mit diesem Werkzeug könnte die Struktur eines Patterns anschaulich definiert oder mehrere Pattern zusammengesetzt werden. Die Schritte zur Konstruktion eines Patterns könnten in automatisch generiertem *PaL*-Quelltext festgehalten werden.

Es könnten Bibliotheken für die Sprache *PaL* angelegt werden. Diese könnten abstrakte Implementationen aller bekannten Design Patterns enthalten.

6.3 Verwandte Arbeiten

Auf einem Workshop zu dem Thema „Language Support for Design Patterns and Frameworks“ (siehe [6] und [13]) in Verbindung mit der ECOOP’97 wurden einige Artikel vorgestellt, die sich ebenfalls mit der Sprachunterstützung von Design Patterns beschäftigen. In den Artikeln von Bosch, Hedin und Jacobsen werden ebenfalls Probleme objektorientierter Programmiersprachen im Umgang mit Design Patterns herausgearbeitet. Die dort vorgestellten Konzepte zur Lösung dieser Probleme durch neue Sprachen oder Spracherweiterungen unterscheiden sich aber zum Teil stark von dieser Arbeit.

6.3.1 Jan Bosch: Design Patterns as Language Constructs

Einführend werden in diesem Artikel [7] im wesentlichen die gleichen Probleme objektorientierter Sprachen identifiziert, wie in dieser Arbeit. Als Lösung dieser Probleme wird das Layered Object Model (LayOM) vorgestellt. Design Patterns werden in diesem Modell durch sogenannte Layer implementiert. Programme oder einzelne Layer-Klassen, die so implementiert wurden, lassen sich in C++ Klassen übersetzen. Das Problem bei diesem Modell ist, daß die Design Patterns bei der Übertragung in das LayOM umstrukturiert werden müssen. In dem Artikel werden einige Design Patterns kurz vorgestellt. Es wird auch angedeutet, daß sich mit diesem Modell Pattern kombinieren lassen.

6.3.2 Görel Hedin: Language Support for D. P. using Attribute Extensions

Dieser Artikel [8] konzentriert sich auf die Identifizierung und Validierung von Design Patterns im Quelltext. Dazu wird eine Technik verwendet, die sich attributierter Grammatiken bedient. Durch Nutzung von Kommentaren in einer speziellen Notation wird der Quelltext mit formalen Informationen über Pattern und Teilnehmerklassen attributiert. Dieses Konzept läßt zwar keine Wiederverwendung von Design Patterns zu, aber sie bleiben im Quelltext erkennbar. In gewissen Grenzen ist die korrekte Implementation der Design Patterns anhand vorgegebener Regeln automatisch überprüfbar.

6.3.3 Eyoun Eli Jacobsen: Design Patterns as Program Extracts

In diesem Artikel [9] wird ein Ansatz vorgestellt, in dem die Implementation in zwei Level unterteilt wird, in ein Program Level und ein Extract Level. Das Program Level stellt die Struktur des Programms für den konkreten Einsatz dar. Das Extract Level beschreibt, welche abstrakten Design Patterns dafür zum Einsatz gekommen sind und wie diese kombiniert wurden. In dem kurzen Artikel wird nicht deutlich, ob neben den Klassenstrukturen der Design Patterns auch die Implementationen der Methoden wiederverwendbar sind.

Anhang

A.1 Die *PaL*-Syntax

```
<STRING> ::=
Ergebnis der lexikalischen Analyse

<NAT> ::=
Ergebnis der lexikalischen Analyse

<IDENTIFIER> ::=
Ergebnis der lexikalischen Analyse

<EIFFELFEATURENAME> ::=
<IDENTIFIER>

<PATTERNNAME> ::=
<IDENTIFIER>

<PATTERNREF> ::=
‘<’ <IDENTIFIER> ‘>’

<PATTERNFEATURENAME> ::=
<IDENTIFIER>

<COMPONENTNAME> ::=
<IDENTIFIER>

<COMPONENTFEATURENAME> ::=
<IDENTIFIER>

<COMPONENT_OR_FEATURENAME> ::=
( <COMPONENTNAME> | <PATTERNFEATURENAME> )

<PATTERNORCOMPONENTFEATURE> ::=
        ( <PATTERNFEATURENAME> | <COMPONENTFEATURENAME> )


<PATTERNDECL> ::=
pattern <PATTERNNAME
                [ refine { <REFINE> }+ ]
                [ creation <PATTERNFEATURENAME> { , <PATTERNFEATURENAME> }* ]
                { component <COMPONENT>}*
                { [ intern | extern ] feature <PATTERNFEATURE>}*
        end

<REFINE> ::=
[ select ] <PATTERNNAME> <PATTERNREF> [
                [ rename <RENAME> { , <RENAME> ) }* ]
                [ select <COMPONENT_OR_FEATURENAME> { , <COMPONENT_OR_FEATURENAME> ] }* ]
        end ]

<RENAME> ::=
        <COMPONENT_OR_FEATURENAME> as <COMPONENT_OR_FEATURENAME>

<COMPONENT> ::=
        <COMPONENTNAME>
                [ cast <CAST> ]
                [ inherit { <INHERIT> }+ ]
                [ creation <COMPONENTFEATURENAME> { , <COMPONENTFEATURENAME> }* ]
                { feature <COMPONENTFEATURE> }*
        end

<CAST> ::=
        [ rename <RENAMECOMPONENTFEATURE> {, <RENAMECOMPONENTFEATURE> }*]
        [ select <CASTSELECT> {, <CASTSELECT> }*]
        end

<RENAMECOMPONENTFEATURE> ::=
        <COMPONENTFEATURENAME> [ from <PATTERNREF> ] as <COMPONENTFEATURENAME>
```

```
<CASTSELECT> ::=
        <COMPONENTFEATURENAME> from <PATTERNREF>

<INHERIT> ::=
        <COMPONENTNAME> { ; <COMPONENTNAME> }

<COMPONENTFEATURE> ::=
        <COMPONENTFEATURENAME>
        ( <ATTRIBUTEDECL> | <PROCEDUREDECL> | <FUNCTIONDECL> )

<PATTERNFEATURE> ::=
        <PATTERNFEATURENAME>
        ( <ATTRIBUTEDECL> | <PROCEDUREDECL> | <FUNCTIONDECL> )

<ATTRIBUTEDECL> ::=
        : <TYPE>

<PROCEDUREDECL> ::=
        [ '(' <TYPEDECL> { ; <TYPEDECL> }* ')' ]
        is <ROUTUNE>

<FUNCTIONDECL> ::=
        [ '(' <TYPEDECL> { ; <TYPEDECL> }* ')' ] : <TYPE>
        is <ROUTUNE>

<TYPE> ::=
        ( currentpattern | {<IDENTIFIER> )

<TYPEDECL> ::=
        <IDENTIFIER> : <TYPE>

<ROUTUNE> ::=
        [ local { <TYPEDECL> { ; <TYPEDECL> }* } ]
        ( deferred | do <COMPOUND> )
        end

<COMPOUND> ::=
        <INSTRUCTION> { ; <INSTRUCTION> }

<INSTRUCTION> ::=
        ( <CREATION>
        | <IF>
        | <LOOP>
        | <CALL>
        | <LET>
        | <TRY> )

<CREATION> ::=
        !! <WRITABLE>.<LOCALCALL>

<IF> ::=
        if <EXPRESSION> then <COMPOUND>
        { elseif <EXPRESSION> then <COMPOUND> }*
        [ else <COMPOUND> ]
        end

<LOOP> ::=
        from
        <COMPOUND>
        until
        <EXPRESSION>
        loop
        <COMPOUND>
        end

<CALL> ::=
        ( <LOCALCALL>
        | <LOCAL>.<LOCALCALL> )

<LET> ::=
        <WRITABLE> := <EXPRESSION>

<TRY> ::=
        <WRITABLE> ?= <EXPRESSION>

<WRITABLE> ::=
        ( <ATTRIBUTE> | <LOCAL> )
```

```
<LOCAL> ::=
        ( <IDENTIFIER> | result )

<LOCALCALL> ::=
        <PATTERNORCOMPONENTFEATURE> [‘(’ EXPRESSION { , EXPRESSION }* ‘)’ ]

<EXPRESSION> ::=
        ( <CONSTANT>
        | ‘(’ <EXPRESSION> ‘)’
        | <UNARY> <EXPRESSION>
        | <CALL>
        | <EXPRESSION> <BINARY> <EXPRESSION>

<CONSTANT> ::=
        ( <INTEGER> | <REAL> | <STRING> )

<INTEGER> ::=
        [ - ] <NAT>

<REAL> ::=
        <INTEGER> . <NAT>

<UNARY> ::=
        ( - | not )

<BINARY> ::=
        ( = | /= | + | - | * | / | or | and )
```

A.2 Die Steuerung der Übersetzung mit c.pra

```
% Datei:         c.pra
% Dokumentation: Unterkapitel 4.1.4 Aufbau des Übersetzers

readpal :         Refine ./pal By ./terminal.
transformation1 : Interpret In ./flat &debug By  ./debug.
transformation2 : ./classes.
writeeiffel :     Interpret
                  In Refine ./gen  By Refine ./classname By lib/conv
                  &writeclass By ./writecl.

Do

 Reading ./example/source.pal Do
  Run readpal -> PAT*
 End Reading;

 Nl; Nl;

% Write PAT*; Nl; Nl;

 Run transformation1(PAT*) -> FLAT*;

% Write FLAT*; Nl; Nl;

 Run transformation2(FLAT*) -> CLASS*;

% Write CLASS*; Nl; Nl;

 Run writeeiffel(CLASS*);

End.
```

A.3 Das Parsen mit pal.gs

```
% Datei:         pal.gs
% Dokumentation: Unterkapitel 4.2 Das Parsen

Axiom Is main

main: -> PAT*

[r0000] main -> PAT*
         :
           patterns -> PAT*,
           end.

[r0010] patterns -> [PAT|PAT*]
         :
           pattern -> PAT,
           patterns -> PAT*.

[r0011] patterns -> [] : .

% Parsen eines Patterns

[r0012] pattern -> <IDpname, PBODY>
         :
           "pattern",
           id -> IDpname,
           pbody -> PBODY,
           "end".

[r0020] pbody -> <REFINE*, IDcreation*, COMP*, PFEAT*>
         :
           refines -> REFINE*,
            creations -> IDcreation*,
           components -> COMP*,
           patfeatures -> PFEAT*.

[r0030] refines -> [REFINE|REFINE*]
         :
           "refine",
           refine -> REFINE,
           refines2 -> REFINE*.

[r0031] refines -> [] : .



[r0032] refines2 -> [REFINE|REFINE*]
         :
           refine -> REFINE,
           refines2 -> REFINE*.

[r0033] refines2 -> [] : .

[r0034] refine -> <SEL, IDpname, IDref,  REFCL>
         :
           select -> SEL,
           patternref -> (IDpname, IDref),
           refclause -> REFCL.

SEL = sel + nosel + unsel + unsel2

[r0040] select -> sel
         :
           "select".

[r0041] select -> nosel : .

[r0050] patternref -> (IDpname, IDref)
         :
           id -> IDpname,
           "<",
           id -> IDref,
           ">".

[r0051] patternref -> (IDpname, IDpname)
         :
           id -> IDpname.
```

```
[r0060] refclause -> <RENAME*, IDselect*>
         :
           renames -> RENAME*,
           selects -> IDselect*,
           "end".

[r0061] refclause -> <[], []> : .

[r0070] renames -> [RENAME|RENAME*]
         :
           "rename",
           rename -> RENAME,
           renames2 -> RENAME*.

[r0071] renames -> [] : .

[r0072] renames2 -> [RENAME|RENAME*]
         :
           ",",
           rename -> RENAME,
           renames2 -> RENAME*.

[r0073] renames2 -> [] : .

[r0074] rename -> <IDrenbefore, IDrenafer>
         :
           id -> IDrenbefore,
           "as",
           id -> IDrenafer.

[r0080] components -> [COMP|COMP*]
         :
           component -> COMP,
           components -> COMP*.

[r0081] components -> [] : .

[r0090] patfeatures -> [PFEAT|PFEAT*]
         :
           patfeature -> PFEAT,
           patfeatures -> PFEAT*.

[r0091] patfeatures -> [] : .

[r0092] patfeature -> <intern, nosel, FEAT>
         :
           "feature",
           featureclause -> FEAT.

[r0093] patfeature -> <intern, nosel, FEAT>
         :
           "intern",
           "feature",
           featureclause -> FEAT.

[r0094] patfeature -> <extern, nosel, FEAT>
         :
           "extern",
           "feature",
           featureclause -> FEAT.

% Parsen einer Komponente

[r0100] component -> <IDcname, CBODY>
         :
           "component",
           id -> IDcname,
           cbody -> CBODY,
           "end".

[r0110] cbody -> <CAST, INHER*, IDcreation*, FEAT*>
         :
           cast -> CAST,
           inherits -> INHER*,
           creations -> IDcreation*,
           features -> FEAT*.
```

```
[r0120] cast -> <CASTREN*, CASTSEL*>
         :
           "cast",
           castrens -> CASTREN*,
           castsels -> CASTSEL*,
           "end".

[r0121] cast -> <[], []> : .

[r0130] castrens -> [CASTREN|CASTREN*]
         :
           "rename",
           castren -> CASTREN,
           castrens2 -> CASTREN*.

[r0131] castrens -> [] : .

[r0132] castrens2 -> [CASTREN|CASTREN*]
         :
           ",",
           castren -> CASTREN,
           castrens2 -> CASTREN*.

[r0133] castrens2 -> [] : .

[r0134] castren -> <IDfnamebefore, IDref, IDfnameafter>
         :
           id -> IDfnamebefore,
           "from",
           "<",
           id -> IDref,
           ">",
           "as",
           id -> IDfnameafter.

[r0135] castren -> <IDfnamebefore, '', IDfnameafter>
         :
           id -> IDfnamebefore,
           "as",
           id -> IDfnameafter.

[r0140] inherits -> [INHER|INHER*]
         :
           "inherit",
           inherit -> INHER,
           inherits2 -> INHER*.

[r0141] inherits -> [] : .

[r0142] inherits2 -> [INHER|INHER*]
         :
           ";",
           inherit -> INHER,
           inherits2 -> INHER*.

[r0143] inherits2 -> [] : .

[r0144] inherit -> <IDcname, []>
         :
           id -> IDcname.

[r0150] castsels -> [CASTSEL|CASTSEL*]
         :
           "select",
           castsel -> CASTSEL,
           castsels2 -> CASTSEL*.

[r0151] castsels -> [] : .

[r0152] castsels2 -> [CASTSEL|CASTSEL*]
         :
           ",",
           castsel -> CASTSEL,
           castsels2 -> CASTSEL*.

[r0153] castsels2 -> [] : .
```

```
[r0154] castsel -> <IDfname, IDref>
         :
           id -> IDfname,
           "from",
           "<",
           id -> IDref,
           ">".

[r0160] selects -> [IDselect|IDselect*]
         :
           "select",
           id -> IDselect,
           featlist -> IDselect*.

[r0161] selects -> [] : .

[r0170] creations -> [IDcreation|IDcreation*]
         :
           "creation",
           id -> IDcreation,
           featlist -> IDcreation*.

[r0171] creations -> [] : .

[r0180] featlist -> [IDfname|IDfname*]
         :
           ",",
           id -> IDfname,
           featlist -> IDfname*.

[r0181] featlist -> [] : .

[r0190] features -> [FEAT|FEAT*]
         :
           feature -> FEAT,
           features -> FEAT*.

[r0191] features -> [] : .

% Parsen eines Features

[r0200] feature -> FEAT
         :
           "feature",
           featureclause -> FEAT.

[r0210] featureclause -> <IDfname, FEATBODY>
         :
           id -> IDfname,
           featurebody -> FEATBODY.

[r0220] featurebody -> attribute(TYPE)
         :
           ":",
           type -> TYPE.

[r0221] featurebody -> procedure([], IMPLEMENT)
         :
           implement -> IMPLEMENT,
           "end".

[r0222] featurebody -> procedure([TYPEDECL|TYPEDECL*], IMPLEMENT)
         :
           "(",
           typedecl -> TYPEDECL,
           typedecls -> TYPEDECL*,
           ")",
           implement -> IMPLEMENT,
           "end".

[r0223] featurebody -> function(TYPE, [], IMPLEMENT)
         :
           ":",
           type -> TYPE,
           implement -> IMPLEMENT,
           "end".
```

```
[r0224] featurebody -> function(TYPE, [TYPEDECL|TYPEDECL*], IMPLEMENT)
         :
           "(",
           typedecl -> TYPEDECL,
           typedecls -> TYPEDECL*,
           ")",
           ":",
           type -> TYPE,
           implement -> IMPLEMENT,
           "end".

[r0230] typedecls -> [TYPEDECL|TYPEDECL*]
         :
           ";",
           typedecl -> TYPEDECL,
           typedecls -> TYPEDECL*.

[r0231] typedecls -> [] : .

% Typdeklaration, siehe Unterkapitel 4.2.4 Typdeklarationen

[r0232] typedecl -> <IDvariable, TYPE>
         :
           id -> IDvariable,
           ":",
           type -> TYPE.

[r0240] type -> type('', IDtype)
         :
           id -> IDtype.

[r0250] implement -> <TYPEDECL*, SOURCE>
         :
           "is",
           locals -> TYPEDECL*,
           source -> SOURCE.

[r0260] locals -> [TYPEDECL|TYPEDECL*]
         :
           "local",
           typedecl -> TYPEDECL,
           typedecls -> TYPEDECL*.

[r0261] locals -> [] : .

% Parsen der Implementation eines Features

[r0300] source -> <deferred, []>
         :
           "deferred".

[r0301] source -> <instructions(INSTRUCTION*), CRITFEAT*>
         :

           "do",
           compound([]) -> (INSTRUCTION*, CRITFEAT*).

[r0310] compound(CRITFEAT*) -> ([INSTRUCTION|INSTRUCTION*], CRITFEAT''*)
         :
           instr(CRITFEAT*) -> (INSTRUCTION, CRITFEAT'*),
           instrs(CRITFEAT'*) ->  (INSTRUCTION*, CRITFEAT''*).

[r0311] compound(CRITFEAT*) -> ([], CRITFEAT*) : .

[r0320] instrs(CRITFEAT*) -> ([INSTRUCTION|INSTRUCTION*], CRITFEAT''*)
         :
           ";",
           instr(CRITFEAT*) -> (INSTRUCTION, CRITFEAT'*),
           instrs(CRITFEAT'*) ->  (INSTRUCTION*, CRITFEAT''*).

[r0321] instrs(CRITFEAT*) -> ([], CRITFEAT*) : .

[r0322] instr(CRITFEAT*) -> (creation(LOCALCALL), CRITFEAT'*)
         :
           "!!",
           id -> IDvariable,
           ".",
           localcall(IDvariable,CRITFEAT* ++[<'current', '' ,IDvariable>])->(LOCALCALL,CRITFEAT'*,_).
```

```
[r0323] instr(CRITFEAT*) -> (if([<EXPRESSION,INSTRUCTION1*>|IFTHEN*], INSTRUCTION2*), CRITFEAT4*)
         :
           "if",
           expression(CRITFEAT*) -> (EXPRESSION, CRITFEAT1*),
           "then",
           compound(CRITFEAT1*) -> (INSTRUCTION1*, CRITFEAT2*),
           elseif(CRITFEAT2*) -> (IFTHEN*, CRITFEAT3*),
           else(CRITFEAT3*) -> (INSTRUCTION2*, CRITFEAT4*),
           "end".

[r0324] instr(CRITFEAT*) -> (loop(INSTRUCTION1*, EXPRESSION, INSTRUCTION2*), CRITFEAT3*)
         :
           "from",
           compound(CRITFEAT*) -> (INSTRUCTION1*, CRITFEAT1*),
           "until",
           expression(CRITFEAT1*) -> (EXPRESSION, CRITFEAT2*),
           "loop",
           compound(CRITFEAT2*) -> (INSTRUCTION2*, CRITFEAT3*),
           "end".

[r0325] instr(CRITFEAT*) -> (instrcall(CALL), CRITFEAT'*)
         :
           call(CRITFEAT*) -> (CALL, CRITFEAT'*).

[r0326] instr(CRITFEAT*) -> (let(LOCALCALL, EXPRESSION), CRITFEAT''*)
         :
           localcall('current', CRITFEAT*) -> (LOCALCALL, CRITFEAT'*, _),
           ":=",
           expression(CRITFEAT'*) -> (EXPRESSION, CRITFEAT''*).

[r0327] instr(CRITFEAT*) -> (try(LOCALCALL, EXPRESSION), CRITFEAT''*)
         :
           localcall('current', CRITFEAT*) -> (LOCALCALL, CRITFEAT'*, _),
           "?=",
           expression(CRITFEAT'*) -> (EXPRESSION, CRITFEAT''*).

[r0330] elseif(CRITFEAT*) -> ([<EXPRESSION,INSTRUCTION*>|IFTHEN*], CRITFEAT3*)
         :
           "elseif",
           expression(CRITFEAT*) -> (EXPRESSION, CRITFEAT1*),
           "then",
           compound(CRITFEAT1*) -> (INSTRUCTION*, CRITFEAT2*),
           elseif(CRITFEAT2*) -> (IFTHEN*, CRITFEAT3*).

[r0331] elseif(CRITFEAT*) -> ([], CRITFEAT*) : .

[r0332] else(CRITFEAT*) -> (INSTRUCTION*, CRITFEAT'*)
         :
           "else",
           compound(CRITFEAT*) -> (INSTRUCTION*, CRITFEAT'*).

[r0333] else(CRITFEAT*) -> ([], CRITFEAT*) : .

% Feature- Aufrufe, siehe Unterkapitel 4.2.3 Sonderbehandlung der Feature- Calls

[r0340] call(CRITFEAT*) -> (call1(LOCALCALL), CRITFEAT'*)
         :
           localcall('current', CRITFEAT*) -> (LOCALCALL, CRITFEAT'*, _).

[r0341] call(CRITFEAT*) -> (call2(LOCALCALL1, LOCALCALL2), CRITFEAT''*)
         :
           localcall('current', CRITFEAT*) -> (LOCALCALL1, CRITFEAT'*, IDvariable),
           ".",
           localcall(IDvariable, CRITFEAT'*) -> (LOCALCALL2, CRITFEAT''*, _).

[r0342] localcall(IDvariable, CRITFEAT*) -> (loccall(EXPRESSION*), CRITFEAT'*, IDfname)
         :
           id -> IDfname,
           "(",
           expressions(CRITFEAT* ++ [<IDvariable, '', IDfname>]) -> (EXPRESSION*, CRITFEAT'*),
           ")".

[r0343] localcall(IDvariable,CRITFEAT*)->(loccall([]),CRITFEAT* ++[<IDvariable,'',IDfname>],IDfname)
         :
           id -> IDfname.

% Ausdruecke, siehe Unterkapitel 4.4.2 Linksrekursive Ausdruecke
```

```
[r0350] expressions(CRITFEAT*) -> ([EXPRESSION|EXPRESSION*], CRITFEAT''*)
         :
           expression(CRITFEAT*) -> (EXPRESSION, CRITFEAT'*),
           expressions2(CRITFEAT'*) -> (EXPRESSION*, CRITFEAT''*).

[r0351] expressions2(CRITFEAT*) -> ([EXPRESSION|EXPRESSION*], CRITFEAT''*)
         :
           ",",
           expression(CRITFEAT*) -> (EXPRESSION, CRITFEAT'*),
           expressions2(CRITFEAT'*) -> (EXPRESSION*, CRITFEAT''*).

[r0352] expressions2(CRITFEAT*) -> ([], CRITFEAT*) : .

[r0353] expression(CRITFEAT*) -> (simpleexpr(EXPRESSION), CRITFEAT'*)
         :
           sexpression(CRITFEAT*) -> (EXPRESSION, CRITFEAT'*).

[r0354] expression(CRITFEAT*) -> (rekexpr(EXPRESSION), CRITFEAT'*)
         :
           rexpression(CRITFEAT*) -> (EXPRESSION, CRITFEAT'*).

[r0355] sexpression(CRITFEAT*) -> (exprconst(CONSTANT), CRITFEAT*)
         :
           constant -> CONSTANT.

[r0356] sexpression(CRITFEAT*) -> (expr(EXPRESSION), CRITFEAT'*)
         :
           "(",
           expression(CRITFEAT*) -> (EXPRESSION, CRITFEAT'*),
           ")".

[r0357] sexpression(CRITFEAT*) -> (unexpr(UNARY, EXPRESSION), CRITFEAT'*)
         :
           unary -> UNARY,
           expression(CRITFEAT*) -> (EXPRESSION, CRITFEAT'*).

[r0358] sexpression(CRITFEAT*) -> (exprcall(CALL), CRITFEAT'*)
         :
           call(CRITFEAT*) -> (CALL, CRITFEAT'*).

[r0359] rexpression(CRITFEAT*) -> (binexpr(EXPRESSION1, BINARY, EXPRESSION2), CRITFEAT''*)
         :
           sexpression(CRITFEAT*) -> (EXPRESSION1, CRITFEAT'*),
           binary -> BINARY,
           expression(CRITFEAT'*) -> (EXPRESSION2, CRITFEAT''*).

[r0360] unary -> '-' : "-".

[r0361] unary -> 'not' : "not".

[r0370] binary -> '=' : "=".

[r0371] binary -> '/=' : "/=".

[r0372] binary -> '+' : "+".

[r0373] binary -> '-' : "-".

[r0374] binary -> '*' : "*".

[r0375] binary -> '/' : "/".

[r0376] binary -> 'or' : "or".

[r0377] binary -> 'and' : "and".


[r0380] constant -> constint(INTEGER)
         :
           integer -> INTEGER.

[r0381] constant -> constreal(REAL)
         :
           real -> REAL.

[r0382] constant -> conststring(STRING)
         :
           string -> STRING.
```

```
[r0390] integer -> <’-’, NAT>
         :
           "-",
           nat -> NAT.

[r0391] integer -> <’’, NAT>
         :
           nat -> NAT.

[r0400] real -> <INTEGER, NAT>
         :
           integer -> INTEGER,
           ".",
           nat -> NAT.
```

A.4 Die Erkennung von Morphemklassen mit terminal.lg

```
% Datei:         terminal.lg
% Dokumentation: Unterkapitel 4.2.1 Erkennung von Morphemklassen

Sets
 letter         = ’A’ .. ’Z’ | ’a’ .. ’z’.
 digit          = ’0’ .. ’9’.
 but_eoln       = Any - Eoln.
 but_star       = Any - ’*’.
 but_div_star   = Any - "/*".
 but_unquote    = Any - ’"’.

Classes
 spaces     = (Space | Tab | Eoln)+.
 id         = (letter | (letter (letter | digit | ’_’)* (letter | digit))) : lib/conv
chars2identifier.
 nat        = digit+                    : lib/conv chars2integer.
 end        = Eof.
 comment    = ’/’ ’*’ ( but_star | ’*’+ but_div_star )* ’*’+ ’/’
            | ’--’ but_eoln*.
 string     = ’"’ ( but_unquote )* ’"’ : lib/conv charsQuoted2string.

Switches
 Skip spaces.
 Skip comment.
```

A.5 Die erste Transformation mit flat.ir

```
% Datei:         flat.ir
% Dokumentation: Unterkapitel 4.3 Die erste Transformation

Axiom Is main

SEL = sel + nosel + unsel + unsel2

main: PAT* -> FLAT*

% Bei der Transformationen werden die Pattern flach gemacht und entsprechend der Refines geordnet

[r0000] list(PAT*,[]) -> FLAT*
        ----------------------
        main(PAT*) -> FLAT*

[r0010] pat2flat(PAT, FLAT*) -> FLAT,
        list(PAT'* ++ PAT''*, FLAT* ++ [FLAT]) -> FLAT'*
        ------------------------------------------------
        list(PAT'* ++ [PAT] ++ PAT''*, FLAT*) -> FLAT'*

[r0011] list([], FLAT*) -> FLAT*

% Zum Ordnen der Pattern und zum Aufloesen der Refines werden die Parameter noch weiter zerlegt:
% Es werden die Refines isoliert, alle bereits flachen Pattern uebergeben, das aktuelle Pattern ohne
% Refines (also primitiv flach) und die temporaere Liste der bereits realisierten Refines.

[r0020] refinesinflat(REFINE*, COMP*, FLAT*, <IDpname, <[], IDcreation*, COMP*, PFEAT*>>, []) -> <IDpname', <INHER*, IDcreation'*, COMP'*, PFEAT'*>>
        ------------------------------------------------------------------------------------------------------------------------------------------
        pat2flat(<IDpname, <REFINE*, IDcreation*, COMP*, PFEAT*>>, FLAT*) -> <IDpname', <INHER*, IDcreation'*, COMP'*, PFEAT'*>>

% Um ein Refine zu realisieren, wird das Rename auf die Components (Name, Namen der Eltern und
% alle Typen) und Features angewendet. Dnach werden alle Selects gesetzt.
% Dieser Vorgang wird so oft wiederholt, bis keine Refines mehr existieren.

[r0030] FLAT* = _ ++ [<@IDpname, <_, IDcreation*, COMP2*, PFEAT1*>>] ++ _,
        renamecomp(RENAME*, COMP2*) -> (RENAME'*, COMP3*),
        rencomptyp(RENAME*, COMP3*) -> COMP4*,
        renamepfeat(RENAME'*, PFEAT1*) -> PFEAT2*,
        renamepfeattyp(RENAME*, PFEAT2*) -> PFEAT3*,
        selectpfeat(SEL, IDselect*, PFEAT3*) -> (IDselect'*, PFEAT4*),
        inhercasts(COMP*, COMP4*, []) -> COMP1*,
        castpfeat(COMP1*, PFEAT4*, IDref) -> PFEAT5*,
        castcomp(COMP1*, COMP4*, IDref, []) -> COMP5*,
        setselect(COMP1*, SEL, IDselect'*, COMP5*, IDref, []) -> FULLCOMP*,
        refinesinflat(REFINE*, COMP*,  FLAT*, FLAT, [<'', <[<IDpname, []>], IDcreation*, FULLCOMP*, PFEAT5*>>|TEMPPAT*]) -> FLAT'
        ------------------------------------------------------------------------------------------------------------------------
        refinesinflat([<SEL, IDpname, IDref, <RENAME*, IDselect*>>|REFINE*], COMP*, FLAT*, FLAT, TEMPPAT*) -> FLAT'
```

```
% Wenn es keine Refines mehr gibt, werden die temporaeren Pattern und das aktuelle abgeflachte Pattern
% zusammengemischt.

[r0031] comp2fcomp(COMP*) -> FULLCOMP*,
        merge(<IDpname, <[], IDcreation*, FULLCOMP*, PFEAT*>>, TEMPPAT*) -> FLAT'
        ---------------------------------------------------------------------------
        refinesinflat([], _, _, <IDpname, <_, IDcreation*, COMP*, PFEAT*>>, TEMPPAT*) -> FLAT'

% Dokumentation der Regeln [r1000] bis [r1093] siehe Unterkapitel 4.3.1 Die Umbenennung der Komponenten
% Das Rename auf Components wird so lange angewendet, bis kein Rename mehr auf einen Component- Namen
% zutrifft. Das aktuell angewendete Rename wird aus der mange der Renames entfernt.

[r1000] COMP* = COMP'* ++ [<@IDrenbefore, CBODY>] ++ COMP''*,
        renamecomp(RENAME* ++ RENAME'*, COMP'* ++ COMP''*) -> (RENAME''*, COMP2*)
        ---------------------------------------------------------------------------------------------------
        renamecomp(RENAME* ++ [<IDrenbefore, IDrenafer>] ++ RENAME'*, COMP*) -> (RENAME''*, [<IDrenafer, CBODY>|COMP2*])

% Die restlichen Components werden ohne Aenderungen uebernommen. Die uebbrig gebliebenen Renames sind zum
% Umbenennen der Patternfeatures.

[r1001] renamecomp(RENAME*, COMP*) -> (RENAME*, COMP*)

% Zum Umbenennen der Namen der Eltern der Components in der Inherit- Klausel wird wieder die gesamte
% Menge der Renames herangezogen und auf die Inherit- Klausel jeder einzelnen Componente angewendet.

[r1010] renameinh(RENAME*, INHER*) -> INHER'*,
        renamefeattyp([<'currentcomponent', IDcname>|RENAME*], FEAT*) -> FEAT'*,
        rencomptyp(RENAME*, COMP*) -> COMP'*
        ---------------------------------------------------------------------------------------------------------------------------
        rencomptyp(RENAME*, [<IDcname, <CAST, INHER*, IDcreation*, FEAT*>>|COMP*]) -> [<IDcname, <CAST, INHER'*, IDcreation*, FEAT'*>>|COMP'*]

[r1011] rencomptyp(_, []) -> []

% Es wird solange versucht ein Rename aus der Menge der Renames auf einen Elternnamen anzuwenden, bis
% keines mehr zutrifft.

[r1020] RENAME* = _ ++ [<@IDrenbefore, IDrenafer>] ++ _,
        renameinh(RENAME*, INHER* ++ INHER'*) -> INHER''*
        ---------------------------------------------------------------------------
        renameinh(RENAME*, INHER* ++ [<IDrenbefore, _>] ++ INHER'*) -> [<IDrenafer, []>|INHER''*]

% Die Restlichen Inherits werden direkt uebernommen.

[r1021] renameinh(_, INHER*) -> INHER*

[r1030] renfeatbody(RENAME*, intern, FEATBODY) -> FEATBODY',
        renamefeattyp(RENAME*, FEAT*) -> FEAT'*
        ---------------------------------------------------------------------------
        renamefeattyp(RENAME*, [<IDfname, FEATBODY>|FEAT*]) -> [<IDfname, FEATBODY'>|FEAT'*]
```

```
[r1031] renamefeattyp(_, []) -> []

[r1040] renametype(RENAME*, TYPE) -> TYPE'
        ------------------------------------------------------------
        renfeatbody(RENAME*, intern, attribute(TYPE)) -> attribute(TYPE')

[r1041] renametypeerror(RENAME*, TYPE)
        ------------------------------------------------------------
        renfeatbody(RENAME*, extern, attribute(TYPE)) -> attribute(TYPE)

[r1042] rentypedecls(RENAME*, TYPEDECL*) -> TYPEDECL'*,
        renameimpl(RENAME*, IMPLEMENT) -> IMPLEMENT'
        ------------------------------------------------------------------------------------------
        renfeatbody(RENAME*, intern, procedure(TYPEDECL*, IMPLEMENT)) -> procedure(TYPEDECL'*, IMPLEMENT')

[r1043] rentypedeclserror(RENAME*, TYPEDECL*),
        renameimpl(RENAME*, IMPLEMENT) -> IMPLEMENT'
        ------------------------------------------------------------------------------------------
        renfeatbody(RENAME*, extern, procedure(TYPEDECL*, IMPLEMENT)) -> procedure(TYPEDECL*, IMPLEMENT')

[r1044] renametype(RENAME*, TYPE) -> TYPE',
        rentypedecls(RENAME*, TYPEDECL*) -> TYPEDECL'*,
        renameimpl(RENAME*, IMPLEMENT) -> IMPLEMENT'
        ------------------------------------------------------------------------------------------------------------
        renfeatbody(RENAME*, intern, function(TYPE, TYPEDECL*, IMPLEMENT)) -> function(TYPE', TYPEDECL'*, IMPLEMENT')

[r1045] renametypeerror(RENAME*, TYPE),
        rentypedeclserror(RENAME*, TYPEDECL*),
        renameimpl(RENAME*, IMPLEMENT) -> IMPLEMENT'
        ------------------------------------------------------------------------------------------------------------
        renfeatbody(RENAME*, extern, function(TYPE, TYPEDECL*, IMPLEMENT)) -> function(TYPE, TYPEDECL*, IMPLEMENT')

[r1050] renametype(RENAME*, TYPE) -> TYPE',
        rentypedecls(RENAME*, TYPEDECL*) -> TYPEDECL'*
        ------------------------------------------------------------------------------
        rentypedecls(RENAME*, [<IDvariable, TYPE>|TYPEDECL*]) -> [<IDvariable, TYPE'>|TYPEDECL'*]

[r1051] rentypedecls(_, []) -> []

[r1052] renametypeerror (RENAME*, TYPE),
        rentypedeclserror (RENAME*, TYPEDECL*)
        ------------------------------------------------------
        rentypedeclserror (RENAME*, [<_, TYPE>|TYPEDECL*])

[r1053] rentypedeclserror (_, [])

[r1060] rentypedecls(RENAME*, TYPEDECL*) -> TYPEDECL'*,
        renamesource(RENAME*, CRITFEAT*) ->  CRITFEAT'*
        ------------------------------------------------------------------------------------
        renameimpl(RENAME*, <TYPEDECL*, <COMPOUND, CRITFEAT*>>) -> <TYPEDECL'*, <COMPOUND, CRITFEAT'*>>
```

```
[r1070] rencrittype(RENAME*, CRITFEAT) -> CRITFEAT',
        rencritfeat(RENAME*, CRITFEAT') -> CRITFEAT'',
        renamesource(RENAME*, CRITFEAT*) ->  CRITFEAT'*
        ------------------------------------------------------------------
        renamesource(RENAME*, [CRITFEAT|CRITFEAT*]) ->  [CRITFEAT''|CRITFEAT'*]

[r1071] renamesource(_, []) ->  []

[r1080] rencrittype(_ ++ [<IDrenbefore, IDrenafter>] ++ _, <IDvariable, @IDrenbefore, IDfname>) -> <IDvariable, IDrenafter, IDfname>

[r1081] rencrittype(_, CRITFEAT) -> CRITFEAT

[r1090] renametype (_ ++ [<IDrenbefore, IDrenafter>] ++ _, type(IDprefix, @IDrenbefore)) -> type(IDprefix, IDrenafter)

[r1091] renametype(_, TYPE) -> TYPE

[r1092] &debug externparam(<IDrenbefore, IDrenafter>)
        ------------------------------------------------------------------------
        renametypeerror (_ ++ [<IDrenbefore, IDrenafter>] ++ _, type(_, @IDrenbefore))

[r1093] renametypeerror (_, _)

% Dokumentation der Regeln [r1100] bis [r1112] siehe Unterkapitel 4.3.2 Die Umbenennung der Patternfeatures

[r1100] CRITFEAT = <IDvariable, 'currentpattern', @IDrenbefore>
        ------------------------------------------------------------------------------------------------
        rencritfeat(_ ++ [<IDrenbefore, IDrenafter>] ++ _, CRITFEAT) -> <IDvariable, 'currentpattern', IDrenafter>

[r1101] rencritfeat(_, CRITFEAT) -> CRITFEAT

% Zum Umbenennen der Patternfeatures werden die Renames angewendet, die beim Umbenennen der Components
% uebriggeblieben sind. Es muessen alle Renames umgesetzt werden.

[r1110] PFEAT* = PFEAT'* ++ [<intern, _, <@IDrenbefore, FEATBODY>>] ++ PFEAT''*,
        renamepfeat(RENAME* ++ RENAME'*, PFEAT'* ++ PFEAT''*) -> PFEAT2*
        ------------------------------------------------------------------------------------------------------------------
        renamepfeat(RENAME* ++ [<IDrenbefore, IDrenafter>] ++ RENAME'*, PFEAT*) -> [<intern, nosel, <IDrenafter, FEATBODY>>|PFEAT2*]

% Wenn alle Renames angewendet wurden, werden die restlichen Patternfeatures ohne Aenderung uebernommen.

[r1111] renamepfeat([], PFEAT*) -> PFEAT*

[r1112] &debug unknownren(RENAME),
        renamepfeat(RENAME*, []) -> []
        ----------------------------------------
        renamepfeat([RENAME|RENAME*], PFEAT*) -> PFEAT*

% Dokumentation der Regeln [r1120] bis [r1121] siehe Unterkapitel 4.3.1 Die Umbenennung der Komponenten
```

```
[r1120] renfeatbody(RENAME*, EXTINT, FEATBODY) -> FEATBODY',
        renamepfeattyp(RENAME*, PFEAT*) -> PFEAT'*
        ------------------------------------------------------------------------------------------------------------
        renamepfeattyp(RENAME*, [<EXTINT, _, <IDfname, FEATBODY>>|PFEAT*]) -> [<EXTINT, nosel, <IDfname, FEATBODY'>>|PFEAT'*]

[r1121] renamepfeattyp(_, []) -> []

% Dokumentation der Regeln [r1130] bis [r1133] siehe Unterkapitel 4.3.4 Die Selektion von Patternfeatures

[r1130] PFEAT* = PFEAT'* ++ [<EXTINT, _, <@IDfname, FEATBODY>>] ++ PFEAT''*,
        selectpfeat(SEL, IDselect* ++ IDselect'*, PFEAT'* ++ PFEAT''*) -> (IDselect''*, PFEAT2*)
        ------------------------------------------------------------------------------------------------------------
        selectpfeat(SEL, IDselect* ++ [IDfname] ++ IDselect'*, PFEAT*) -> (IDselect''*, [<EXTINT, sel, <IDfname, FEATBODY>>|PFEAT2*])

[r1131] selectpfeat(sel, IDselect*, PFEAT*) -> (IDselect'*, PFEAT'*)
        ------------------------------------------------------------------------------------------
        selectpfeat(sel, IDselect*, [<EXTINT, _, FEAT>|PFEAT*]) -> (IDselect'*, [<EXTINT, sel, FEAT>|PFEAT'*])

[r1132] selectpfeat(nosel, IDselect*, PFEAT*) -> (IDselect'*, PFEAT'*)
        ------------------------------------------------------------------------------------------
        selectpfeat(nosel, IDselect*, [<EXTINT, _, FEAT>|PFEAT*]) -> (IDselect'*, [<EXTINT, unsel, FEAT>|PFEAT'*])

[r1133] selectpfeat(_, IDselect*, []) -> (IDselect*, [])

% Dokumentation der Regeln [r1140] bis [r1212] siehe Unterkapitel 4.3.3 Die Umbenennung von Komponentenfeatures

[r1140] inhercast(CAST, INHER*, COMP*) -> CAST',
        inhercasts(COMP1* ++ COMP1'*, COMP2* ++ COMP2'*, [<IDcname, <CAST', [], [], []>>|COMP*]) -> COMP'*
        ------------------------------------------------------------------------------------------------------------------------
        inhercasts(COMP1* ++ [<IDcname, <CAST, _, _, _>>] ++ COMP1'*, COMP2* ++ [<@IDcname, <_, INHER*, _, _>>] ++ COMP2'*, COMP*) -> COMP'*

[r1141] inhercast(<[], []>, INHER*, COMP*) -> CAST',
        inhercasts(COMP1*, COMP2* ++ COMP2'*, [<IDcname, <CAST', [], [], []>>|COMP*]) -> COMP'*
        ------------------------------------------------------------------------------
        inhercasts(COMP1*, COMP2* ++ [<IDcname, <_, INHER*, _, _>>] ++ COMP2'*, COMP*) -> COMP'*

[r1142] inhercasts(COMP1*, [], COMP*) -> COMP1* ++ COMP*

[r1143] COMP* = _ ++ [<@IDcname, <<CASTREN2*, _>, _, _, _>>] ++ _,
        inhercast(<CASTREN1* ++ CASTREN2*, CASTSEL*>, INHER*, COMP*) -> CAST
        ------------------------------------------------------------
        inhercast(<CASTREN1*, CASTSEL*>, [<IDcname, _>|INHER*], COMP*) -> CAST

[r1144] inhercast(CAST, [], _) -> CAST

[r1150] castfeatbody(FEATBODY, COMP1*, IDref) -> FEATBODY',
        castpfeat(COMP1*, PFEAT*, IDref) -> PFEAT'*
        ------------------------------------------------------------------------------------------------------------
        castpfeat(COMP1*, [<EXTINT, SEL, <IDfname, FEATBODY>>|PFEAT*], IDref) -> [<EXTINT, SEL, <IDfname, FEATBODY'>>|PFEAT'*]

[r1151] castpfeat(_, [], _) -> []
```

```
% Beim Umbenennen der Features der Components in einer speziellen Verfeinerung wird zuerst ueberprueft,
% ob im Elternpattern dieser Verfeinerung eine Componente unter gleichem Namen existiert. Wenn ja,
% dann werden alle CASTRENames gesucht, die die gleiche Verfeinerungsbezeichnung haben und auf die
% Features passen. Das gleiche wird mit den Creates gemacht. Dieser Vorgang wird so oft wiederholt,
% bis es keine Uebereinstimmungen mehr gibt.

[r1160] COMP1* = _ ++ [<@IDcname, <<CASTREN*, _>, _, _, _>>] ++ _,
        renamecast(CASTREN*, FEAT*, COMP1*, IDref) -> FEAT'*,
        rencreate(CASTREN*, IDcreation*, IDref) -> IDcreation'*,
        castcomp(COMP1*, COMP2*, IDref, [<IDcname, <CAST, INHER*, IDcreation'*, FEAT'*>>|COMP3*]) -> COMP4*
        ------------------------------------------------------------------------------------------
        castcomp(COMP1*, [<IDcname, <CAST, INHER*, IDcreation*, FEAT*>>|COMP2*], IDref, COMP3*) -> COMP4*

[r1161] renamecast([], FEAT*, COMP1*, IDref) -> FEAT'*,
        castcomp(COMP1*, COMP2*, IDref, [<IDcname, <CAST, INHER*, IDcreation*, FEAT'*>>|COMP3*]) -> COMP4*
        ------------------------------------------------------------------------------------------
        castcomp(COMP1*, [<IDcname, <CAST, INHER*, IDcreation*, FEAT*>>|COMP2*], IDref, COMP3*) -> COMP4*

[r1162] castcomp(_, [], _, COMP3*) -> COMP3*

[r1170] CASTREN* = CASTREN'* ++ [<@IDfnamebefore, '', IDfnameafter>] ++ CASTREN''*,
        castfeatbody(FEATBODY, COMP1*, IDref) -> FEATBODY',
        renamecast(CASTREN'* ++ CASTREN''*, FEAT*, COMP1*, IDref) -> FEAT''*
        ------------------------------------------------------------------------------------------------------
        renamecast(CASTREN*, [<IDfnamebefore, FEATBODY>|FEAT*], COMP1*, IDref) -> [<IDfnameafter, FEATBODY'>|FEAT''*]

[r1171] CASTREN* = CASTREN'* ++ [<@IDfnamebefore, @IDref, IDfnameafter>] ++ CASTREN''*,
        castfeatbody(FEATBODY, COMP1*, IDref) -> FEATBODY',
        renamecast(CASTREN'* ++ CASTREN''*, FEAT*, COMP1*, IDref) -> FEAT''*
        ------------------------------------------------------------------------------------------------------
        renamecast(CASTREN*, [<IDfnamebefore, FEATBODY>|FEAT*], COMP1*, IDref) -> [<IDfnameafter, FEATBODY'>|FEAT''*]

[r1172] castfeatbody(FEATBODY, COMP1*, IDref) -> FEATBODY',
        renamecast(CASTREN*, FEAT*, COMP1*, IDref) -> FEAT''*
        ------------------------------------------------------------------------------------------
        renamecast(CASTREN*, [<IDfname, FEATBODY>|FEAT*], COMP1*, IDref) -> [<IDfname, FEATBODY'>|FEAT''*]

[r1173] &debug unknowncastren(<IDfnamebefore, IDref, IDfnameafter>),
        renamecast(CASTREN* ++ CASTREN'*, [], [], IDref) -> FEAT*
        ------------------------------------------------------------------------------------------
        renamecast(CASTREN* ++ [<IDfnamebefore, IDref, IDfnameafter>] ++ CASTREN'*, [], _, @IDref) -> FEAT*

[r1174] renamecast(_, [], _, _) -> []

[r1180] castfeatbody(attribute(TYPE), _, _) -> attribute(TYPE)

[r1181] FEATBODY = procedure(TYPEDECL1*, <TYPEDECL2*, <COMPOUND, CRITFEAT*>>),
        castsimpl(CRITFEAT*, COMP1*, IDref) -> CRITFEAT'*
        ------------------------------------------------------------------------------------------
        castfeatbody(FEATBODY, COMP1*, IDref) -> procedure(TYPEDECL1*, <TYPEDECL2*, <COMPOUND, CRITFEAT'*>>)
```

```
[r1182] FEATBODY = function(TYPE, TYPEDECL1*, <TYPEDECL2*, <COMPOUND, CRITFEAT*>>),
        castsimpl(CRITFEAT*, COMP1*, IDref) -> CRITFEAT'*
        ------------------------------------------------------------------------------------------------------
        castfeatbody(FEATBODY, COMP1*, IDref) -> function(TYPE, TYPEDECL1*, <TYPEDECL2*, <COMPOUND, CRITFEAT'*>>)

[r1190] castimpl(CASTREN*, CRITFEAT1*, IDcname, IDref) -> CRITFEAT2*,
        castsimpl(CRITFEAT2*, COMP1*, IDref) -> CRITFEAT3*
        ------------------------------------------------------------------------------------------------
        castsimpl(CRITFEAT1*, [<IDcname, <<CASTREN*, _>, _, _, _>>|COMP1*], IDref) -> CRITFEAT3*

[r1191] castsimpl(CRITFEAT*, [], _) -> CRITFEAT*

[r1192] CRITFEAT* = [<IDvariable, @IDtype, IDfnamebefore>|CRITFEAT1*],
        castcritfeat(CASTREN*, IDfnamebefore, IDref) -> IDfnameafter,
        castimpl(CASTREN*, CRITFEAT1*, IDtype, IDref) -> CRITFEAT'*
        ---------------------------------------------------------------------------------------
        castimpl(CASTREN*, CRITFEAT*, IDtype, IDref) -> [<IDvariable, IDtype, IDfnameafter>|CRITFEAT'*]

[r1193] CRITFEAT* = [<IDvariable, IDtype1, IDfname>|CRITFEAT1*],
        castimpl(CASTREN*, CRITFEAT1*, IDtype2, IDref) -> CRITFEAT'*
        ---------------------------------------------------------------------------------------
        castimpl(CASTREN*, CRITFEAT*, IDtype2, IDref) -> [<IDvariable, IDtype1, IDfname>|CRITFEAT'*]

[r1194] castimpl(_, [], _, _) -> []

[r1200] castcritfeat(_ ++ [<IDfnamebefore, '', IDfnameafter>] ++ _, @IDfnamebefore, _) -> IDfnameafter

[r1201] castcritfeat(_ ++ [<IDfnamebefore, IDref, IDfnameafter>] ++ _, @IDfnamebefore, @IDref) -> IDfnameafter

[r1202] castcritfeat(_, IDfname, _) -> IDfname

% Wie die Features werden auch die Creates umbenannt.

[r1210] CASTREN* = CASTREN'* ++ [<@IDfnamebefore, '', IDfnameafter>] ++ CASTREN''*,
        rencreate(CASTREN'* ++ CASTREN''*, IDcreation* ++ IDcreation'*, IDref) -> IDcreation''*
        ------------------------------------------------------------------------------------------------
        rencreate(CASTREN*, IDcreation* ++ [IDfnamebefore] ++ IDcreation'*, IDref) -> [IDfnameafter|IDcreation''*]

[r1211] CASTREN* = CASTREN'* ++ [<@IDfnamebefore, @IDref, IDfnameafter>] ++ CASTREN''*,
        rencreate(CASTREN'* ++ CASTREN''*, IDcreation* ++ IDcreation'*, IDref) -> IDcreation''*
        ------------------------------------------------------------------------------------------------
        rencreate(CASTREN*, IDcreation* ++ [IDfnamebefore] ++ IDcreation'*, IDref) -> [IDfnameafter|IDcreation''*]

[r1212] rencreate(_, IDcreation*, _) -> IDcreation*

% Dokumentation der Regeln [r1220] bis [r1282] siehe Unterkapitel 4.3.4 Die Selektion von Komponentenfeatures
```

```
[r1220] COMP1* = _ ++ [<@IDcname, <<_, CASTSEL*>, _, _, _>>] ++ _,
        testselect(SEL, IDselect*, IDcname) -> <SEL', IDselect'*>,
        inhinfullc(CASTSEL*, SEL', <IDcname, CBODY>, IDref, FULLCOMP*, []) -> FULLFEAT*,
        setselect(COMP1*, SEL, IDselect'*, COMP* ++ COMP'*, IDref, [<IDcname, FULLFEAT*, CBODY>|FULLCOMP*]) -> FULLCOMP'*
        ------------------------------------------------------------------------------------------------------------
        setselect(COMP1*, SEL, IDselect*, COMP* ++ [<IDcname, CBODY>] ++ COMP'*, IDref, FULLCOMP*) -> FULLCOMP'*

[r1221] testselect(SEL, IDselect*, IDcname) -> <SEL', IDselect'*>,
        inhinfullc([], SEL', <IDcname, CBODY>, IDref, FULLCOMP*, []) -> FULLFEAT*,
        setselect(COMP1*, SEL, IDselect'*, COMP*, IDref, [<IDcname, FULLFEAT*, CBODY>|FULLCOMP*]) -> FULLCOMP'*
        ------------------------------------------------------------------------------------------
        setselect(COMP1*, SEL, IDselect*, [<IDcname, CBODY>|COMP*], IDref, FULLCOMP*) -> FULLCOMP'*

[r1222] setselect(_, _, [], [], _, FULLCOMP*) -> FULLCOMP*

[r1223] &debug unknownsel(IDselect*)
        ------------------------------------------------------
        setselect(_, _, IDselect*, [], _, FULLCOMP*) -> FULLCOMP*

[r1230] testselect(_, IDselect* ++ [IDcname] ++ IDselect'*, @IDcname) -> <sel, IDselect* ++ IDselect'*>

[r1231] testselect(sel, IDselect*, _) -> <sel, IDselect*>

[r1232] testselect(_, IDselect*, _) -> <nosel, IDselect*>

[r1240] FULLCOMP* = _ ++ [<@IDcname2, FULLFEAT3*, _>] ++ _,
        inherfullf(FULLFEAT3*, FULLFEAT1*) -> FULLFEAT1'*,
        inhinfullc(CASTSEL*, SEL, <IDcname1, <CAST, INHER*, IDcreation*, FEAT*>>, IDref, FULLCOMP*, FULLFEAT1'*) -> FULLFEAT2*
        ------------------------------------------------------------------------------------------------------------------------
        inhinfullc(CASTSEL*, SEL, <IDcname1, <CAST, [<IDcname2, _>|INHER*], IDcreation*, FEAT*>>, IDref, FULLCOMP*, FULLFEAT1*) -> FULLFEAT2*

[r1241] feat2fullf(FEAT*, FULLFEAT1*) -> FULLFEAT1'*,
        setfeatsel(FULLFEAT1'*, SEL, CASTSEL*, IDref) -> FULLFEAT2*
        ---------------------------------------------------------------------------
        inhinfullc(CASTSEL*, SEL, <_, <_, [], _, FEAT*>>, IDref, _, FULLFEAT1*) -> FULLFEAT2*

[r1250] feat2fullf(FEAT*, FULLFEAT1* ++ [<IDfname, nosel>] ++ FULLFEAT1'*) -> FULLFEAT2*
        ---------------------------------------------------------------------------------------
        feat2fullf([<IDfname, _>|FEAT*], FULLFEAT1* ++ [<@IDfname, _>] ++ FULLFEAT1'*) -> FULLFEAT2*

[r1251] feat2fullf(FEAT*, [<IDfname, nosel>|FULLFEAT1*]) -> FULLFEAT2*
        ------------------------------------------------------
        feat2fullf([<IDfname, _>|FEAT*], FULLFEAT1*) -> FULLFEAT2*

[r1252] feat2fullf([], FULLFEAT*) -> FULLFEAT*

[r1260] inher1feat(FULLFEAT, FULLFEAT1*) -> FULLFEAT1'*,
        inherfullf(FULLFEAT*, FULLFEAT1'*) -> FULLFEAT2*
        ------------------------------------------------------
        inherfullf([FULLFEAT|FULLFEAT*], FULLFEAT1*) -> FULLFEAT2*
```

```
[r1261] inherfullf([], FULLFEAT*) -> FULLFEAT*

[r1262] inher1feat(<IDfname, _>, FULLFEAT* ++ [<@IDfname, _>] ++ FULLFEAT'*) -> FULLFEAT* ++ [<IDfname, nosel>] ++ FULLFEAT'*

[r1263] inher1feat(<IDfname, _>, FULLFEAT*) -> [<IDfname, nosel>|FULLFEAT*]

[r1270] set1featsel(FULLFEAT1, SEL, CASTSEL1*, IDref) -> <FULLFEAT2, CASTSEL2*>,
        setfeatsel(FULLFEAT1*, SEL, CASTSEL2*, IDref) -> FULLFEAT2*
        ------------------------------------------------------------------------------
        setfeatsel([FULLFEAT1|FULLFEAT1*], SEL, CASTSEL1*, IDref) -> [FULLFEAT2|FULLFEAT2*]

[r1271] &debug unknowncastsel(<IDfname, IDref>),
        setfeatsel([], SEL, CASTSEL* ++ CASTSEL'*, IDref) -> FULLFEAT2*
        ------------------------------------------------------------------------------
        setfeatsel([], SEL, CASTSEL* ++ [<IDfname, IDref>] ++ CASTSEL'*, @IDref) -> FULLFEAT2*

[r1272] setfeatsel([], _, _, _) -> []

[r1280] set1featsel(<IDfname, _>, _, CASTSEL* ++ [<@IDfname, IDref>] ++ CASTSEL'*, @IDref) -> <<IDfname, sel>, CASTSEL* ++ CASTSEL'*>

[r1281] set1featsel(<IDfname, _>, sel, CASTSEL*, _) -> <<IDfname, sel>, CASTSEL*>

[r1282] set1featsel(<IDfname, _>, _, CASTSEL*, _) -> <<IDfname, unsel>, CASTSEL*>

%%%%%%%%%%%%%%%%%%%%%%%%%%

% Dokumentation der Regeln [r2000] bis [r2020] siehe Unterkapitel 4.3.6 Das Zusammensetzen des Patterns

[r2000] comp2fcomp(COMP*) -> FULLCOMP*
        ------------------------------------------------------------------------------------------------------------------------
        comp2fcomp([<IDcname, <_, INHER*, IDcreation*, FEAT*>>|COMP*]) -> [<IDcname, [], <<[], []>, INHER*, IDcreation*, FEAT*>>|FULLCOMP*]

[r2001] comp2fcomp([]) -> []

[r2010] mix(TEMPPAT1, TEMPPAT2) -> TEMPPAT1',
        merge(TEMPPAT1', TEMPPAT*) -> FLAT
        ---------------------------------------
        merge(TEMPPAT1, [TEMPPAT2|TEMPPAT*]) -> FLAT

[r2011] fcomp2comp(FULLCOMP*, []) -> COMP*,
        testsel(PFEAT*),
        pfeats2tdecl(PFEAT*) -> TYPEDECL*,
        typepfeats(PFEAT*, [<'current', type('', 'currentpattern')>|TYPEDECL*]) -> PFEAT'*
        ------------------------------------------------------------------------------------------------------------
        merge(<IDpname, <INHER*, IDcreation*, FULLCOMP*, PFEAT*>>, []) -> <IDpname, <INHER*, IDcreation*, COMP*, PFEAT'*>>

[r2020] TEMPPAT1 = <IDpname, <INHER1*, IDcreation*, FULLCOMP1*, PFEAT1*>>,
        mixcomp(FULLCOMP1*, FULLCOMP2*, []) -> FULLCOMP3*,
        mixpfeat(PFEAT1*, PFEAT2*) -> PFEAT3*
        ------------------------------------------------------------------------------------------------------------------
        mix(TEMPPAT1, <_, <INHER2*, _, FULLCOMP2*, PFEAT2*>>) -> <IDpname, <INHER1* ++ INHER2*, IDcreation*, FULLCOMP3*, PFEAT3*>>
```

```
% Dokumentation der Regeln [r2030] bis [r2062] siehe Unterkapitel 4.3.7 Die Zusammensetzung der Komponenten

[r2030] FULLCOMP2* = FULLCOMP2'* ++ [<@IDcname, FULLFEAT2*, <_, INHER2*, IDcreation2*, FEAT2*>>] ++ FULLCOMP2''*,
        mixfeat(FULLFEAT1*, FEAT1*, FULLFEAT2*, FEAT2*) -> <FULLFEAT3*, FEAT3*>,
        composeids(IDcreation1*, IDcreation2*) -> IDcreation3*,
        [<IDcname, FULLFEAT3*, <<[], []>, INHER1* ++ INHER2*, IDcreation3*, FEAT3*>>|FULLCOMP4*] = FULLCOMP5*,
        mixcomp(FULLCOMP1* ++ FULLCOMP1'*, FULLCOMP2'* ++ FULLCOMP2''*, FULLCOMP5*) -> FULLCOMP3*
        ------------------------------------------------------------------------------------------------------------------------------
        mixcomp(FULLCOMP1* ++ [<IDcname, FULLFEAT1*, <_, INHER1*, IDcreation1*, FEAT1*>>] ++ FULLCOMP1'*, FULLCOMP2*, FULLCOMP4*) -> FULLCOMP3*

[r2031] addfeat(FULLFEAT2*, FEAT*) -> <FULLFEAT3*, FEAT'*>,
        mixcomp([], FULLCOMP2*, [<IDcname, FULLFEAT3*, <<[], []>, INHER*, IDcreation*, FEAT'*>>|FULLCOMP4*]) -> FULLCOMP3*
        ------------------------------------------------------------------------------------------------------
        mixcomp([], [<IDcname, FULLFEAT2*, <_, INHER*, IDcreation*, FEAT*>>|FULLCOMP2*], FULLCOMP4*) -> FULLCOMP3*

[r2032] mixcomp([], [], FULLCOMP4*) -> FULLCOMP4*

[r2033] mixcomp([], FULLCOMP2*, FULLCOMP1* ++  FULLCOMP4*) -> FULLCOMP3*
        ------------------------------------------------------------
        mixcomp(FULLCOMP1*, FULLCOMP2*, FULLCOMP4*) -> FULLCOMP3*

[r2040] composeids(ID1* ++ ID1'*, ID2* ++ ID2'*) -> ID3*
        ------------------------------------------------------------------
        composeids(ID1* ++ [ID] ++ ID1'*, ID2* ++ [@ID] ++ ID2'*) -> [ID|ID3*]

[r2041] composeids(ID1*, ID2*) -> ID1* ++ ID2*

% Nicht selektiertes neues Feature mit existierendem 1 x unselektiertem Feature: Markierung 2 x unselektiert
[r2050] FULLFEAT1* = FULLFEAT1'* ++ [<@IDfname, unsel>] ++ FULLFEAT1''*,
        FULLFEAT1'* ++ [<IDfname, unsel2>] ++ FULLFEAT1''* = FULLFEAT4*,
        mixfeat(FULLFEAT4*, FEAT1*, FULLFEAT2* ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>
        ------------------------------------------------------------------------------------------------------
        mixfeat(FULLFEAT1*, FEAT1*, FULLFEAT2* ++ [<IDfname, unsel>] ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>

% Nicht selektiertes neues Feature mit existierendem selektiertem oder 2 x unselektiertem Feature: wird weggelassen
[r2051] FULLFEAT1* = _ ++ [<@IDfname, _>] ++ _,
        mixfeat(FULLFEAT1*, FEAT1*, FULLFEAT2* ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>
        ------------------------------------------------------------------------------------
        mixfeat(FULLFEAT1*, FEAT1*, FULLFEAT2* ++ [<IDfname, unsel>] ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>

% 2 x selektiertes Feature: nur Fehlerausgabe
[r2052] FULLFEAT1* = _ ++ [<@IDfname, sel>] ++ _,
        &debug twoselectsc(IDfname),
        mixfeat(FULLFEAT1*, FEAT1*, FULLFEAT2* ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>
        ------------------------------------------------------------------------------
        mixfeat(FULLFEAT1*, FEAT1*, FULLFEAT2* ++ [<IDfname, sel>] ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>
```

```
% selektiertes neues Feature überschreibt unselektiertes neues Feature
[r2053] FULLFEAT1* = FULLFEAT1'* ++ [<@IDfname, _>] ++ FULLFEAT1''*,
        FULLFEAT1'* ++ [<IDfname, sel>] ++ FULLFEAT1''* = FULLFEAT4*,
        FEAT2* = _ ++ [<@IDfname, FEATBODY>] ++ _,
        FEAT1* = FEAT1'* ++ [<@IDfname, _>] ++ FEAT1''*,
        mixfeat(FULLFEAT4*, FEAT1'* ++ [<IDfname, FEATBODY>] ++ FEAT1''*, FULLFEAT2* ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>
        ------------------------------------------------------------------------------------------------------------------------
        mixfeat(FULLFEAT1*, FEAT1*, FULLFEAT2* ++ [<IDfname, sel>] ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>

% selektiertes neues Feature, aber vorhandenens neuimplementiertes Feature: wird weggelassen
[r2054] FEAT1* = _ ++ [<@IDfname, _>] ++ _,
        mixfeat(FULLFEAT1*, FEAT1*, FULLFEAT2* ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>
        ------------------------------------------------------------------------------
        mixfeat(FULLFEAT1*, FEAT1*, FULLFEAT2* ++ [<IDfname, sel>] ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>

% selektiertes oder unselektiertes neues Feature ohne Neuimplementation: wird hinzugefügt
[r2055] FEAT2* = _ ++ [<@IDfname, FEATBODY>] ++ _,
        mixfeat([<IDfname, SEL>|FULLFEAT1*], [<IDfname, FEATBODY>|FEAT1*], FULLFEAT2* ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>
        ------------------------------------------------------------------------------------------------------------------------
        mixfeat(FULLFEAT1*, FEAT1*, FULLFEAT2* ++ [<IDfname, SEL>] ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>

% selektiertes geerbtes Feature überschreibt unselektiertes neues Feature
[r2056] FULLFEAT1* = FULLFEAT1'* ++ [<@IDfname, _>] ++ FULLFEAT1''*,
        FULLFEAT1'* ++ [<IDfname, sel>] ++ FULLFEAT1''* = FULLFEAT4*,
        FEAT1* = FEAT1'* ++ [<@IDfname, _>] ++ FEAT1''*,
        mixfeat(FULLFEAT4*, FEAT1'* ++ FEAT1''*, FULLFEAT2* ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>
        ------------------------------------------------------------------------------------------------------
        mixfeat(FULLFEAT1*, FEAT1*, FULLFEAT2* ++ [<IDfname, sel>] ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>

% selektiertes geerbtes Feature: wird nur in neuer FULLFEAT- Liste hinzugefügt
[r2057] mixfeat([<IDfname, sel>|FULLFEAT1*], FEAT1*, FULLFEAT2* ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>
        ------------------------------------------------------------------------------------------------
        mixfeat(FULLFEAT1*, FEAT1*, FULLFEAT2* ++ [<IDfname, sel>] ++ FULLFEAT2'*, FEAT2*) -> <FULLFEAT3*, FEAT3*>

% Keine neu verfeinerten Features
[r2058] &debug writeFFs(FULLFEAT1*), &debug writeFEATs(FEAT1*),
        ----------------------
        mixfeat(FULLFEAT1*, FEAT1*, [], _) -> <FULLFEAT1*, FEAT1*>

[r2060] FEAT* = FEAT'* ++ [<@IDfname, FEATBODY>] ++ FEAT''*,
        addfeat(FULLFEAT2* ++ FULLFEAT2'*, FEAT'* ++ FEAT''*) -> <FULLFEAT3*, FEAT2*>
        ------------------------------------------------------------------------------------------------------------------------
        addfeat(FULLFEAT2* ++ [<IDfname, SEL>] ++ FULLFEAT2'*, FEAT*) -> <[<IDfname, SEL>|FULLFEAT3*], [<IDfname, FEATBODY>|FEAT2*]>

[r2061] addfeat(FULLFEAT2* ++ FULLFEAT2'*, FEAT*) -> <FULLFEAT3*, FEAT2*>
        ------------------------------------------------------------------------------------------------
        addfeat(FULLFEAT2* ++ [<IDfname, sel>] ++ FULLFEAT2'*, FEAT*) -> <[<IDfname, sel>|FULLFEAT3*], FEAT2*>

[r2062] addfeat(_, _) -> <[], []>
```

```
% Dokumentation der Regeln [r2070] bis [r2151] siehe Unterkapitel 4.3.8 Zuordnung der Typen zu den Feature- Calls

[r2070] type1feat(FEAT, TYPEDECL*) -> FEAT',
        typepfeats(PFEAT*, TYPEDECL*) -> PFEAT'*
        ------------------------------------------------------------------------------
        typepfeats([<EXTINT, SEL, FEAT>|PFEAT*], TYPEDECL*) ->  [<EXTINT, SEL, FEAT'>|PFEAT'*]

[r2071] typepfeats([], _) ->  []

[r2072] type1feat(FEAT, TYPEDECL*) -> FEAT',
        typefeats(FEAT*, TYPEDECL*) -> FEAT'*
        ------------------------------------------------
        typefeats([FEAT|FEAT*], TYPEDECL*) -> [FEAT'|FEAT'*]

[r2073] typefeats([], _) -> []

[r2074] type1feat(<ID, attribute(TYPE)>, _) -> <ID, attribute(TYPE)>

[r2075] FEAT = <IDfname, procedure(TYPEDECL1*, <TYPEDECL2*, <COMPOUND, CRITFEAT*>>)>,
        vari2type(CRITFEAT*, TYPEDECL1* ++ TYPEDECL2* ++ TYPEDECL*) -> CRITFEAT'*
        ---------------------------------------------------------------------------------------------
        type1feat(FEAT, TYPEDECL*) -> <IDfname, procedure(TYPEDECL1*, <TYPEDECL2*, <COMPOUND, CRITFEAT'*>>)>

[r2076] FEAT = <IDfname, function(TYPE, TYPEDECL1*, <TYPEDECL2*, <COMPOUND, CRITFEAT*>>)>,
        vari2type(CRITFEAT*, TYPEDECL1* ++ TYPEDECL2* ++ TYPEDECL* ++ [<'result', TYPE>]) -> CRITFEAT'*
        ---------------------------------------------------------------------------------------------------
        type1feat(FEAT, TYPEDECL*) -> <IDfname, function(TYPE, TYPEDECL1*, <TYPEDECL2*, <COMPOUND, CRITFEAT'*>>)>

[r2080] TYPEDECL* = TYPEDECL'* ++ [<@IDvariable, type(IDprefix, IDtype)>] ++ TYPEDECL''*,
        vari2type(CRITFEAT*, TYPEDECL'* ++ [<IDvariable, type(IDprefix, IDtype)>] ++ TYPEDECL''*) -> CRITFEAT'*
        ---------------------------------------------------------------------------------------------
        vari2type([<IDvariable, _, IDfname>|CRITFEAT*], TYPEDECL*) -> [<IDvariable, IDtype, IDfname>|CRITFEAT'*]

[r2081] vari2type(CRITFEAT*, TYPEDECL*) -> CRITFEAT'*
        ----------------------------------------
        vari2type([CRITFEAT|CRITFEAT*], TYPEDECL*) -> [CRITFEAT|CRITFEAT'*]

[r2082] vari2type([], _) -> []

[r2090] FULLCOMP* = FULLCOMP'* ++ [<IDcname, FULLFEAT*, <CAST, INHER*, IDcreation*, FEAT*>>] ++ FULLCOMP''*,
        inherftypes(INHER*, FEATTYPES*, []) -> TYPEDECL1*,
        feats2tdecl(FEAT*, TYPEDECL1*) -> TYPEDECL2*,
        typefeats(FEAT*, [<'current', type('', 'currentcomponent')>|TYPEDECL2*]) -> FEAT'*,
        testunsel(IDcname, FULLFEAT*),
        fcomp2comp(FULLCOMP'* ++ FULLCOMP''*, [<IDcname, TYPEDECL2*>|FEATTYPES*]) -> COMP*
        ---------------------------------------------------------------------------------------------
        fcomp2comp(FULLCOMP*, FEATTYPES*) -> [<IDcname, <CAST, INHER*, IDcreation*, FEAT'*>>|COMP*]

[r2091] fcomp2comp([], _) -> []
```

```
[r2100] &debug noselectc(<IDcname, IDfname>),
        testunsel(IDcname, FULLFEAT* ++ FULLFEAT'*)
        ------------------------------------------------------------------
        testunsel(IDcname, FULLFEAT* ++ [<IDfname, unsel2>] ++ FULLFEAT'*)

[r2101] testunsel(_, _)

[r2110] &debug noselectp(IDfname),
        testsel(PFEAT1* ++ PFEAT2*)
        ---------------------------------------------------------
        testsel(PFEAT1* ++ [<_, unsel2, <IDfname, _>>] ++ PFEAT2*)

[r2111] testsel(_)

[r2120] FEATTYPES* = _ ++ [<@IDcname, TYPEDECL2*>] ++ _,
        mixtypedecl(TYPEDECL1*, TYPEDECL2*) -> TYPEDECL3*,
        inherftypes(INHER*, FEATTYPES*, TYPEDECL3*) -> TYPEDECL4*
        -------------------------------------------------------
        inherftypes([<IDcname, _>|INHER*], FEATTYPES*, TYPEDECL1*) -> TYPEDECL4*

[r2121] inherftypes([], _, TYPEDECL*) -> TYPEDECL*

[r2130] TYPEDECL2* = TYPEDECL2'* ++ [<@IDfname, type(_, @IDtype)>] ++ TYPEDECL2''*,
        mixtypedecl(TYPEDECL1* ++ TYPEDECL1'*, TYPEDECL2'* ++ TYPEDECL2''*) -> TYPEDECL3*
        ------------------------------------------------------------------------------------------------------------------
        mixtypedecl(TYPEDECL1* ++ [<IDfname, type(_, IDtype)>] ++ TYPEDECL1'*, TYPEDECL2*) -> [<IDfname, type('', IDtype)>|TYPEDECL3*]

[r2131] TYPEDECL2* = TYPEDECL2'* ++ [<@IDfname, type(_, _)>] ++ TYPEDECL2''*,
        mixtypedecl(TYPEDECL1* ++ TYPEDECL1'*, TYPEDECL2'* ++ TYPEDECL2''*) -> TYPEDECL3*
        ------------------------------------------------------------------------------------------------------------------
        mixtypedecl(TYPEDECL1* ++ [<IDfname, type(_, IDtype)>] ++ TYPEDECL1'*, TYPEDECL2*) -> [<IDfname, type('warning', IDtype)>|TYPEDECL3*]

[r2132] mixtypedecl(TYPEDECL1*, TYPEDECL2*) -> TYPEDECL1* ++ TYPEDECL2*

[r2140] feat21tdecl(FEATBODY) -> TYPE,
        feats2tdecl(FEAT*, TYPEDECL1* ++ TYPEDECL1'*) -> TYPEDECL2*
        ------------------------------------------------------------------------------------------------------
        feats2tdecl([<IDfname, FEATBODY>|FEAT*], TYPEDECL1* ++ [<@IDfname, _>]  ++ TYPEDECL1'*) -> [<IDfname, TYPE>|TYPEDECL2*]

[r2141] feat21tdecl(FEATBODY) -> TYPE,
        feats2tdecl(FEAT*, TYPEDECL1*) -> TYPEDECL2*
        ---------------------------------------------------------------------------
        feats2tdecl([<IDfname, FEATBODY>|FEAT*], TYPEDECL1*) -> [<IDfname, TYPE>|TYPEDECL2*]

[r2142] feats2tdecl(FEAT*, TYPEDECL1*) -> TYPEDECL2*
        ----------------------------------------------
        feats2tdecl([_|FEAT*], TYPEDECL1*) -> TYPEDECL2*
```

```
[r2143] &debug difftypes(<IDfname, IDtype>),
        feats2tdecl([], TYPEDECL1* ++ TYPEDECL1’*) -> TYPEDECL2*
        ------------------------------------------------------------------------------------------------------------------------
        feats2tdecl([], TYPEDECL1* ++ [<IDfname, type(’warning’, IDtype)>] ++ TYPEDECL1’*) -> [<IDfname, type(’’, IDtype)>|TYPEDECL2*]

[r2144] feats2tdecl([], TYPEDECL1*) -> TYPEDECL1*

[r2145] feat21tdecl(attribute(TYPE)) -> TYPE

[r2146] feat21tdecl(function(TYPE, _, _)) -> TYPE

[r2150] feat21tdecl(FEATBODY) -> TYPE,
        pfeats2tdecl(PFEAT* ++ PFEAT’*) -> TYPEDECL*
        ------------------------------------------------------------------------------------
        pfeats2tdecl(PFEAT* ++ [<_, _, <IDfname, FEATBODY>>] ++ PFEAT’*) -> [<IDfname, TYPE>|TYPEDECL*]

[r2151] pfeats2tdecl(_) -> []

% Dokumentation der Regeln [r2160] bis [r2171] siehe Unterkapitel 4.3.6 Das Zusammensetzen des Patterns

% 2 x selektiertes Patternfeature: nur Fehlerausgabe
[r2160] PFEAT2* = PFEAT2’* ++ [<EXTINT2, sel, <@IDfname, _>>] ++ PFEAT2’’*,
        &debug twoselectsp(IDfname),
        extint(EXTINT1, EXTINT2) -> EXTINT3,
        mixpfeat(PFEAT1* ++ PFEAT1’*, PFEAT2’* ++ PFEAT2’’*) -> PFEAT3*
        ------------------------------------------------------------------------------------------------------------------------
        mixpfeat(PFEAT1* ++ [<EXTINT1, sel, <IDfname, FEATBODY>>] ++ PFEAT1’*, PFEAT2*) -> [<EXTINT3, sel, <IDfname, FEATBODY>>|PFEAT3*]

% unselektiertes Patternfeature wird wegen selektiertem Patternfeature aus anderer Verfeinerung ignoriert
[r2161] PFEAT2* = PFEAT2’* ++ [<EXTINT2, _, <@IDfname, _>>] ++ PFEAT2’’*,
        extint(EXTINT1, EXTINT2) -> EXTINT3,
        mixpfeat(PFEAT1* ++ PFEAT1’*, PFEAT2’* ++ PFEAT2’’*) -> PFEAT3*
        ------------------------------------------------------------------------------------------------------------------------
        mixpfeat(PFEAT1* ++ [<EXTINT1, sel, <IDfname, FEATBODY>>] ++ PFEAT1’*, PFEAT2*) -> [<EXTINT3, sel, <IDfname, FEATBODY>>|PFEAT3*]

% selektiertes Patternfeature überschreibt unselektiertes Patternfeature aus anderer Verfeinerung
[r2162] PFEAT2* = PFEAT2'* ++ [<EXTINT2, sel, <@IDfname, FEATBODY>>] ++ PFEAT2''*,
        extint(EXTINT1, EXTINT2) -> EXTINT3,
        mixpfeat(PFEAT1* ++ PFEAT1'*, PFEAT2'* ++ PFEAT2''*) -> PFEAT3*
        ------------------------------------------------------------------------------------------------------------------------
        mixpfeat(PFEAT1* ++ [<EXTINT1, unsel, <IDfname, _>>] ++ PFEAT1'*, PFEAT2*) -> [<EXTINT3, sel, <IDfname, FEATBODY>>|PFEAT3*]

% selektiertes Patternfeature überschreibt unselektiertes Patternfeature aus anderer Verfeinerung
[r2163] PFEAT2* = PFEAT2'* ++ [<EXTINT2, sel, <@IDfname, FEATBODY>>] ++ PFEAT2''*,
        extint(EXTINT1, EXTINT2) -> EXTINT3,
        mixpfeat(PFEAT1* ++ PFEAT1'*, PFEAT2'* ++ PFEAT2''*) -> PFEAT3*
        ------------------------------------------------------------------------------------------------------------------------
        mixpfeat(PFEAT1* ++ [<EXTINT1, unsel2, <IDfname, _>>] ++ PFEAT1'*, PFEAT2*) -> [<EXTINT3, sel, <IDfname, FEATBODY>>|PFEAT3*]
```

```
% Patternfeature wird wegen neuimplementiertem Patternfeature ignoriert
[r2164] PFEAT2* = PFEAT2'* ++ [<EXTINT2, _, <@IDfname, _>>] ++ PFEAT2''*,
        extint(EXTINT1, EXTINT2) -> EXTINT3,
        mixpfeat(PFEAT1* ++ PFEAT1'*, PFEAT2'* ++ PFEAT2''*) -> PFEAT3*
        ------------------------------------------------------------------------------------------------------------------------
        mixpfeat(PFEAT1* ++ [<EXTINT1, nosel, <IDfname, FEATBODY>>] ++ PFEAT1'*, PFEAT2*) -> [<EXTINT3, nosel, <IDfname, FEATBODY>>|PFEAT3*]

% 2. unselektiertes Patternfeature: wird als Fehlerkandidat mit unsel2 markiert
[r2165] PFEAT2* = PFEAT2'* ++ [<EXTINT2, unsel, <@IDfname, FEATBODY>>] ++ PFEAT2''*,
        extint(EXTINT1, EXTINT2) -> EXTINT3,
        mixpfeat(PFEAT1* ++ PFEAT1'*, PFEAT2'* ++ PFEAT2''*) -> PFEAT3*
        ------------------------------------------------------------------------------------------------------------------------
        mixpfeat(PFEAT1* ++ [<EXTINT1, _, <IDfname, _>>] ++ PFEAT1'*, PFEAT2*) -> [<EXTINT3, unsel2, <IDfname, FEATBODY>>|PFEAT3*]

% Disjunkte Paternfeature- Mengen werden addiert
[r2166] mixpfeat(PFEAT1*, PFEAT2*) -> PFEAT1* ++ PFEAT2*

[r2170] extint(intern, intern) -> intern

[r2171] extint(_, _) -> extern
```

A.6 Die zweite Transformation mit classes.ir

```
% Datei:         classes.ir
% Dokumentation: Unterkapitel 4.4 Die zweite Transformation

Axiom Is main

main: FLAT* -> CLASS*

% Alle flachen Pattern werden in Eiffelklassen umgewandelt und um die redefines ergaenzt.

[r0000] list(FLAT*) -> CLASS1*,
        redefines(CLASS1*, []) -> CLASS2*
        ---------------------------------
        main(FLAT*) -> CLASS2*

% Transformation der Patternstrukturen in Klassenstrukturen und Anpassung der Typen

[r0010] flat2classes(FLAT) -> <CLASS1, CLASS1*>,
        expandtypes(CLASS1*, CLASS1*) -> CLASS2*,
        list(FLAT*) -> CLASS3*
        ------------------------------------------------
        list([FLAT|FLAT*]) -> [CLASS1|CLASS2*] ++ CLASS3*

[r0011] list([]) -> []

% Dokumentation der Regeln [r1000] bis [r1132] siehe Unterkapitel 4.4.1 Die Abbildung der externen Schnittstelle des Patterns
```

```
[r1000] flat2extclass(FLAT) -> CLASS1,
        flat2intclass(FLAT) -> CLASS2,
        FLAT = <IDpname, <_, _, COMP*, _>>,
        comps2classes(IDpname, COMP*) -> CLASS*
        ------------------------------------------
        flat2classes(FLAT) -> <CLASS1, [CLASS2|CLASS*]>

[r1100] inh2clinh('', INHER*) -> CLINHER*,
        pfeat2deffeat(PFEAT*) -> FEAT*
        ------------------------------------------------------------------------------
        flat2extclass(<IDfname, <INHER*, _, _, PFEAT*>>) -> < '', IDfname, <CLINHER*, [], FEAT*>>

[r1110] inh2clinh(IDprefix, INHER*) -> CLINHER*
        ---------------------------------------------------------------------------------------
        inh2clinh(IDprefix, [<IDcname, IDcname*>|INHER*]) -> [<IDprefix, IDcname, IDcname*>|CLINHER*]

[r1111] inh2clinh(_, []) -> []

[r1120] feat2deffeat(FEATBODY) -> FEATBODY',
        pfeat2deffeat(PFEAT*) -> FEAT*
        ------------------------------------------------------------------------------
        pfeat2deffeat([<extern, _, <IDfname, FEATBODY>>|PFEAT*]) -> [<IDfname, FEATBODY'>|FEAT*]

[r1121] pfeat2deffeat(PFEAT*) -> FEAT*
        ------------------------------------------
        pfeat2deffeat([<intern, _, _>|PFEAT*]) -> FEAT*

[r1122] pfeat2deffeat([]) -> []

[r1130] feat2deffeat(attribute(TYPE)) -> attribute(TYPE)

[r1131] feat2deffeat(procedure(TYPEDECL1*, <_, <_, _>>)) -> procedure(TYPEDECL1*, <[], <deferred, []>>)

[r1132] feat2deffeat(function(TYPE, TYPEDECL1*, <_, <_, _>>)) -> function(TYPE, TYPEDECL1*, <[], <deferred, []>>)

% Dokumentation der Regeln [r1200] bis [r1211] siehe Unterkapitel 4.4.2 Die Abbildung des vollständigen Patterns

[r1200] pfeat2feat(PFEAT*) -> FEAT*
        ------------------------------------------------------------------------------------------------------------
        flat2intclass(<IDpname, <_, IDcreation*, _, PFEAT*>>) -> <IDpname, IDpname, <[<'', IDpname, []>], IDcreation*, FEAT*>>

[r1210] pfeat2feat(PFEAT*) -> FEAT*
        ------------------------------------------------
        pfeat2feat([<_, _, FEAT>|PFEAT*]) -> [FEAT|FEAT*]

[r1211] pfeat2feat([]) -> []

% Dokumentation der Regeln [r1300] bis [r1310] siehe Unterkapitel 4.4.3 Die Abbildung der Komponenten
```

```
[r1300] comp2class(IDpname, COMP) -> CLASS,
        comps2classes(IDpname, COMP*) -> CLASS*
        ------------------------------------------------------
        comps2classes(IDpname, [COMP|COMP*]) -> [CLASS|CLASS*]

[r1301] comps2classes(_, []) -> []

[r1310] inh2clinh(IDpname, INHER*) -> CLINHER*
        ------------------------------------------------------------------------------------------------------------
        comp2class(IDpname, <IDcname, <_, INHER*, IDcreation*, FEAT*>>) -> <IDpname, IDcname, <CLINHER*, IDcreation*, FEAT*>>

% Dokumentation der Regeln [r2000] bis [r2062] siehe Unterkapitel 4.4.4 Die Anpassung der Typen an die Klassennamen

[r2000] CLASS* = [<IDprefix, IDcname, <CLINHER*, IDcreation*, FEAT*>>|CLASS1*],
        expfeatures(FEAT*, CLASS2*) -> FEAT'*,
        expandtypes(CLASS1*, CLASS2*) -> CLASS3*
        ------------------------------------------------------------------------------------
        expandtypes(CLASS*, CLASS2*) -> [<IDprefix, IDcname, <CLINHER*, IDcreation*, FEAT'*>>|CLASS3*]

[r2001] expandtypes([], _) -> []

[r2010] expfeatbody(FEATBODY, CLASS*) -> FEATBODY',
        expfeatures(FEAT*, CLASS*) -> FEAT'*
        ------------------------------------------------------------------------
        expfeatures([<IDfname, FEATBODY>|FEAT*], CLASS*) -> [<IDfname, FEATBODY'>|FEAT'*]

[r2011] expfeatures([], _) -> []

[r2020] exp1type(TYPE, CLASS*) -> TYPE'
        ------------------------------------------------------
        expfeatbody(attribute(TYPE), CLASS*) -> attribute(TYPE')

[r2021] FEATBODY = procedure(TYPEDECL1*, <TYPEDECL2*, <COMPOUND, CRITFEAT*>>),
        exptypedecl(TYPEDECL1*, CLASS*) -> TYPEDECL1'*,
        exptypedecl(TYPEDECL2*, CLASS*) -> TYPEDECL2'*,
        expcritfeats(CRITFEAT*, CLASS*) -> CRITFEAT'*
        ------------------------------------------------------------------------------------
        expfeatbody(FEATBODY, CLASS*) -> procedure(TYPEDECL1'*, <TYPEDECL2'*, <COMPOUND, CRITFEAT'*>>)

[r2022] FEATBODY = function(TYPE, TYPEDECL1*, <TYPEDECL2*, <COMPOUND, CRITFEAT*>>),
        exp1type(TYPE, CLASS*) -> TYPE',
        exptypedecl(TYPEDECL1*, CLASS*) -> TYPEDECL1'*,
        exptypedecl(TYPEDECL2*, CLASS*) -> TYPEDECL2'*,
        expcritfeats(CRITFEAT*, CLASS*) -> CRITFEAT'*
        ------------------------------------------------------------------------------------------
        expfeatbody(FEATBODY, CLASS*) -> function(TYPE', TYPEDECL1'*, <TYPEDECL2'*, <COMPOUND, CRITFEAT'*>>)

[r2030] exp1type(TYPE, CLASS*) -> TYPE',
        exptypedecl(TYPEDECL*, CLASS*) -> TYPEDECL'*
        ------------------------------------------------------------------------------
        exptypedecl([<IDvariable, TYPE>|TYPEDECL*], CLASS*) -> [<IDvariable, TYPE'>|TYPEDECL'*]
```

```
[r2031] exptypedecl([], _) -> []

[r2040] exp1type(type(_, 'currentpattern'), _ ++ [<IDpname, @IDpname, _>] ++ _) -> type(IDpname, IDpname)

[r2041] exp1type(type(_, IDcname), _ ++ [<IDprefix, @IDcname, _>] ++ _) -> type(IDprefix, IDcname)

[r2042] exp1type(type(_, IDtype), _) -> type('', IDtype)

[r2050] exp1critfeat(CRITFEAT, CLASS*) -> CRITFEAT',
        expcritfeats(CRITFEAT*, CLASS*) -> CRITFEAT'*
        ------------------------------------------------------------------
        expcritfeats([CRITFEAT|CRITFEAT*], CLASS*) -> [CRITFEAT'|CRITFEAT'*]

[r2051] expcritfeats([], _) -> []

[r2060] exp1critfeat(<_, 'currentpattern', IDfname>, _ ++ [<IDpname, @IDpname, _>] ++ _) -> <IDpname, IDpname, IDfname>

[r2061] exp1critfeat(<_, IDcname, IDfname>, _ ++ [<IDprefix, @IDcname, _>] ++ _) -> <IDprefix, IDcname, IDfname>

[r2062] exp1critfeat(<_, IDpname, IDfname>, _) -> <IDpname, IDpname, IDfname>

% Dokumentation der Regeln [r3000] bis [r2041] siehe Unterkapitel 4.4.5 Die Berechnung der Redefines

[r3000] CLASS* = CLASS1* ++ [<IDprefix, IDcname, <CLINHER*, IDcreation*, FEAT*>>] ++ CLASS1'*,
        mininher(CLINHER*) -> CLINHER'*,
        testinher(CLINHER*, FEAT*, CLASSFEATS*) -> <CLINHER''*, IDfname*>,
        redefines(CLASS1* ++ CLASS1'*, [<IDprefix, IDcname, IDfname*>|CLASSFEATS*]) -> CLASS2*
        ------------------------------------------------------------------------------------
        redefines(CLASS*, CLASSFEATS*) -> [<IDprefix, IDcname, <CLINHER''*, IDcreation*, FEAT*>>|CLASS2*]

[r3001] redefines([], _) -> []

[r3010] CLASSFEATS* = _ ++ [<@IDprefix, @IDcname, IDfname1*>] ++ _,
        comparefeats(FEAT*, IDfname1*) -> IDfname2*,
        testinher(CLINHER*, FEAT*, CLASSFEATS*) -> <CLINHER'*, IDfname3*>,
        composeids(IDfname1*, IDfname3*) -> IDfname4*
        ------------------------------------------------------------------------------------------------------------------
        testinher([<IDprefix, IDcname, _>|CLINHER*], FEAT*, CLASSFEATS*) -> <[<IDprefix, IDcname, IDfname2*>|CLINHER'*], IDfname4*>

[r3011] idoffeats(FEAT*) -> IDfname*
        ---------------------------------------
        testinher([], FEAT*, _) -> <[], IDfname*>

[r3020] idoffeats(FEAT*) -> IDfname*
        ------------------------------------------------
        idoffeats([<IDfname, _>|FEAT*]) -> [IDfname|IDfname*]

[r3021] idoffeats([]) -> []
```

```
[r3030] IDfname1* = _ ++ [@IDfname] ++ _,
        comparefeats(FEAT*, IDfname1*) -> IDfname2*
        ------------------------------------------------------------------
        comparefeats([<IDfname, _>|FEAT*], IDfname1*) -> [IDfname|IDfname2*]

[r3031] comparefeats(FEAT*, IDfname1*) -> IDfname2*
        ---------------------------------------------------
        comparefeats([<_, _>|FEAT*], IDfname1*) -> IDfname2*

[r3032] comparefeats([], _) -> []

[r3040] composeids(ID1* ++ ID1'*, ID2* ++ ID2'*) -> ID3*
        ------------------------------------------------------------------
        composeids(ID1* ++ [ID] ++ ID1'*, ID2* ++ [@ID] ++ ID2'*) -> [ID|ID3*]

[r3041] composeids(ID1*, ID2*) -> ID1* ++ ID2*

[r3050] mininher(CLINHER1* ++ <IDprefix, IDcname, _> ++ CLINHER2* ++ CLINHER3*) -> CLINHER4*
        ------------------------------------------------------------------------------------------------------------
        mininher(CLINHER1* ++ <IDprefix, IDcname, _> ++ CLINHER2* ++ <@IDprefix, @IDcname, _> ++ CLINHER3*) -> CLINHER4*

[r3051] mininher(CLINHER*) -> CLINHER*
```

A.7 Die Steuerung der Generierung mit gen.ir

```
% Datei:         gen.ir
% Dokumentation: Unterkapitel 4.5.1 Die Steuerung der Quelltextgenerierung

Axiom Is main

main: CLASS*

[r0000] writeclasses(CLASS*),
        &writeclass writesummary(CLASS*)
        -------------------------------
        main(CLASS*)

[r0010] names2file(IDprefix, IDcname) -> IDfile,
        &writeclass writeeclass(<IDfile, <IDprefix, IDcname, CLASSBODY>>),
        writeclasses(CLASS*)
        ---------------------------------------------------------------
        writeclasses([<IDprefix, IDcname, CLASSBODY>|CLASS*])

[r0011] writeclasses([])
```

A.8 Die Steuerung der Generierung mit writecl.pra

```
% Datei:         writecl.pra
% Dokumentation: Unterkapitel 4.5.1 Die Steuerung der Quelltextgenerierung

generate : Refine ./eiffel By ./token
           And Refine ./classname By lib/conv.

Procedure writeeclass With <ID, CLASS>
Begin

 Writing ./example/ID.e Do
  Run generate([CLASS])
 End Writing

End Procedure.

Procedure writesummary With CLASS*
Begin

 Writing ./example/summary.eiffel Do
  Run generate(CLASS*)
 End Writing

End Procedure.
```

A.9 Die Generierung von Klassennamen mit classname.ir

```
% Datei:         classname.ir
% Dokumentation: Unterkapitel 4.5.2 Die Erzeugung der Klassennamen

[r0000] concatnames(IDprefix, IDcname) -> IDconcat,
        identifier2chars(IDconcat) -> Char1*,
        chars2identifierLC(Char1*) -> IDfile
        ------------------------------------------
        names2file(IDprefix, IDcname) -> IDfile

[r0010] concatnames('', IDcname) -> IDcname

[r0011] identifier2chars(IDprefix) -> Char1*,
        identifier2chars(IDcname) -> Char2*,
        identifier2chars('_') -> Char3*,
        chars2identifier(Char1* ++ Char2* ++ Char3*) -> IDconcat
        ------------------------------------------------------
        concatnames(IDprefix, IDcname) -> IDconcat
```

```
[r0020] identifier2chars(IDcname) -> Char1*,
        identifier2chars('_') -> Char2*,
        chars2identifier(Char1* ++ Char1* ++ Char2*) -> IDconcat
        ------------------------------------------------------
        concatcreation('', IDcname) -> IDconcat

[r0021] identifier2chars(IDprefix) -> Char1*,
        identifier2chars(IDcname) -> Char2*,
        identifier2chars('_') -> Char3*,
        chars2identifier(Char1* ++ Char2* ++ Char3*) -> IDconcat
        ------------------------------------------------------
        concatcreation(IDprefix, IDcname) -> Idconcat
```

A.10 Das Schreiben von Eiffel-Quelltext mit eiffel.gs

```
% Datei:         eiffel.gs
% Dokumentation: Unterkapitel 4.5.3 Das Schreiben des Eiffel-Quelltextes

Axiom Is main

%Include ./aeiffel

[r0000] main([]) : .

[r0001] main([CLASS|CLASS*])
         :
           class(CLASS),
           main(CLASS*).

% Schreiben einer Klasse

[r0010] class(<IDprefix, IDcname, <CLINHER*, IDcreation*, FEAT*>>)
         :
           "class ",
           concatnames(IDprefix, IDcname) -> IDconcat,
           id(IDconcat),
           nl,
           inherits1(CLINHER*),
           create(IDcreation*),
           feature(FEAT*),
           "end",
           nl, nl.

[r0020] inherits1([CLINHER|CLINHER*])
         :
           "  inherit",
           inherit(CLINHER),
           inherits2(CLINHER*).

[r0021] inherits1([]) : .

[r0022] inherits2([CLINHER|CLINHER*])
         :
           ";",
           inherit(CLINHER),
           inherits2(CLINHER*).

[r0023] inherits2([])
         :
           nl.

[r0024] inherit(<IDprefix, IDcname, IDfname*>)
         :
           nl, "    ",
           concatnames(IDprefix, IDcname) -> IDconcat,
           id(IDconcat),
           nl,
           redefine(IDfname*),
           "    end".

[r0030] idlist([ID|ID*])
         :
           ", ",
           id(ID),
           idlist(ID*).
```

```
[r0031] idlist([]) : .

[r0040] redefine([IDfname|IDfname*])
         :
           "      redefine ",
           id(IDfname),
           idlist(IDfname*),
           nl.

[r0041] redefine([]) : .

[r0050] create([IDcreation|IDcreation*])
         :
           "  creation ",
           id(IDcreation),
           idlist(IDcreation*),
           nl.

[r0051] create([]) : .

% Schreiben eines Features

[r0200] feature([<IDfname, FEATBODY>|FEAT*])
         :
           "  feature ",
           id(IDfname),
           featurebody(FEATBODY),
           nl,
           feature(FEAT*).

[r0201] feature([]) : .

[r0220] featurebody(attribute(TYPE))
         :
           ": ",
           type(TYPE).

[r0221] featurebody(procedure(TYPEDECL*, IMPLEMENT))
         :
           typedecls(TYPEDECL*),
           implement(IMPLEMENT),
           "    end".

[r0222] featurebody(function(TYPE, TYPEDECL*, IMPLEMENT))
         :
           typedecls(TYPEDECL*),
           ": ",
           type(TYPE),
           implement(IMPLEMENT),
           "    end".

[r0230] typedecls([TYPEDECL|TYPEDECL*])
         :
           "(",
           typedecl(TYPEDECL),
           typedeclssemi(TYPEDECL*),
           ")".

[r0231] typedecls([]) : .

[r0232] typedeclssemi([TYPEDECL|TYPEDECL*])
         :
           "; ",
           typedecl(TYPEDECL),
           typedeclssemi(TYPEDECL*).

[r0233] typedeclssemi([]) : .

[r0234] typedeclsnl([TYPEDECL|TYPEDECL*])
         :
           ";", nl,
           "      ",
           typedecl(TYPEDECL),
           typedeclsnl(TYPEDECL*).

[r0235] typedeclsnl([]) : .
```

```
[r0236] typedecl(<IDvariable, TYPE>)
         :
           id(IDvariable),
           ": ",
           type(TYPE).

[r0240] type(type(_, 'currentpattern'))
         :
           "like current".

[r0241] type(type(IDprefix, IDtype))
         :
           concatnames(IDprefix, IDtype) -> IDconcat,
           id(IDconcat).

[r0250] implement(<TYPEDECL*, SOURCE>)
         :
           " is",
           locals(TYPEDECL*),
           nl,
           source(SOURCE),
           nl.

[r0260] locals([TYPEDECL|TYPEDECL*])
         :
           nl,
           "    local",
           nl, "      ",
           typedecl(TYPEDECL),
           typedeclsnl(TYPEDECL*).

[r0261] locals([]) : .

% Schreiben der Implementation eines Features

[r0300] source(<deferred, []>)
         :
           "    deferred".

[r0301] source(<instructions(INSTRUCTION*), CRITFEAT*>)
         :
           "    do",
           compound(INSTRUCTION*, CRITFEAT*) -> [].

[r0310] compound([INSTRUCTION|INSTRUCTION*], CRITFEAT*) -> CRITFEAT''*
         :
           nl, "      ",
           instr(INSTRUCTION, CRITFEAT*) -> CRITFEAT'*,
           instrs(INSTRUCTION*, CRITFEAT'*) -> CRITFEAT''*.

[r0311] compound([], CRITFEAT*) -> CRITFEAT* : .

[r0320] instrs([INSTRUCTION|INSTRUCTION*], CRITFEAT*) -> CRITFEAT''*
         :
           ";", nl, "      ",
           instr(INSTRUCTION, CRITFEAT*) -> CRITFEAT'*,
           instrs(INSTRUCTION*, CRITFEAT'*) -> CRITFEAT''*.

[r0321] instrs([], CRITFEAT*) -> CRITFEAT* : .

[r0322] instr(creation(LOCALCALL), [<_, _, IDfname1>] ++ [<IDprefix, IDtype, IDfname2>|CRITFEAT*])
         -> CRITFEAT'*
         :
           "!",
           concatcreation(IDprefix, IDtype) -> IDconcat,
           id(IDconcat),
           "!",
           id(IDfname1),
           ".",
           localcall(LOCALCALL, [<IDprefix, IDtype, IDfname2>|CRITFEAT*]) -> CRITFEAT'*.
```

```
[r0323] instr(if([<EXPRESSION,INSTRUCTION1*>|IFTHEN*], INSTRUCTION2*), CRITFEAT*) -> CRITFEAT4*
         :
           "if ",
           expression(EXPRESSION, CRITFEAT*) -> CRITFEAT1*,
           " then ",
           compound(INSTRUCTION1*, CRITFEAT1*) -> CRITFEAT2*,
           elseif(IFTHEN*, CRITFEAT2*) -> CRITFEAT3*,
           else(INSTRUCTION2*, CRITFEAT3*) -> CRITFEAT4*,
           nl,
           "      end".

[r0324] instr(loop(INSTRUCTION1*, EXPRESSION, INSTRUCTION2*), CRITFEAT*) -> CRITFEAT3*
         :
           "from",
           compound(INSTRUCTION1*, CRITFEAT*) -> CRITFEAT1*,
           nl,
           "      until ",
           expression(EXPRESSION, CRITFEAT1*) -> CRITFEAT2*,
           nl,
           "      loop",
           compound(INSTRUCTION2*, CRITFEAT2*) -> CRITFEAT3*,
           nl,
           "      end".

[r0325] instr(instrcall(CALL), CRITFEAT*) -> CRITFEAT'*
         :
           call(CALL, CRITFEAT*) -> CRITFEAT'*.

[r0326] instr(let(LOCALCALL, EXPRESSION), CRITFEAT*) -> CRITFEAT''*
         :
           localcall(LOCALCALL, CRITFEAT*) -> CRITFEAT'*,
           " := ",
           expression(EXPRESSION, CRITFEAT'*) -> CRITFEAT''*.

[r0327] instr(try(LOCALCALL, EXPRESSION), CRITFEAT*) -> CRITFEAT''*
         :
           localcall(LOCALCALL, CRITFEAT*) -> CRITFEAT'*,
           " ?= ",
           expression(EXPRESSION, CRITFEAT'*) -> CRITFEAT''*.

[r0330] elseif([<EXPRESSION,INSTRUCTION*>|IFTHEN*], CRITFEAT*) -> CRITFEAT3*
         :
           nl,
           "      elseif ",
           expression(EXPRESSION, CRITFEAT*) -> CRITFEAT1*,
           " then",
           compound(INSTRUCTION*, CRITFEAT1*) -> CRITFEAT2*,
           elseif(IFTHEN*, CRITFEAT2*) -> CRITFEAT3*.

[r0331] elseif([], CRITFEAT*) -> CRITFEAT* : .

[r0332] else(INSTRUCTION*, CRITFEAT*) -> CRITFEAT'*
         :
           nl,
           "      else",
           compound(INSTRUCTION*, CRITFEAT*) -> CRITFEAT'*.

[r0333] else([], CRITFEAT*) -> CRITFEAT* : .

[r0340] call(call1(LOCALCALL), CRITFEAT*) -> CRITFEAT'*
         :
           localcall(LOCALCALL, CRITFEAT*) -> CRITFEAT'*.

[r0341] call(call2(LOCALCALL1, LOCALCALL2), CRITFEAT*) -> CRITFEAT''*
         :
           localcall(LOCALCALL1, CRITFEAT*) -> CRITFEAT'*,
           ".",
           localcall(LOCALCALL2, CRITFEAT'*) -> CRITFEAT''*.

[r0342] localcall(loccall([]), [<_, _, IDfname>|CRITFEAT*]) -> CRITFEAT*
         :
           id(IDfname).

[r0343] localcall(loccall(EXPRESSION*), [<_, _, IDfname>|CRITFEAT*]) -> CRITFEAT'*
         :
           id(IDfname),
           "(",
           expressions(EXPRESSION*, CRITFEAT*) -> CRITFEAT'*,
           ")".
```

```
[r0350] expressions([EXPRESSION|EXPRESSION*], CRITFEAT*) -> CRITFEAT''*
         :
           expression(EXPRESSION, CRITFEAT*) -> CRITFEAT'*,
           expressions2(EXPRESSION*, CRITFEAT'*) -> CRITFEAT''*.

[r0351] expressions2([EXPRESSION|EXPRESSION*], CRITFEAT*) -> CRITFEAT''*
         :
           ", ",
           expression(EXPRESSION, CRITFEAT*) -> CRITFEAT'*,
           expressions2(EXPRESSION*, CRITFEAT'*) -> CRITFEAT''*.

[r0352] expressions2([], CRITFEAT*) -> CRITFEAT* : .

[r0353] expression(simpleexpr(EXPRESSION), CRITFEAT*) -> CRITFEAT'*
         :
           sexpression(EXPRESSION, CRITFEAT*) -> CRITFEAT'*.

[r0354] expression(rekexpr(EXPRESSION), CRITFEAT*) -> CRITFEAT'*
         :
           rexpression(EXPRESSION, CRITFEAT*) -> CRITFEAT'*.

[r0355] sexpression(exprconst(CONSTANT), CRITFEAT*) -> CRITFEAT*
         :
           constant(CONSTANT).

[r0356] sexpression(expr(EXPRESSION), CRITFEAT*) -> CRITFEAT'*
         :
           "(",
           expression(EXPRESSION, CRITFEAT*) -> CRITFEAT'*,
           ")".

[r0357] sexpression(unexpr(UNARY, EXPRESSION), CRITFEAT*) -> CRITFEAT'*
         :
           unary(UNARY),
           expression(EXPRESSION, CRITFEAT*) -> CRITFEAT'*.

[r0358] sexpression(exprcall(CALL), CRITFEAT*) -> CRITFEAT'*
         :
           call(CALL, CRITFEAT*) -> CRITFEAT'*.

[r0359] rexpression(binexpr(EXPRESSION1, BINARY, EXPRESSION2), CRITFEAT*) -> CRITFEAT''*
         :
           sexpression(EXPRESSION1, CRITFEAT*) -> CRITFEAT'*,
           binary(BINARY),
           expression(EXPRESSION2, CRITFEAT'*) -> CRITFEAT''*.

[r0360] unary('-') : "-".

[r0361] unary('not') : "not ".

[r0370] binary('=') : " = ".

[r0371] binary('/=') : " /= ".

[r0372] binary('+') : " + ".

[r0373] binary('-') : " - ".

[r0374] binary('*') : " * ".

[r0375] binary('/') : " / ".

[r0376] binary('or') : " or ".

[r0377] binary('and') : " and ".

[r0380] constant(constint(INTEGER))
         :
           integer(INTEGER).

[r0381] constant(constreal(REAL))
         :
           real(REAL).

[r0382] constant(conststring(STRING))
         :
           string(STRING).
```

```
[r0390] integer(<’-’, NAT>)
         :
           "-",
           nat(NAT).

[r0391] integer(<’’, NAT>)
         :
           nat(NAT).

[r400] real(<INTEGER, NAT>)
         :
           integer(INTEGER),
           ".",
           nat(NAT).
```

A.11 Die Fehlerausgabe mit debug.pra

```
% Datei:         debug.pra
% Dokumentation: Unterkapitel 4.3 Die erste Transformation

Procedure twoselectsc With IDfname
Begin
 Nl;
 Write ’ERROR: 2 selects at component feature ’;
 Write IDfname;
 Nl; Nl;
End Procedure.

Procedure twoselectsp With IDfname
Begin
 Nl;
 Write ’ERROR: 2 selects at pattern feature ’;
 Write IDfname;
 Nl; Nl;
End Procedure.

Procedure noselectc With <IDcname, IDfname>
Begin
 Nl;
 Write ’ERROR: No select at component feature ’;
 Write IDfname;
 Write ’ in component ’;
 Write IDcname;
 Nl; Nl;
End Procedure.

Procedure noselectp With IDfname
Begin
 Nl;
 Write ’ERROR: No select at pattern feature ’;
 Write IDfname;
 Nl; Nl;
End Procedure.

Procedure unknownsel With IDselect*
Begin
 Nl;
 Write ’ERROR: Following selects don’t make sense: ’;
 Write IDselect*;
 Nl; Nl;
End Procedure.

Procedure unknowncastsel With <IDfname, IDref>
Begin
 Nl;
 Write ’ERROR: Following select in cast clause doesn’t make sense: ’;
 Write IDfname;
 Write ’ from <’;
 Write IDref;
 Write ’>’;
 Nl; Nl;
End Procedure.
```

```
Procedure unknowncastren With <IDfnamebefore, IDref, IDfnameafter>
Begin
 Nl;
 Write ’ERROR: Following rename in cast clause doesn’t make sense: ’;
 Write IDfnamebefore;
 Write ’ from <’;
 Write IDref;
 Write ’> as ’;
 Write IDfnameafter;
 Nl; Nl;
End Procedure.

Procedure unknownren With <IDrenbefore, IDrenafter>
Begin
 Nl;
 Write ’ERROR: Following rename doesn’t make sense: ’;
 Write IDrenbefore;
 Write ’ as ’;
 Write IDrenafter;
 Nl; Nl;
End Procedure.


Procedure difftypes With <IDfname, IDtype>
Begin
 Nl;
 Write ’WARNING: multiple inheritance with different resulttypes at feature ’;
 Write IDfname;
 Nl;
 Write ’         type ’;
 Write IDtype;
 Nl;
 Write ’ selected’;
 Nl; Nl;
End Procedure.

Procedure externparam With <IDrenbefore, IDrenafter>
Begin
 Nl;
 Write ’ERROR: rename extern parameter type ’;
 Write IDrenbefore;
 Write ’ as ’;
 Write IDrenafter;
 Write ’ is not allowed’;
 Nl; Nl;
End Procedure.
```

A.12 Das Beispiel in source.pal

```
-- Datei:         source.pal
-- Dokumentation: Kapitel 5 Einführung in die Sprache PaL

/* eine normale Liste */

pattern PLIST

  creation
    make

  component LIST

    creation
      make

    feature make is
      do
        first := void;
        cursor := void
      end -- make
```

```
feature add(an_item: ITEM) is
  local
    temp_item: ITEM
  do
    if is_empty
    then
      first := an_item;
      rewind
    else
      temp_item := cursor;
      cursor := an_item;
      an_item.set_next(temp_item.next);
      temp_item.set_next(an_item)
    end
  end -- add

feature delete is
  local
    temp_item: ITEM
  do
    if not is_empty
    then
      if (first = temp_item1)
      then
        first := cursor.next;
        rewind
      else
        temp_item := prior(cursor);
        temp_item.set_next(cursor.next);
        cursor := temp_item
      end
    end
  end -- delete

feature get: ITEM is
  do
    result := cursor
  end -- get

feature rewind is
  do
    cursor := first
  end -- rewind

feature next_item is
   do
    if (cursor /= void)
    then
      cursor := cursor.next
    end
  end -- next_item

feature prior(an_item : ITEM): ITEM is
  local
    temp_item: ITEM
  do
    from
      temp_item := cursor;
      rewind
    until
      ((cursor.next = an_item) or (cursor.next = void))
    loop
      next_item
    end;
    result := cursor;
    cursor := temp_item
  end -- prior

feature is_not_valid: Boolean is
  do
    result := (cursor = void)
  end -- is_not_valid

feature is_empty: Boolean is
  do
    result := (first = void)
  end -- is_empty
```

```
feature first: ITEM

    feature cursor: ITEM

  end -- component LIST

  component ITEM
    creation make

    feature make is
      do
        next := void
      end -- make

    feature next: ITEM

    feature set_next(an_item: ITEM) is
      do
        next := an_item
      end -- set_next

  end -- component ITEM

  extern feature make is
    do
      !!the_list.make
    end -- make

  intern feature the_list: LIST

end -- pattern PLIST

/* Zusammensetzung des Kompositum
   LIST wird auf COMPOSITE und
   ITEM auf COMPONENT abgebildet */

pattern PCOMPOSITE

  refine

    PLIST
      rename
        LIST as COMPOSITE,
        ITEM as COMPONENT,
        the_list as the_compos
    end

  creation
    make

  component COMPONENT

    feature operation(a_parameter: PARAMETER) is
      deferred
      end

  end -- component COMPONENT

  component LEAF

    inherit
      COMPONENT

    creation
      make

  end -- component LEAF

  component COMPOSITE

    inherit
      COMPONENT

    feature operation(a_parameter: PARAMETER) is
      do
        operation_items(a_parameter)
      end
```

```
    feature operation_items(a_parameter: PARAMETER) is
      do
        from
          rewind
        until
          is_not_valid
        loop
          cursor.operation(a_parameter);
          next_item
        end
      end -- operation_items

  end -- component COMPOSITE

  component PARAMETER
  end -- component PARAMETER

  extern feature make is
    do
      !!the_compos.make
    end -- make

end -- pattern PCOMPOSITE


pattern PVISITOR

  component VISITOR

    feature visit_concrete_element(anelement: CONCRETE_ELEMENT) is
      deferred
      end -- visit_concrete_element

    feature make is
      deferred
      end -- make

  end -- component VISITOR

  component CONCRETE_VISITOR

    inherit
      VISITOR

    creation
      make

    feature make is
      do
      end -- make

  end -- component CONCRETE_VISITOR

  component ELEMENT

    feature accept(avisitor: VISITOR) is
      deferred
      end -- accept

  end -- component ELEMENT

  component CONCRETE_ELEMENT

    inherit
      ELEMENT

    feature accept(avisitor: VISITOR) is
      do
        avisitor.visit_concrete_element(current)
      end -- accept

  end -- component CONCRETE_ELEMENT

end -- pattern PVISITOR

/* Verknüpfung von Kompositum und Besucher
   CONCRETE wird auf COMPONENT und
   CONCRETE_ELEMENT auf COMPOSITE und LEAF abgebildet */
```

```
pattern PCOMPOSITEVISITOR

  refine

    select PVISITOR <Ref_COMPOSITE>
      rename
        ELEMENT as COMPONENT,
        CONCRETE_ELEMENT as COMPOSITE
    end

    PVISITOR <Ref_LEAF>
      rename
        ELEMENT as COMPONENT,
        CONCRETE_ELEMENT as LEAF
    end

    PCOMPOSITE
      rename
        PARAMETER as VISITOR
    end

  component VISITOR

    cast
      rename
        visit_concrete_element from <Ref_COMPOSITE> as visit_composite,
        visit_concrete_element from <Ref_LEAF> as visit_leaf
    end

  end -- component VISITOR

  component COMPONENT

    cast
      rename
        operation as accept
    end

  end -- component COMPONENT


  component COMPOSITE

    cast
      rename
        operation_items as accept_items
    end

  end -- component COMPOSITE

end -- pattern PCOMPOSITEVISITOR

/* Das fertige Programm
   CONCRETE_VISITOR wird auf FRAME_VISITOR und CIRCLE_VISITOR abgebildet
   hinzu kommt eine make Routine zur Demonstration */

pattern PCOMPOSITEVISITOR2

  refine

    select PCOMPOSITEVISITOR
      rename
        CONCRETE_VISITOR as FRAME_VISITOR
    end

    PCOMPOSITEVISITOR
      rename
        CONCRETE_VISITOR as CIRCLE_VISITOR
    end

  creation
    make
```

```
  component FRAME_VISITOR

    feature visit_composite(acomposite: COMPOSITE) is
      do
        io.put_string("composite[ ");
        acomposite.accept_items(current);
        io.put_string("] ")
      end -- visit_composite

    feature visit_leaf(aleaf: LEAF) is
      do
        io.put_string("[leaf] ")
      end -- visit_leaf

  end -- component FRAME_VISITOR

  component CIRCLE_VISITOR

    feature visit_composite(acomposite: COMPOSITE) is
      do
        io.put_string("composite( ");
        acomposite.accept_items(current);
        io.put_string(") ")
      end -- visit_composite

    feature visit_leaf(aleaf: LEAF) is
      do
        io.put_string("(leaf) ")
      end -- visit_leaf

  end -- component CIRCLE_VISITOR

  extern feature make is
    local
      templeaf: LEAF;
      tempcompos: COMPOSITE;
      frame: FRAME_VISITOR;
      circle: CIRCLE_VISITOR
    do
      !!the_compos.make;
      !!templeaf.make;
      the_compos.add(templeaf);
      !!templeaf.make;
      the_compos.add(templeaf);
      !!tempcompos.make;
      !!templeaf.make;
      tempcompos.add(templeaf);
      !!templeaf.make;
      tempcompos.add(templeaf);
      !!templeaf.make;
      tempcompos.add(templeaf);
      the_compos.add(tempcompos);

      !!frame.make;
      !!circle.make;
      the_compos.accept(frame);
      io.put_string("%N");
      the_compos.accept(circle)
    end -- make

end -- pattern PCOMPOSITEVISITOR2

/* Ergebnis ist eine konsequente Trennung zwischen Struktur im Kompositum
   und Verhalten in den Besuchern */
```

Literaturverzeichnis

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns. Addison-Wesley. 1995. ISBN 0-201-63361-2

[2] Normen Seemann. A Design Patterns Oriented Programming Environment. University of Rostock, 1999. Master's Thesis

[3] Bünnig, Stefan; Forbrig, Peter; Lämmel, Ralf; Seemann, Normen: A Programming Language for Design Patterns, ATPS'99, To be published by Springer

[4] Pete Thomas and Ray Weedon. Object- Oriented Programming in Eiffel. Addison- Wesley. 1995. ISBN 0-201-59387-40

[5] Ralf Lämmel, Günter Riedewald and Jörg Harm. Specification formalisms in LDL. Preprint CS-08-96, University of Rostock, Department of Computer Science, August 1997

[6] Jan Bosch, Görel Hedin and Kai Koskimies. Language Support for Design Patterns and Frameworks, Jyväskylä, Finland, 1997

[7] Jan Bosch. Design Patterns as Language Constructs. Journal of Object-Oriented Programming, 1998

[8] Görel Hedin. Language Support for Design Patterns using Attribute Extensions, Aarhus University, Computer Science Department. in [6]

[9] Eyoun Eli Jacobsen. Design Patterns as Program Extracts, Aalborg University, Department of Computer Science. in [6]

Internet-Adressen

[10] LDL-Homepage
http://www.informatik.uni-rostock.de/FB/Praktik/psuet/ldl/

[11] SWI-Prolog
ftp://swi.psy.uva.nl/pub/SWI-Prolog/

[12] SmallEiffel
http://www.loria.fr/SmallEiffel

[13] Language Support for Design Patterns and Frameworks
http://bilbo.ide.hk-r.se/~bosch/lsdf/

Selbständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, Stefan Bünnig

Thesen

1. Vererbung, Delegation und generische Klassen sind die Mittel der Wiederverwendung in objektorientierten Sprachen. Diese Mittel sind nicht mächtig genug, um ganze Design Patterns oder andere Strukturen wiederzuverwenden.
2. In objektorientierten Programmiersprachen lassen sich Design Patterns nicht abstrakt vorimplementieren, um sie für eine konkrete Anwendung wiederzuverwenden. Alle für das Design Pattern erforderlichen Klassen und Hilfsmethoden müssen stets neu implementiert werden.
3. Design Patterns besitzen in objektorientierten Programmiersprachen keine First-Class-Repräsentation. Sie sind in der Klassenstruktur eines objektorientierten Programms schwer identifizierbar, was sich negativ auf die Wartbarkeit auswirkt.
4. Durch eine auf Design Patterns optimierte Sprache läßt sich der Einsatz von Design Patterns rationalisieren und die Wartbarkeit der Software erhöhen.
5. Mit der Programmiersprache *PaL* wurde die objektorientierte Programmierung erweitert, um die Probleme der herkömmlichen Sprachen im Umgang mit Design Patterns zu lösen.
6. *PaL* bietet das spezielle Sprachkonstrukt `pattern`, das es ermöglicht, ein Design Pattern zu implementieren. Dabei werden Teilnehmer durch Komponenten, deren Struktur durch objektorientierte Beziehungen und Interaktionen durch Methoden der Komponenten oder des Patterns ausgedrückt.
7. Mit der Sprache *PaL* lassen sich nicht nur Design Patterns, sondern auch andere Klassenstrukturen voneinander abgrenzen und benennen.
8. Das zentrale Mittel der Verfeinerung ist in *PaL* durch den `refine`-Operator realisiert. Durch ihn ist es möglich, Pattern wiederzuverwenden, zu kombinieren und zu erweitern.
9. Mit der Sprache *PaL* ist es möglich, Bibliotheken von abstrakt implementierten Design Patterns und anderen Klassenstrukturen anzulegen. Die leichte Wiederverwendung, Anpassung und Kombination der Design Patterns fördert deren Anwendung.
10. In *PaL* besteht zusätzlich die Möglichkeit, Pattern zu instanziieren. Wie beim Substitutionsprinzip in objektorientierten Programmiersprachen kann eine Instanz eines Patterns durch die Instanz eines davon verfeinerten Patterns substituiert werden.
11. Durch die Instanziierung ganzer Pattern ist es möglich, Programme mit erhöhter Modularität zu Entwickeln und den Aufbau ganzer Klassenstrukturen zu verbergen und auf eine Schnittstelle zu abstrahieren.
12. Eiffel ist eine geeignete Zielsprache für einen Prototyp-Präcompiler für *PaL*. Auf diese Weise kann ein gut lesbares Compilat erzeugt werden.
13. LDL ermöglicht es durch seine Formalismen, das Parsen des *PaL*-Quelltextes, die Transformation der Syntaxbäume und das Generieren von Eiffel-Quelltext zu spezifizieren. Aufgrund der Ausführbarkeit dieser Spezifikationen entsteht als Ergebnis ein Prototyp-Präcompiler.