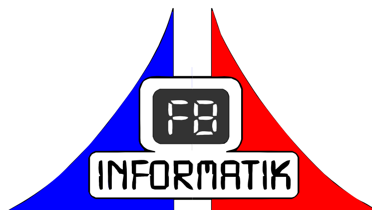
Patternorientierte Programmierung am Anwendungsbeispiel

Studienarbeit

Universität Rostock



Fachbereich Informatik

vorgestellt von Bünnig, Stefan
geboren am 13.08.1974 in Rathenow
Matrikel-Nr.: 094200949

und von Seemann, Normen
geboren am 31.05.1976 in Rostock
Matrikel-Nr.: 094200998

Betreuer: Prof. Dr.-Ing. habil. Peter Forbrig,
Dr.-Ing. Ralf Lämmel
Abgabedatum: 15.11.1999

Zusammenfassung

Diese Studienarbeit ist eine Anwendbarkeitsstudie des in [4] vorgestellten Modells und der in [1] vorgestellten Programmiersprache *PaL*. Zunächst werden die in [3] aufgeführten Design Patterns in *PaL* implementiert. Die Umsetzung der Design Patterns in die Sprache *PaL* wird ausführlich dokumentiert und diskutiert.
Der patternorientierte Entwurf wird danach als Erweiterung des objektorientierten Entwurfs eingeführt. Bei dieser neuen Form des Entwurfs steht die Wiederverwendung von Patterns im Vordergrund. Ausgehend von elementaren Design Patterns und dem Baukastenprinzip entstehen durch Verfeinerung komplexere, applikationsspezifische Patterns. Auf Grundlage der Standardbibliothek der Design Patterns wird unter Nutzung dieser Entwurfsstrategie die Zeichenapplikation *DrawIt* entwickelt. *DrawIt* stellt somit ein nichttriviales Beispiel für die Anwendbarkeit des patternorientierten Paradigmas dar.

Abstract

The aim of this study thesis is to present a case study to show the applicability of the model introduced in [4] and the language *PaL* introduced in [1]. First, all design patterns listed in [3] are implemented in *PaL* in order to create a standard library of design patterns. For this purpose, these patterns have to be formalized so that later extensions can be applied easily. Programming issues including considerations for later reuse are thoroughly discussed.
The pattern oriented design can be conceived as extension of the object oriented design. This newly introduced way of designing software systems emphasizes the reuse of patterns. Starting from basic patterns, more complex and application specific patterns are constructed. Finally, the drawing application *DrawIt* is developed based on the pattern oriented design and the standard library of design patterns. In this way, a non-trivial example for pattern oriented software engineering is given.

CR-classification

D.1.5, D.2.1, D.2.2, D.2.3, D2.10, D.2.13, D.3.3, F.3.3

Keywords

Programmierungstechniken, Design Patterns, Softwareengineering, Softwareentwicklung, Wiederverwendbarkeit, Programmiersprachen, Anwendbarkeitsstudie

Inhaltsverzeichnis

1 Einführung **5**

2 Eine Standardbibliothek grundlegender Design Patterns **7**

2.1 Anpassung der Design Patterns an die Sprache *PaL* . . . 7
2.2 Dokumentation der Patterns . . . 8
2.3 Hilfspatterns . . . 9
2.3.1 Container (Container) . . . 11
2.3.2 Liste (List) . . . 12
2.3.3 Parameter (Parameter) . . . 15
2.4 Die Bibliothek der Design Patterns . . . 17
2.4.1 Fabrikmethode (Factory Method) . . . 17
2.4.2 Abstrakte Fabrik (Abstract Factory) . . . 20
2.4.3 Iterator (Iterator) . . . 24
2.4.4 Kompositum (Composite) . . . 28
2.4.5 Interpreter (Interpreter) . . . 31
2.4.6 Fliegengewicht (Flyweight) . . . 33
2.4.7 Dekorierer (Decorator) . . . 36
2.4.8 Proxy (Proxy) . . . 39
2.4.9 Zuständigkeitskette (Chain of Responsibility) . . . 42
2.4.10 Besucher (Visitor) . . . 44
2.4.11 Strategie (Strategy) . . . 47
2.4.12 Zustand (State) . . . 50
2.4.13 Brücke (Bridge) . . . 52
2.4.14 Befehl (Command) . . . 55
2.4.15 Memento (Memento) . . . 58
2.4.16 Beobachter (Observer) . . . 60
2.4.17 Vermittler (Mediator) . . . 63
2.4.18 Erbauer (Builder) . . . 65
2.4.19 Prototyp (Prototype) . . . 67
2.4.20 Singleton (Singleton) . . . 69
2.4.21 Fassade (Facade) . . . 71
2.4.22 Adapter (Adapter) . . . 73
2.4.23 Schablonenmethode (Template Method) . . . 75
2.5 Beispiele für die Kombination von Patterns . . . 77
2.5.1 Liste + Iterator . . . 77
2.5.2 Kompositum + Iterator . . . 78
2.5.3 Kompositum + Liste + Iterator . . . 79
2.5.4 Kompositum + Dekorierer . . . 80
2.5.5 Beobachter + Liste + Iterator . . . 81

3 Eine Zeichenapplikation als Anwendungsstudie **83**
3.1 Die Funktionalität von *DrawIt* 83
3.2 Entwurf der Benutzungsschnittstelle 84
3.3 Entwurf der Design Patterns der Applikation 87
3.3.1 Entwurfsprobleme 87
3.3.2 Interne Repräsentation der Grafiken 88
3.3.3 Funktionalität für das grafische Kompositum 92
3.3.4 Realisierung von Markierern 95
3.3.5 Realisierung von Verbindern 98
3.3.6 Kommandos 101
3.3.7 Benutzungsschnittstelle 104
3.4 Herleitung der Design Patterns der Applikation 107
3.4.1 Herleitung der allgemeinen besuchergestützten Befehlskette 109
3.4.2 Herleitung des abstrakten grafischen Kompositums 112
3.4.3 Herleitung des Applikationspatterns 115

4 Auswertung **117**
4.1 Spezielle Probleme und mögliche Lösungen 117
4.2 Resümee 119

A **121**
A.1 Erweiterung der Sprache *PaL* und des Compilers 121
A.2 Die Komponentenstruktur des Design Patterns `PDOCUMENT_WINDOW` 123
A.3 Der Quelltext von DrawIt 125

Kapitel 1

Einführung

Seit einigen Jahren werden Design Patterns[1] bei der Entwicklung größerer Softwareprojekte in zunehmendem Maße erfolgreich angewendet. Ein *Design Pattern* beschreibt die Lösung für ein ständig wiederkehrendes Problem. Ein Design Patterns sollte so allgemein wie möglich beschrieben werden, um die Anwendbarkeit in jedem erdenklichen Fall zu garantieren. Auf der anderen Seite muss das Design Pattern aber auch so konkret definiert sein, dass dessen Anwendung eine Hilfe in der Softwareentwicklung darstellt. In [3] wurden 23 grundlegende Design Patterns definiert und kategorisiert. Diese Design Patterns basieren auf dem objektorientierten Paradigma, sind dort jedoch nur informal beschrieben.

In [4] wurde nun ein formales Modell zur Definition von Design Patterns entwickelt, welches als Erweiterung des objektorientierten Modells zu verstehen ist. Es handelt sich hierbei um ein sogenanntes *Execution Model*, das auch als semantischer Raum für eine *patternorientierte* Programmiersprache genutzt werden kann. Eine solche patternorientierte Sprache, d.h. eine Sprache die syntaktisch und semantisch die Beschreibung und die Anwendung von Design Patterns unterstützt, wurde in [4] oder auch in [1] eingeführt.

Das Ziel dieser Studienarbeit ist es, die Anwendbarkeit des patternorientierten Paradigmas sowie deren Vorteile gegenüber dem objektorientiertem Paradigma zu demonstrieren. Die Sprache *PaL* aus [1] zusammen mit dem dazugehörigen Compiler sollen hierfür den Ausgangspunkt bilden.

In Kapitel 2 von Stefan Bünnig werden zunächst die in [3] informal beschriebenen Design Patterns in der Sprache *PaL* implementiert und als *Standardbibliothek* zur Verfügung gestellt. Dafür müssen die Design Patterns aus [3] formalisiert und verallgemeinert werden.

Unter Benutzung dieser Standardbibliothek wird im Kapitel 3 von Normen Seemann eine komplexere Anwendung in *PaL* realisiert. Diese Anwendung soll als Fallstudie betrachtet werden, um die verbesserten Wiederverwendungsmöglichkeiten des patternorientierten Modells und von *PaL* zu zeigen.

In Kapitel 4 wird abschließend die Arbeit mit dem patternorientierten Modell und der Sprache *PaL* ausgewertet. Dabei soll deren Anwendbarkeit und Durchführbarkeit im Rahmen der komplexen Anwendung *DrawIt* untersucht werden.

[1] In dieser Studienarbeit werden wir den Begriff *Design Pattern* immer auf das Gebiet der Softwaretechnik beziehen.

Kapitel 2

Eine Standardbibliothek grundlegender Design Patterns

Die Sprache *PaL* soll den Einsatz von Design Patterns erleichtern und deren Wiederverwendbarkeit und Konfigurierbarkeit ermöglichen. Daher sollen die Design Pattern aus [3] zunächst in der Sprache *PaL* implementiert und zu einer Patternbibliothek zusammengefasst werden. Die Umsetzung der 23 Design Patterns wird in diesem Kapitel ausführlich behandelt. Im Anschluss daran werden einige Standardkombinationen der Patterns aufgeführt.

2.1 Anpassung der Design Patterns an die Sprache *PaL*

Die in [3] aufgeführten Design Patterns beschreiben Ideen zur Lösung von Designproblemen, ohne die Patterns auf einen bestimmten Anwendungsfall festzulegen. Ein in *PaL* implementiertes Design Pattern muss genauso vielseitig einsetzbar sein, wie die beschriebene Designidee.
Design Patterns erleichtern dem Entwickler die Erweiterbarkeit der Software. Sie bieten meist mehrere Schnittstellen zur Erweiterung und Konfiguration. Um dem Leser von [3] diese Möglichkeiten der Konfiguration begreiflich zu machen, werden dort für jede Art der Erweiterung zwei konkrete Ausprägungen angegeben. Zum Beispiel im Pattern Abstrakte Fabrik (20) werden in der Klassenstruktur zwei konkrete Fabriken angegeben, die jeweils zwei konkrete Produkte erzeugen können, die durch zwei abstrakte Produkte verallgemeinert werden. Diese Beschreibung fördert zwar das Verständnis des Design Patterns, ist aber für eine vielseitig wiederverwendbare Implementation in *PaL* zu speziell.
Die Sprache *PaL* ermöglicht durch das Mittel der Verfeinerung die Konfiguration der Design Patterns. In dieser Bibliothek werden daher die allgemeinsten Ausprägungen der Design Patterns imlementiert. Für das Design Pattern Abstrakte Fabrik bedeutet das, es wird nur eine konkrete Fabrik implementiert, die nur ein konkretes Produkt erzeugt, dessen Schnittstelle durch ein abstraktes Produkt dargestellt wird. In einer speziellen Anwendung können dann durch Mehrfachverfeinerung so viele konkrete Fabriken mit so vielen Produkten angelegt werden, wie erforderlich sind.

Die Interaktionen zwischen den Teilnehmerkomponenten sind ein wesentlicher Bestandteil eines Design Patterns. Objekte schicken Botschaften an andere Objekte, die dort Operationen auslösen. Oft ergibt sich erst in einer konkreten Anwendung eines Design Patterns, welche Übergabeparameter diese Operationen benötigen. In solchen Fällen werden in [3] gar keine Übergabeparameter angegeben.
Die Sprache *PaL* bietet zwar die Möglichkeit, Komponenten zu duplizieren und deren Features zu vervielfältigen, aber es gibt keine Möglichkeit, die Anzahl der Parameter einer Methode zu ändern. Da bei den meisten konkreten Anwendungen eines Design Patterns mindestens

ein Übergabeparameter pro Operation erforderlich ist, wäre ein Design Pattern, das wie in [3] mit Operationen ohne Parameter implementiert wird, nur sehr bedingt wiederverwendbar. Daher wird allen Operationen, deren Anzahl der Parameter in einer konkreten Anwendung noch nicht bekannt ist, ein Parameterobjekt zugewiesen, das vom Typ einer Parameterkomponente ist.

```
feature operation(a_parameter: PARAMETER) is
    do
        ...
    end
```

Im Idealfall besitzt die Operation genau einen Parameter. In diesem Fall kann die Parameterkomponente bei der Verfeinerung zur konkreten Anwendung des Design Patterns auf den Typ des Parameters abgebildet werden.
Sind mehrere Parameter erforderlich, so wird der Parameterkomponente für jeden Übergabeparameter ein Attribut zugewiesen. Rückgabewerte von Funktionen lassen sich ebenfalls über ein Attribut in der Parameterkomponente realisieren.
Benötigt eine solche Operation bei einer Anwendung des Design Patterns keine Übergabeparameter, so ist die Parameterkomponente überflüssig. In solchen Fällen kann der Operation als Parameter ein `void` übergeben werden.

Durch diese Anpassungen unterscheiden sich die in *PaL* implementierten Design Patterns auf den ersten Blick von den in [3] beschriebenen. Gerade diese Anpassungen ermöglichen aber erst die generelle und vielseitige Anwendbarkeit der abstrakt implementierten Design Patterns.

Da es bei der Implementation der Design Patterns häufig vorkommt, dass die Anzahl der Parameter einer Methode noch nicht bekannt ist, wird im Unterkapitel „Hilfspatterns" ein primitives Pattern Parameter (15) vorbereitet, das in solchen Fällen verfeinert wird.

Der Katalog der Design Patterns in [3] ist unterteilt in *creational patterns*, *structural patterns* und *behavioural patterns*. Innerhalb dieser Kategorien sind die Design Patterns alphabetisch sortiert.
Die Implementationen der Design Patterns in *PaL* bauen teilweise aufeinander auf. Daher werden die Design Patterns hier so aufgeführt, dass Design Patterns, deren Implementation Vorraussetzung für andere Design Patterns ist, stets vor diesen behandelt werden.

2.2 Dokumentation der Patterns

In [3] werden die Design Patterns nach einem einheitlichen Muster beschrieben. Für jedes Design Pattern werden die folgenden Punkte behandelt:

- Mustername und Klassifizierung
- Zweck
- Auch bekannt als
- Motivation
- Anwendbarkeit
- Struktur
- Teilnehmer

- Interaktionen
- Konsequenzen
- Implementierung
- Beispielcode
- Bekannte Verwendungen
- Verwandte Muster

Das Ziel dieser Arbeit ist es nicht, die gesamten Patternbeschreibungen aus [3] zu wiederholen. Um dies zu vermeiden, wird bei der Dokumentation der *PaL*-Patternbibliothek auf dieses Buch aufgebaut und teilweise direkt auf spezielle Absätze referenziert. Zum besseren Verständnis der folgenden Dokumentation wird die Verwendung von [3] als Nachschlagewerk empfohlen.
In dieser Dokumentation der Patternbibliothek werden die folgenden Aspekte der Implementation und Anwendung der Patterns in der Sprache *PaL* behandelt.

Struktur: Hier wird das Patterndiagramm in der in [1] vorgestellten Notation abgebildet.

Implementierung: In diesem Abschnitt wird die konkrete Implementierung des Patterns besprochen. Es wird auf Abweichungen zu den in [3] beschriebenen Design Patterns und auf die dort aufgeführten Implementierungsvarianten eingegangen.

Code: Hier wird der *PaL*-Quelltext des Patterns angegeben.

Konfigurationsmöglichkeiten: An dieser Stelle werden die Varianten der Konfiguration des Design Patterns für den konkreten Einsatz beschrieben.

Kombination mit anderen Design Patterns: In diesem Abschnitt werden die Kombinationsmöglichkeiten mit anderen Design Patterns in *PaL* aufgezeigt. Diese werden mit dem Abschnitt „Verwandte Muster“ in [3] verglichen.

Anwendung in *DrawIt*: Sollte das Design Pattern in dem Anwendungprogramm *DrawIt*, das im Rahmen dieser Arbeit entwickelt wird, benutzt werden, dann wird der Einsatz an dieser Stelle kurz beschrieben.

Die Implementation in *PaL* kann ein Design Pattern aus [3] nicht vollständig reflektieren. Dies liegt daran, dass bestimmte, dort zur Beschreibung eines Design Patterns aufgeführte Unterpunkte Ideen informell beschreiben, die formal und speziell in *PaL* nicht umsetzbar sind.

Wie in [3] wird bei jeder Erwähnung eines Patterns hinter dem Namen in Klammern die Seite angegeben, auf der das Pattern dokumentiert ist. So kann die Dokumentation des Patterns schnell nachgeschlagen werden.
Bei der Implementation der Design Patterns in *PaL* werden die Bezeichner für Komponenten und Features von [3] übernommen. Da sich die Namen von Patterns und Komponenten teilweise gleichen, wird vor die Bezeichnung des Design Patterns in der *PaL*-Implementation der Buchstabe „P“ gesetzt. Bei Komponenten- und Patternbezeichnern werden nur Großbuchstaben verwendet, Bezeichner von Features enthalten ausschließlich Kleinbuchstaben. Zusammengesetzte Bezeichner werden durch einen Unterstrich „_“ voneinander getrennt.

2.3 Hilfspatterns

Bei den Design Patterns ist es häufig der Fall, dass Komponenten beliebig viele Instanzen einer anderen Komponente verwalten können. Der interne Aufbau solcher Container-Komponenten wird in [3] stets offengelassen. Es wird immer nur die Schnittstelle zum Einfügen

und Entfernen von Objekten beschrieben.
Das Hilfspattern Container (11) soll als Ausgangspattern für solche Design Patterns dienen, die Komponenten mit Containerfunktion enthalten. Dieses Container-Pattern lässt die Implementation der Container-Funktionalität noch offen, es deutet lediglich die Schnittstellen an.
Unter der Verwendung dieses Hilfspatterns lassen sich die Design Patterns ebenso abstrakt implementieren, wie sie in [3] beschrieben werden. Damit die Design Patterns aus der Patternbibliothek schnell praktisch eingesetzt werden können, ist eine konkrete Implementation der Container-Funktionalität erforderlich. Diese soll durch die Implementation des Patters Liste (12) zur Verfügung gestellt werden.

Als drittes Hilfspattern wird das bereits weiter oben erwähnte Pattern Parameter (15) dokumentiert.

2.3.1 Container (Container)

Struktur

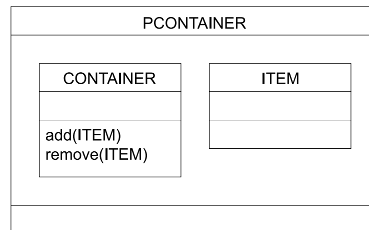


Abbildung 2.1: *Die Struktur des Hilfspatterns Container*

Implementierung

Das Pattern `PCONTAINER` besteht aus den zwei Komponenten `CONTAINER` und `ITEM`. Die Komponente `CONTAINER` enthält die abstrakten Features `add` und `remove` zum Hinzufügen und Entfernen von Objekten vom Typ `ITEM`.

Code

```
pattern PCONTAINER
    component CONTAINER
        feature add(an_item: ITEM) is
            deferred
            end – add
        feature remove(an_item: ITEM) is
            deferred
            end – remove
    end – component CONTAINER
    component ITEM
    end – component ITEM
end – pattern PCONTAINER
```

Konfigurationsmöglichkeiten

Eine konkrete Anwendung dieses Patterns wird im Folgenden am Pattern Liste (12) demonstriert.

Kombination mit anderen Design Patterns

Dieses Pattern ist Grundlage für die Implementation der Design Patterns Iterator (24), Kompositum (28), Fliegengewicht (33), Beobachter (60), Vermittler (63) und Erbauer (65).

Anwendung in *DrawIt*

Das Container-Pattern kommt in *DrawIt* überall dort zu Einsatz, wo die vorhergenannten Design Patterns verwendet werden.

2.3.2 Liste (List)

Struktur

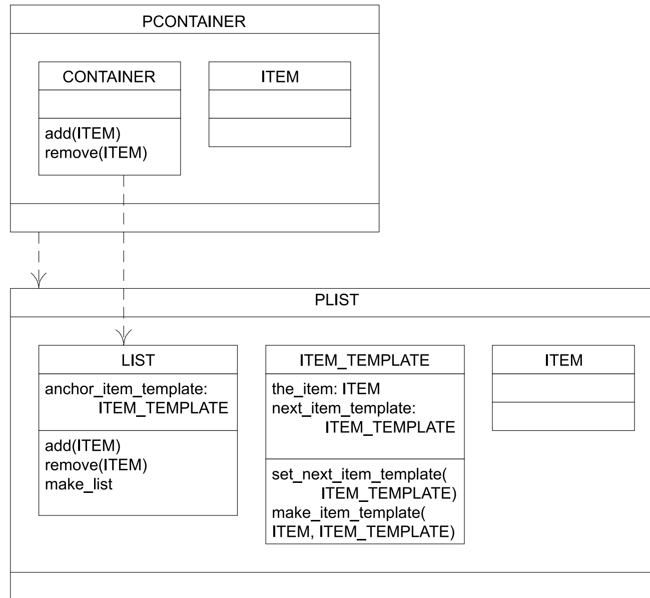


Abbildung 2.2: *Verfeinerung zun Hilfstpattern Liste*

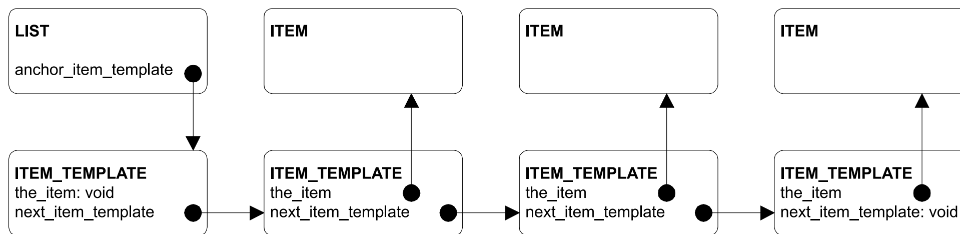


Abbildung 2.3: *Aufbau einer Liste mit drei Items*

Implementierung

Das Liste-Pattern stellt eine Verfeinerung des Container-Patterns dar und implementiert die konkrete interne Struktur eines Containers in Form einer einfach verketteten Liste. Die Funktionalität dieses Patterns beschränkt sich wie beim Pattern Container (11) auf das Einfügen und Entfernen von Items in die Liste bzw. aus der Liste. Zusätzlich wird eine Methode zur Erzeugung einer leeren Liste zur Verfügung gestellt.
An dieser Stelle soll noch keine Funktionalität zum Iterieren der Liste implementiert werden, diese wird durch eine konkrete Anwendung des Design Patterns Iterator (24) realisiert.

Die Sprache *PaL* lässt viele denkbare Kombinationen von Patterns zu. So kann es vorkommen, dass ein Item durch Mehrfachverfeinerung gleichzeitig Element zweier Listen ist. Bei solch einer Konstellation treten Probleme auf, wenn die Items ihre Nachfolger selbst verwalten. In diesem Fall müsste die gesamte Nachfolgerverwaltung der Komponente `ITEM` dupliziert werden. Dieser Ansatz bietet zu viele potentielle Fehlerquellen und ist zu umständlich in der Anwendung.

Die Komponente ITEM soll daher keine Funktionalität enthalten, die mit der Liste in Verbindung steht. Zur Nachfolgerverwaltung wird eine zusätzliche Komponente ITEM_TEMPLATE eingeführt. Diese Komponente stellt auch die Verbindung zwischen der Liste und den Items her. Die Abbildung 2.3 verdeutlicht den Aufbau einer Liste zur Laufzeit.

Eine leere Liste besteht aus einem LIST-Objekt, das mit dem Attribut anchor_item_template auf eine Instanz der Komponente ITEM_TEMPLATE verweist. Beim Hinzufügen eines Items in die Liste wird an das letzte ITEM_TEMPLATE ein weiteres angehängt, das eine Referenz auf das eingefügte ITEM hat. Zum Entfernen von Items muss nur das zugehörige ITEM_TEMPLATE aufgefunden werden und dessen Vorgänger mit dem Nachfolger verknüpft werden. Das erste ITEM_TEMPLATE-Objekt in der Liste besitzt kein Item, es dient der einheitlichen Behandlung von leeren und gefüllten Listen.

Code

```
pattern PLIST
    refine
        PCONTAINER
            rename
                CONTAINER as LIST
        end
    component LIST
        creation
            make_list
        feature make_list is
            do
                !!anchor_item_template.make_item_template(void, void)
            end – make_list
        feature add(an_item: ITEM) is ...
        feature remove(an_item: ITEM) is ...
        feature anchor_item_template: ITEM_TEMPLATE
    end – component LIST
    component ITEM_TEMPLATE
        creation
            make_item_template
        feature the_item: ITEM
        feature next_item_template: ITEM_TEMPLATE
        feature set_next_item_template(new_next_item_template: ITEM_TEMPLATE) is
            do
                next_item_template := new_next_item_template
            end
        feature make_item_template(new_the_item: ITEM; new_next_item_template: ITEM_TEMPLATE) is
            do
                the_item := new_the_item;
                set_next_item_template(new_next_item_template)
            end
    end
end – pattern PLIST
```

Konfigurationsmöglichkeiten

Das Pattern Liste dient der Wiederverwendung der Listen-Implementation. Beim Einsatz des Patterns werden die Komponenten LIST und ITEM entsprechend den Bezeichnungen in der konkreten Anwendung umbenannt.

Kombination mit anderen Design Patterns

Dieses Pattern ist mit allen Design Pattern kombinierbar, die durch Verfeinerung des Patterns Container (11) implementiert wurden. Das sind die Design Patterns Iterator (24), Kompositum (28), Fliegengewicht (33), Beobachter (60), Vermittler (63) und Erbauer (65).
In Kombination mit dem Iterator-Pattern kann das Pattern Liste die Container-Funktionalität für die aufgezählten, abstrakt implementierten Design Patterns zur Verfügung stellen.
(Siehe dazu auch Abschnitte 2.5.1, 2.5.3 und 2.5.5.)

Anwendung in *DrawIt*

Alle Container werden in dieser Anwendung durch das Pattern Liste implementiert. Es kommt daher überall dort zum Einsatz, wo die aufgezählten Design Patterns angewendet werden.
Zusätzlich wird dieses Pattern für die Verwaltung der Befehlskette zur Realisierung der Undo- und Redo-Funktionalität eingesetzt.

2.3.3 Parameter (Parameter)

Struktur

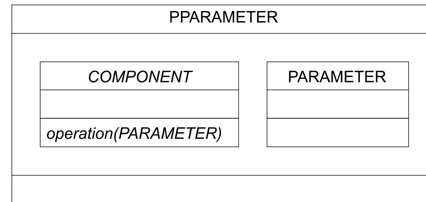


Abbildung 2.4: *Die Struktur des Hilfspatterns Parameter*

Implementierung

Das Pattern besteht aus zwei Komponenten. Die Komponente `COMPONENT` enthält die Operation `operation` mit dem Parameter `a_parameter` vom Typ der zweiten Komponente `PARAMETER`. Diese Komponente enthält noch keine Implementation.

Code

```
pattern PPARAMETER
    component COMPONENT
        feature operation(a_parameter: PARAMETER) is
            deferred
            end – operation
    end – component COMPONENT
    component PARAMETER
    end – component PARAMETER
end – pattern PPARAMETER
```

Konfigurationsmöglichkeiten

Bei der Anwendung dieses Patterns wird die Komponente `COMPONENT` in die Komponente umbenannt, welche die Methode mit unbekannter Parameteranzahl enthält. Das Feature `operation` wird entsprechend dieser Methode umbenannt.
Die Anwendung dieses Hilfspatterns sieht etwa wie folgt aus:

```
pattern PPATTERN_NAME
    refine
        PPARAMETER
            rename
                COMPONENT as NEW_COMPONENT_NAME
        end
    component NEW_COMPONENT_NAME
        cast
            rename
                operation as new_operation_name
        end
    end – component NEW_COMPONENT_NAME
end – pattern PPATTERN_NAME
```

Kombination mit anderen Design Patterns

Das Parameter-Pattern ist ein Hilfspattern, das bei der Implementation der Patternbibliothek immer dann verfeinert wird, wenn Komponenten Operationen mit unbekannter Parameteranzahl enthalten.
Es kommt bei folgenden Design Patterns zum Einsatz: Kompositum (28), Interpreter (31), Fliegengewicht (33), Dekorierer (36), Proxy (39), Zuständigkeitskette (42), Strategie (47), Zustand (50), Brücke (52) und Befehl (55).

Anwendung in *DrawIt*

Das Pattern kommt in *DrawIt* nur indirekt durch die aufgezählten Design Patterns zur Anwendung.

2.4 Die Bibliothek der Design Patterns

2.4.1 Fabrikmethode (Factory Method)

Struktur

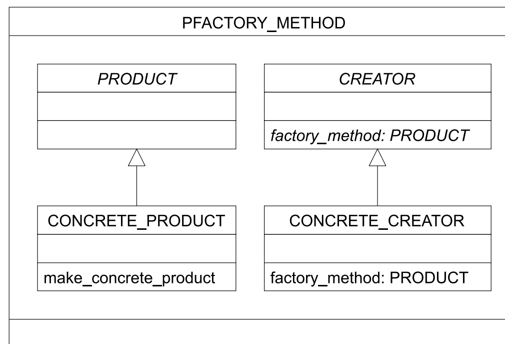


Abbildung 2.5: *Struktur des Patterns Fabrikmethode*

Implementierung

Dieses Design Pattern kann ohne Anpassungen in der Sprache *PaL* implementiert werden. Lediglich die in [3] in der Klasse `Creator` aufgeführte Methode `AnOperation` wird vernachlässigt. Diese Methode deutet nur an, wie die Fabrikmethode aufgerufen wird. Sie hat aber keinen substanziellen Wert für dieses Design Pattern.

Bei der Implementation des Design Patterns wird davon ausgegangen, dass die Erzeugung der Komponente `CONCRETE_PRODUCT` ohne Parameter auskommt. Nur in dem Fall, dass bei einer konkreten Anwendung des Patterns bei der Erzeugung der Komponente `CONCRETE_PRODUCT` Parameter benötigt werden, muss dieser Komponente eine neue Creation-Methode hinzugefügt und das Feature `factory_method` von `CONCRETE_CREATOR` überschrieben werden. Im Normalfall genügt die Standardimplementation der Fabrikmethode.

In [3] werden die folgenden auch für die Sprache *PaL* relevanten Aspekte der Implementation besprochen:

1. *Zwei größere Variationen.* In der einen Variante ist die Komponente `CREATOR` abstrakt und enthält keine Implementation der Fabrikmethode, in der anderen Variante besitzt die Komponente `CREATOR` bereits eine Standardimplementation der Fabrikmethode.
   Hier wurde die erste Variante implementiert, da sie sicherer ist. Durch das Mittel der Verfeinerung ist eine Standardimplementation der Fabrikmethode in der Komponente `CREATOR` nicht erforderlich, da bei mehreren konkreten Creator-Komponenten die Standardimplementation der Fabrikmethode automatisch durch die Duplizierung der Komponente `CONCRETE_CREATOR` übernommen wird.

2. *Parametrisierbare Fabrikmethoden.* In dieser Variante des Design Patterns werden der Fabrikmethode Parameter zugewiesen, damit diese unterschiedliche Objekte erzeugen kann.
   Diese Variante ist bei der *PaL*-Implementation so nicht vorgesehen. Sie lässt sich aber leicht nachbilden, indem man die Parameter der Komponente `CONCRETE_CREATOR` als Attribute zuweist und die Methode `factory_method` so überschreibt, dass die Attribute in die Erzeugung eines konkreten Produktes mit einbezogen werden.

Code

```
pattern PFACTORY_METHOD

    component PRODUCT
    end – component PRODUCT

    component CONCRETE_PRODUCT

        inherit
            PRODUCT

        creation make_concrete_product

        feature make_concrete_product is
            do
            end

    end – component CONCRETE_PRODUCT

    component CREATOR

        feature factory_method: PRODUCT is
            deferred
            end

    end – component CREATOR

    component CONCRETE_CREATOR

        inherit
            CREATOR

        feature factory_method: PRODUCT is

            local
                a_concrete_product: CONCRETE_PRODUCT

            do
                !!a_concrete_product.make_concrete_product;
                result := a_concrete_product
            end – factory_method

    end – component CONCRETE_CREATOR

end – pattern PFACTORY_METHOD
```

Konfigurationsmöglichkeiten

Bei der üblichen Anwendung dieses Design Patterns existiert für jedes konkrete Produkt ein konkreter Erzeuger. Die Verfeinerung für jeweils zwei konkrete Komponenten sieht z.B. wie folgt aus:

```
pattern PFACTORY_METHOD2

    refine

        select PFACTORY_METHOD
            rename
                CONCRETE_PRODUCT as CONCRETE_PRODUCT_A,
                CONCRETE_CREATOR as CONCRETE_CREATOR_A
        end

        PFACTORY_METHOD
            rename
                CONCRETE_PRODUCT as CONCRETE_PRODUCT_B,
                CONCRETE_CREATOR as CONCRETE_CREATOR_B
        end

end – pattern PFACTORY_METHOD2
```

Eine weitere denkbare Konfiguration ist, wenn die Komponente `CONCRETE_CREATOR` unterschiedliche konkrete Produkte erzeugen kann. In diesem Fall muss die Fabrikmethode entsprechend der Anzahl der konkreten Produkte dupliziert werden.

```
pattern PFACTORY_METHOD2

    refine

        select PFACTORY_METHOD <Ref_A>
            rename
                CONCRETE_PRODUCT as CONCRETE_PRODUCT_A
        end

        PFACTORY_METHOD <Ref_B>
            rename
                CONCRETE_PRODUCT as CONCRETE_PRODUCT_B
        end

    component CONCRETE_CREATOR

        cast
            rename
                factory_method from <Ref_A> as factory_method_a,
                factory_method from <Ref_B> as factory_method_b
        end

    end – component CONCRETE_CREATOR

end – pattern PFACTORY_METHOD2
```

Eine noch komplexere Variante wird mit dem Design Pattern Abstrakte Fabrik (20) beschrieben. Dort wird der Erzeuger zu einer Farik.

Kombination mit anderen Design Patterns

Das Design Pattern Abstrakte Fabrik (20) basiert vollständig auf diesem Design Pattern. Wenn man es so verallgemeinert, wie es für die Implementation in *PaL* erforderlich ist, gleicht der Quelltext dem dieses Patterns.

Das Pattern Fabrikmethode ist in dieser Bibliothek ebenfalls Grundlage für die Implementation des Patterns Iterator (24). Es tritt folglich überall dort auf, wo das Pattern Iterator eingesetzt oder mit anderen Design Patterns kombiniert wird.

Das Pattern Fabrikmethode ist grundsätzlich mit allen Design Patterns kombinierbar, in denen es abstrakte und dazugehörige konkrete Komponenten gibt. Diese zweistufigen Komponentenstrukturen können dann mit den Erzeuger- oder den Produkt-Komponenten kombiniert werden.

Anwendung in *DrawIt*

Da die Fabrikmethode im Design Pattern Iterator (24) enthalten ist, kommt es in *DrawIt* überall dort zum Einsatz, wo Container (11) verwendet werden.

In der Implementation von *DrawIt* verbindet dieses Pattern die Design Patterns Befehl (55) und Besucher (44). Es erzeugt zu einem Nutzer-Kommando den entsprechenden Besucher, der die Objektstruktur des Dokuments bearbeitet.

2.4.2 Abstrakte Fabrik (Abstract Factory)

Struktur

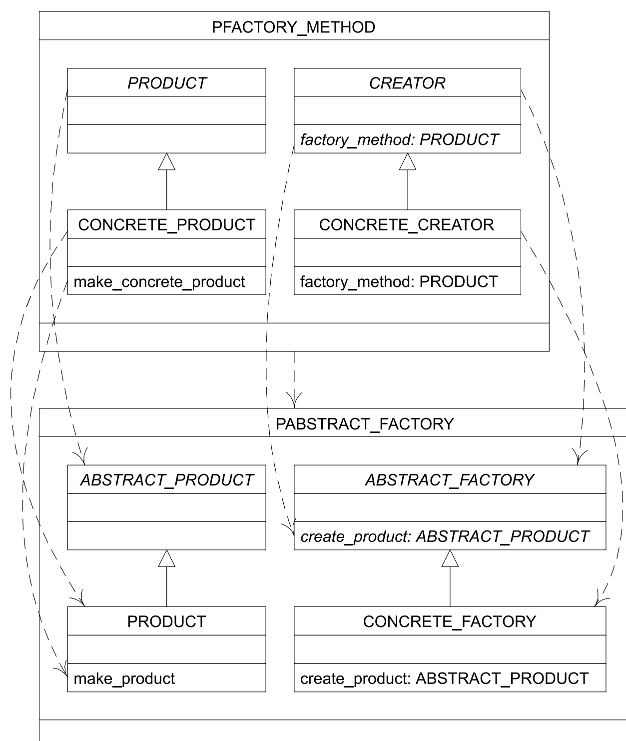


Abbildung 2.6: *Verfeinerung des Patterns Fabrikmethode zur Abstrakten Fabrik*

Implementierung

Wenn man von der Benennung der Komponenten und Features absieht, ist dieses Design Pattern mit der Fabrikmethode (17) identisch. Bei dem Vergleich dieser beiden Muster fällt auf, dass in [3] der Umgang mit Präfixen wie `ABSTRACT` und `CONCRETE` bei der Komponentenbezeichnung inkonsistent umgegangen wird.
Da die Abstrakte Fabrik als komplexe Variante der Fabrikmethode (17) betrachtet werden kann, wird das Pattern Abstrakte Fabrik durch Verfeinerung des Patterns Fabrikmethode (17) implementiert.

In [3] werden die folgenden Varianten der Implementation besprochen:

1. *Fabriken als Singletons.* Durch Kombination mit dem Design Pattern Singleton (69) kann auch in *PaL* eine Fabrik die einzige Instanz sein.

2. *Erzeugen von Produkten.* Hier werden die Varianten zur Erzeugung der Produkte mittels Fabrikmethode (17) und Prototyp (67) vorgestellt. Hier wurde die in [3] ausführlicher dokumentierte Variante der Fabrikmethode (17) implemeniert.

3. *Definieren von erweiterbaren Fabriken.* Hier werden Ansätze zur Erweiterung der Produktfamilie diskutiert. Das Mittel der Verfeinerung vereinheitlicht die Erweiterung des Design Patterns (siehe Konfigurationsmöglichkeiten).

Code

```
pattern PABSTRACT_FACTORY

    refine
        PFACTORY_METHOD
            rename
                CREATOR as ABSTRACT_FACTORY,
                CONCRETE_CREATOR as CONCRETE_FACTORY,
                PRODUCT as ABSTRACT_PRODUKT,
                CONCRETE_PRODUCT as PRODUCT
        end

    component ABSTRACT_FACTORY
        cast
            rename
                factory_method as create_product
        end
    end – component ABSTRACT_FACTORY

    component PRODUCT
        cast
            rename
                make_concrete_product as make_product
        end
    end – component PRODUCT

end – pattern PABSTRACT_FACTORY
```

Konfigurationsmöglichkeiten

Für die Erweiterung dieses Patterns gibt es zwei orthogonale Ansätze. Zum Einen kann die Produktfamilie erweitert werden (mehr abstrakte Produkte), zum Anderen können mehr konkrete Produktfamilien hinzugefügt werden (jeweils mehr Produkte zu den abstrakten Produkten). Im letzteren Fall werden entsprechend viele konkrete Fabriken hinzugefügt.
Da diese beiden Erweiterungen voneinander unabhängig sind, sollten sie in zwei unterschiedlichen Verfeinerungsschritten durchgeführt werden. In dem folgenden Beispiel werden zuerst mehr konkrete Fabriken und dann mehr abstrakte Produkte geschaffen.

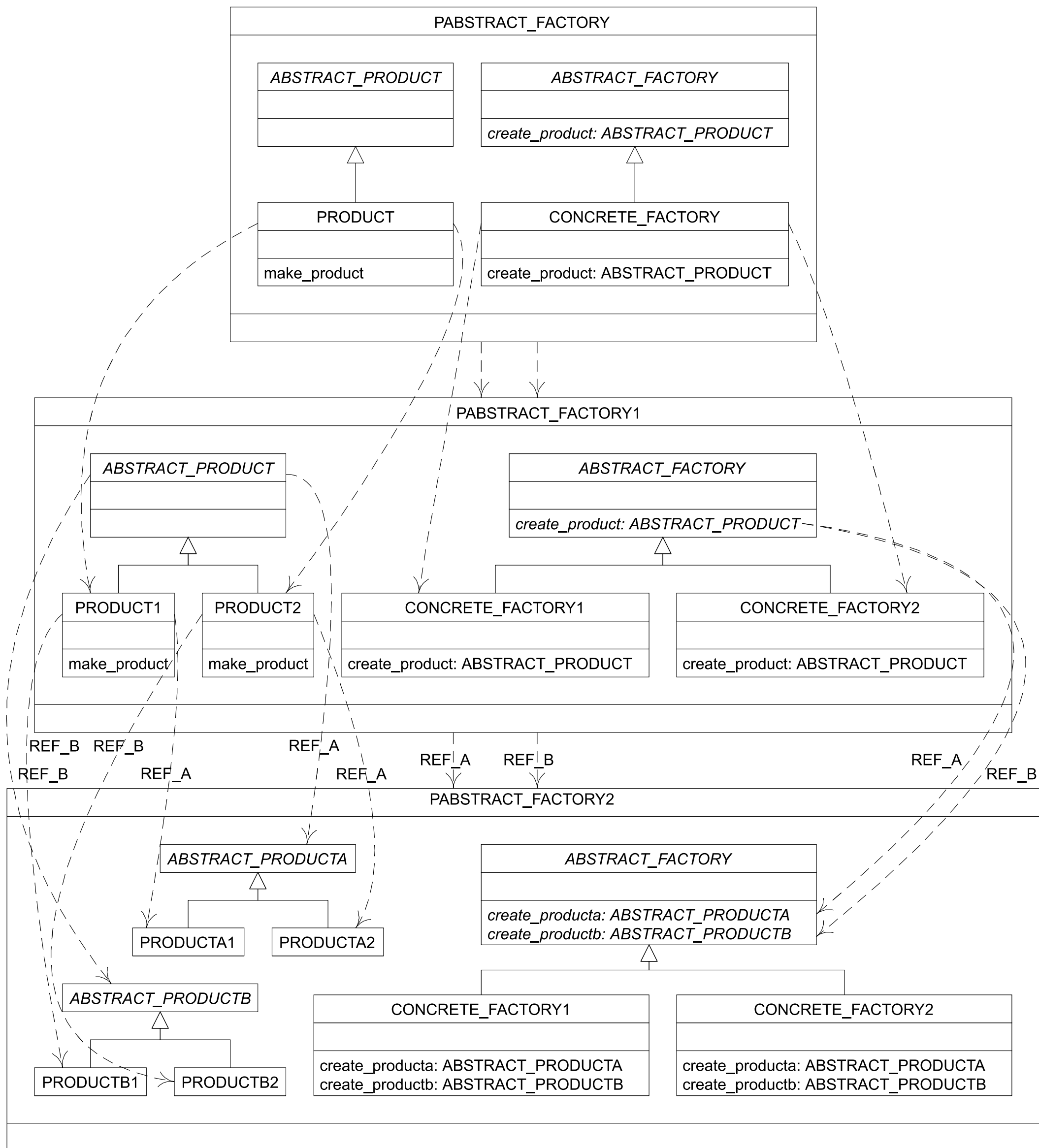


Abbildung 2.7: *Anwendung des Patterns Fabrikmethode*

```
pattern PABSTRACT_FACTORY1

    refine
        sellect PABSTRACT_FACTORY
            rename
                CONCRETE_FACTORY as CONCRETE_FACTORY1,
                PRODUCT as PRODUKT1
        end

        PABSTRACT_FACTORY
            rename
                CONCRETE_FACTORY as CONCRETE_FACTORY2,
                PRODUCT as PRODUKT2
        end

end – pattern PABSTRACT_FACTORY1

pattern PABSTRACT_FACTORY2

    refine
        sellect PABSTRACT_FACTORY1 <Ref_A>
            rename
                ABSTRACT_PRODUCT as ABSTRACT_PRODUCTA,
                PRODUCT1 as PRODUKTA1,
                PRODUCT2 as PRODUKTA2
        end

        PABSTRACT_FACTORY1 <Ref_B>
            rename
                ABSTRACT_PRODUCT as ABSTRACT_PRODUCTB,
                PRODUCT1 as PRODUKTB1,
                PRODUCT2 as PRODUKTB2
        end

    component ABSTRACT_FACTORY
        cast
            rename
                create_product from <Ref_A> as create_producta,
                create_product from <Ref_B> as create_productb
        end
    end – component ABSTRACT_FACTORY

end – pattern PABSTRACT_FACTORY2
```

Kombination mit anderen Design Patterns

Die allgemeine Implementation dieses Patterns ist genauso mit anderen Design Patterns kombinierbar, wie die Fabrikmethode (17). Bei vollständiger Ausprägung mit mehreren Fabriken und einer großen Produktfamilie ist die Kombination mit anderen Design Patterns unüblich.

Anwendung in *DrawIt*

Dieses Pattern kommt in *DrawIt* nicht zur Anwendung, da es dort mehrere Produktfamilien nicht gibt.

2.4.3 Iterator (Iterator)

Struktur

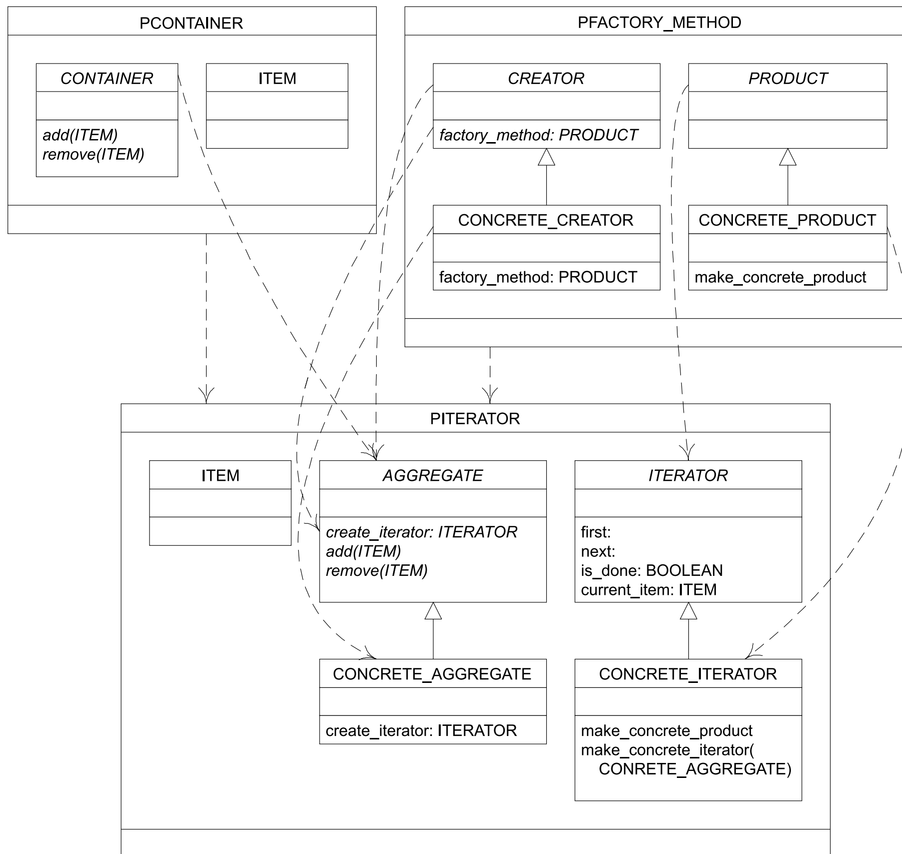


Abbildung 2.8: *Verfeinerung zum Design Pattern Iterator*

Implementierung

Der `Iterator` besitzt die Features `first`, `next`, und `is_done`, um damit durch den Inhalt eines Containers, hier durch die Komponente `AGGREGATE` repräsentiert, zu navigieren. Mit dem Feature `current_item` kann das aktuelle `ITEM` abgefragt werden. Da die Komponente `AGGREGATE` ein Container ist und der Typ `ITEM` im Pattern bekannt sein muss, wird dieses Design Pattern vom Pattern Container (11) verfeinert.

Das Feature `create_iterator` zur Erzeugung eines konkreten Iterators ist eine Fabrikmethode. Daher ist dieses Pattern eine Verfeinerung des Design Patterns Fabrikmethode (17). Da bei der Erzeugung eines Iterators ein Verweis auf das aktuelle Aggregat als Parameter übergeben wird, muss die Fabrikmethode `create_iterator` überschrieben werden.

In [3] werden die folgenden für die Standardimplementation in *PaL* wichtigen Aspekte angesprochen:

1. *Steuerung der Iteration.* Es werden externe und interne Iteratoren vorgestellt. Externe Iteratoren sind flexibler, interne Iteratoren sind in manchen Situationen jedoch einfacher zu handhaben. Der in dieser Bibliothek enthaltene Iterator ist grundsätzlich externer Natur. Er lässt sich aber durch Hinzufügen einer Traversierungsmethode zum Iterator auch zu einem internen Iterator verfeinern.

2. *Definition des Traversierungsalgorithmus.* Hier werden mögliche Stellen aufgeführt, an denen der Traversierungsalgorithmus definiert werden kann. Aufgrund der vielfältigen Möglichkeiten wird bei der Implementation in *PaL* noch kein Ort für den Algorithmus festgelegt.

3. *Robustheit des Iterators.* Da bei dieser Standardimplementation die Art des Aggregates noch nicht definiert ist, lässt sich noch nichts über die Robustheit des Iterators sagen. Aufgrund der vielfältigen Kombinations- und Verfeinerungsmöglichkeiten ist bei einer konkreten Implementation besonders auf Robustheit zu achten.

4. *Zusätzliche Iteratoroptionen.* Durch Verfeinerung ist es möglich, dem Iterator die in [3] vorgeschlagenen Erweiterungen hinzuzufügen.

Code

```
pattern PITERATOR

    refine
        PCONTAINER
            rename
                CONTAINER as AGGREGATE
        end

        PFACTORY_METHOD
            rename
                CREATOR as AGGREGATE,
                CONCRETE_CREATOR as CONCRETE_AGGREGATE,
                PRODUCT as ITERATOR,
                CONCRETE_PRODUCT as CONCRETE_ITERATOR
        end

    component AGGREGATE
        cast
            rename
                factory_method as create_iterator
        end
    end – component AGGREGATE

    component CONCRETE_AGGREGATE

        feature create_iterator: ITERATOR is
            local
                a_concrete_iterator: CONCRETE_ITERATOR
            do
                !!a_concrete_iterator.make_concrete_iterator(current);
                result := a_concrete_iterator
            end – create_iterator

    end – component CONCRETE_AGGREGATE

    component ITERATOR

        feature first is
            deferred
            end

        feature next is
            deferred
            end

        feature is_done: BOOLEAN is
            deferred
            end

        feature current_item: ITEM is
            deferred
            end

    end – component ITERATOR
```

```
    component CONCRETE_ITERATOR
        creation
            make_concrete_iterator
        feature make_concrete_iterator(a_concrete_aggregate: CONCRETE_AGGREGATE) is
            do
                the_aggregate := a_concrete_aggregate
            end
        feature the_aggregate: CONCRETE_AGGREGATE
    end – component CONCRETE_ITERATOR
end – pattern PITERATOR
```

Konfigurationsmöglichkeiten

Ähnlich wie das Pattern Fabrikmethode (17) lässt sich dieses Pattern auf zwei verschiedene Arten erweitern.
In der ersten Variante kann es unterschiedliche konkrete Aggregate geben, zu denen je ein konkreter Iterator existiert. So könnte ein konkretes Aggregat die Items in einer Liste verwalten und ein anderes in einem Array.
In der zweiten Variante existieren für ein konkretes Aggregat mehrere konkrete Iteratoren. So ist es denkbar, dass bei einem konkreten Iterator die Methode `next` nicht zum nächsten `ITEM` springt, sondern zum nächsten `ITEM` mit ganz bestimmten Eigenschaften.
In keinem Fall ist es möglich, dass der abstrakte Iterator für unterschiedliche Arten von Items definiert wird, da `ITEM` in der Schnittstelle von `ITERATOR` als Parameter vorkommt.

Kombination mit anderen Design Patterns

Das Pattern lässt sich hervorragend mit all den Design Patterns kombinieren, die Container-Funktionalität enthalten. Das sind die Design Patterns Kompositum (28), Fliegengewicht (33), Beobachter (60), Vermittler (63) und Erbauer (65). (Siehe auch Abschnitte 2.5.2 und 2.5.5.)

Eine konkrete Anwendung lässt sich z.B. durch die Kombination mit der Liste (12) implementieren. (Siehe auch Abschnitt 2.5.1.)

Anwendung in *DrawIt*

In *DrawIt* wird das Iterator-Pattern in jedem Fall mit der Liste (12) kombiniert. Es iteriert in der Anwendung über die gruppierten Grafiken, über die bei einer Grafik angemeldeten Verbinder und über die Befehlskette.

2.4.4 Kompositum (Composite)

Struktur

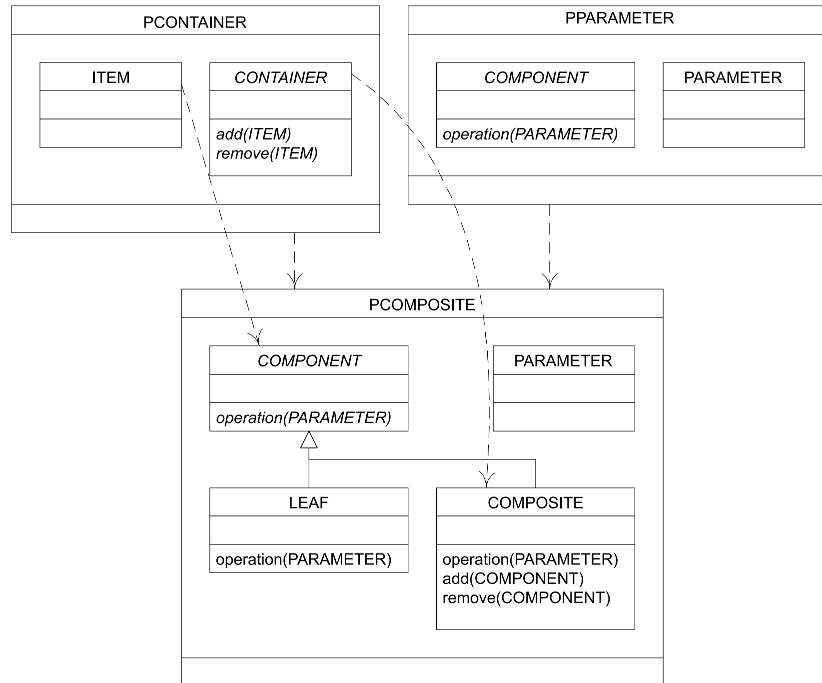


Abbildung 2.9: *Verfeinerung zum Design Pattern Kompositum*

Implementierung

Das Design Pattern Kompositum besteht aus den drei Komponenten `COMPONENT`, `COMPOSITE` und `LEAF`. Da `COMPOSITE` mehrere Objekte vom Typ `COMPONENT` verwalten kann, wurden diese beiden Komponenten vom Pattern Container (11) verfeinert.
Die Funktionalität der Struktur wird durch die Methode `operation` repräsentiert. Da bei der Standardimplementation des Patterns noch nicht bekannt ist, welcher Art die Funktionalität ist, die in einer konkreten Anwendung implementiert wird, lässt sich zu diesem Zeitpunkt noch keine Aussage über die Anzahl der Parameter treffen. Daher wird dieses Design Pattern von dem für solche Fälle implementierten Pattern Parameter (15) verfeinert.

Die Implementation ist noch so abstrakt, wie sie in [3] beschrieben wird. Die Art des Containers ist noch nicht festgelegt. Für einen konkreten Einsatz muss das Design Pattern noch mit einer konkreten Container-Implementation wie z.B. Liste (12) und evtl. mit einem Iterator (24) kombiniert werden.

In [3] werden für die Implementation des Kompositums die folgenden wichtigen Aspekte angesprochen:

1. *Explizite Referenzen auf das Elternobjekt.* Referenzen von Kindobjekten auf die Elternobjekte sind nicht standardmäßig implementiert. Dem Programmierer steht aber nichts im Wege, solche Referenzen einzuführen.

2. *Gemeinsame Nutzung von Komponenten.* Um die Speicheranforderungen zu senken, können Komponenten gemeinsam genutzt werden. Diese Variante wird durch das De-

sign Pattern Fliegengewicht (33) realisiert, das sich mit diesem Pattern kombinieren lässt.

3. *Maximierung der Komponentenschnittstelle.* Siehe 5.
4. *Deklarieren von Verwaltungsoperationen für Kindobjekte.* Siehe 5.
5. *Ort des Behälters für enthaltene Komponenten.* Diese drei Punkte behandeln unterschiedliche Varianten der Verlagerung der Container-Schnittstelle nach `COMPONENT`, um größere Transparenz zu erreichen. Aus Effizienz- und Sicherheitsgründen wurden diese Varianten hier nicht gewählt.
6. *Ordnung der Kindobjekte.* Um eine Ordnung auf den Kindobjekten zu schaffen, wird die Kombination mit dem Design Pattern Iterator (24) vorgeschlagen. Diese Kombination ist auch in der Sprache *PaL* sehr gut möglich.
7. *Verbessern des Laufzeitverhaltens durch Zwischenspeicherung (Caching).* Dies ist schon mehr ein Aspekt der konkreten Anwendung des Patterns. Diese Idee wurde in *DrawIt* realisiert. Eine gruppierte Grafik speichert das kleinste umschließende Rechteck (mbr) und muss es so nicht bei jeder Anfrage aus den Kindobjekten berechnen.
8. *Löschen der Komponente.* Das Löschen von Komponenten und Komposita ist in *PaL* kein Problem, da die Sprache von der Garbage Collection von Eiffel profitiert.
9. *Datenstrukturen zum Speichern von Komponenten.* Hier wird erläutert, dass das Pattern, wie auch in dieser Standardimplementation, mit vielen Arten von Containern funktioniert. In einer Variante werden aber die Kindkomponenten einzeln und direkt im Kompositum gespeichert. Die Variante funktioniert mit dieser Implementation nicht, dafür ist das Design Pattern Interpreter (31) vorgesehen.

Code

```
pattern PCOMPOSITE

    refine
        PCONTAINER
            rename
                CONTAINER as COMPOSITE,
                ITEM as COMPONENT
        end

        PPARAMETER
        end

    component COMPOSITE
        inherit
            COMPONENT
    end – component COMPOSITE

    component LEAF
        inherit
            COMPONENT
    end – component LEAF

end – pattern PCOMPISITE
```

Konfigurationsmöglichkeiten

Für dieses Design Pattern sind zwei orthogonale Möglichkeiten zur Erweiterung vorgesehen. Einerseits lassen sich mehr Operationen hinzufügen. Hierfür wird die Methode `operation` dupliziert. Andererseits können mehr Blätter hinzugefügt werden. In dem Fall wird die Komponente `LEAF` vervielfacht.
Da beide Erweiterungen unabhängig voneinander sind, lassen sie sich am besten in zwei Verfeinerungsschritten durchführen.

```
pattern PCOMPOSITE1

    refine
        select PCOMPOSITE <Ref_1>
        end

        PCOMPOSITE <Ref_2>
        end

    component COMPONENT
        cast
            rename
                operation from <Ref_1> as operation1
                operation from <Ref_2> as operation2
        end
    end – component COMPONENT

end – pattern PCOMPISITE1

pattern PCOMPOSITE2

    refine
        select PCOMPOSITE1
            rename
                LEAF as LEAFA
        end

        PCOMPOSITE1
            rename
                LEAF as LEAFB
        end

end – pattern PCOMPISITE2
```

Kombination mit anderen Design Patterns

Um dem Pattern mehr konkrete Funktionalität zu geben, kann es mit den Patterns Liste (12) und Iterator (24) kombiniert werden. (Siehe auch Abschnitte 2.5.2 und 2.5.3).
Von der Sruktur her ist es gut mit den Design Patterns Dekorierer (36) und Fliegengewicht (33) kombinierbar. (Siehe auch Abschnitt 2.5.4.)
Die Funktionalität der Struktur lässt sich hervorragend durch die Kombination mit dem Design Pattern Besucher (44) auslagern.
In [3] wird zusätzlich die Kombination mit dem Design Pattern Zuständigkeitskette (42) vorgeschlagen, um Verantwortlichkeiten von den Kindobjekten zum Kompositum weiterzuleiten. Auch diese Kombination lässt sich in *PaL* problemlos realisieren.

Anwendung in *DrawIt*

In *DrawIt* werden die gruppierten Grafiken mit Hilfe dieses Patterns realisiert.

2.4.5 Interpreter (Interpreter)

Struktur

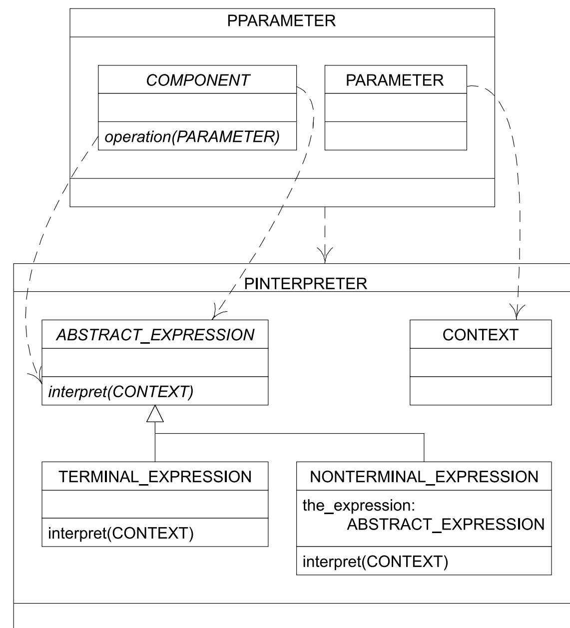


Abbildung 2.10: *Verfeinerung zum Design Pattern Interpreter*

Implementierung

Die abstrakte Form des Interpreter-Patterns wird ähnlich dem Kompositum (28) implementiert. Es besteht aus den Komponenten `ABSTRACT_EXPRESSION`, `NONTERMINAL_EXPRESSION` und `TERMINAL_EXPRESSION`. Diese Komponenten enthalten die Methode `interpret` mit dem Parameter `CONTEXT`. Die Komponente `NONTERMINAL_EXPRESSION` verwendet jedoch keinen Container (11), sondern direkte Referenzen auf die untergeordneten Objekte.

Von der Diskussion der Implementierung in [3] sind die Punkte 2. und 3. für die Implementation in *PaL* interessant:

2. *Definieren der Interpretiere-Operation.* Hier wird als Alternative zur Methode `interpret` die Kombination mit dem Design Pattern Besucher (44) vorgeschlagen. Diese Kombination ist mit dieser Standardimplementation in *PaL* ebenfalls möglich.

3. *Gemeinsames Nutzen von Symbolen mittels des Fliegengewichtmusters.* Für häufig verwendete Terminale wird die Anwendung des Design Patterns Fliegengewicht (33) vorgeschlagen. In *PaL* steht dieser Kombination nichts im Wege.

Code

```
pattern PINTERPRETER

    refine
        PPARAMETER
            rename
                COMPONENT as ABSTRACT_EXPRESSION,
                PARAMETER as CONTEXT
        end

    component ABSTRACT_EXPRESSION
        cast
            rename
                operation as interpret
        end
    end – component ABSTRACT_EXPRESSION

    component NONTERMINAL_EXPRESSION

        inherit
            ABSTRACT_EXPRESSION

        feature the_expression: ABSTRACT_EXPRESSION

    end – component NONTERMINAL_EXPRESSION

    component TERMINAL_EXPRESSION
        inherit
            ABSTRACT_EXPRESSION
    end – component TERMINAL_EXPRESSION

end – pattern PINTERPRETER
```

Konfigurationsmöglichkeiten

Theoretisch ist dieses Pattern durch Vervielfältigung der Komponente `NONTERMINAL_EXPRESSION` und deren Attribut `the_expression` auf alle Anwendungsmöglichkeiten des Patterns konfigurierbar. In der Praxis ist die Anwendung dieses Patterns jedoch sehr umständlich, da die Anzahl der Verfeinerungen mit der Anzahl und vor allem der Komplexität der Ausdrücke zunimmt. Die Wiederverwendung ist in diesem Fall deutlich aufwendiger als die Neuimplementation. Der Einsatz dieses Patterns mittels Verfeinerung ist nur bei wenigen, sehr einfach aufgebauten Ausdrücken zu empfehlen.

Kombination mit anderen Design Patterns

Wie schon bei der Implementierung angesprochen, ist dieses Pattern mit den Design Patterns Besucher (44) und Fliegengewicht (33) kombinierbar. Eine Verknüpfung mit dem Design Pattern Kompositum (28) ist ebenfalls möglich.
In [3] wird zusätzlich die Kombination mit dem Pattern Iterator (24) vorgeschlagen, um über die Kindobjekte der Komponente `NONTERMINAL_EXPRESSION` zu iterieren. Da die Verwendung dieser Patternimplementation nur bei sehr wenigen Kindobjekten sinnvoll ist, bringt die Kombination mit dem Iterator (24) in *PaL* keine Vorteile.

Anwendung in *DrawIt*

Das Pattern wird in *DrawIt* nicht angewendet. Der Einsatz wäre aber denkbar, um eine als Textdatei abgespeicherte Befehlskette zu interpretieren.

2.4.6 Fliegengewicht (Flyweight)

Struktur

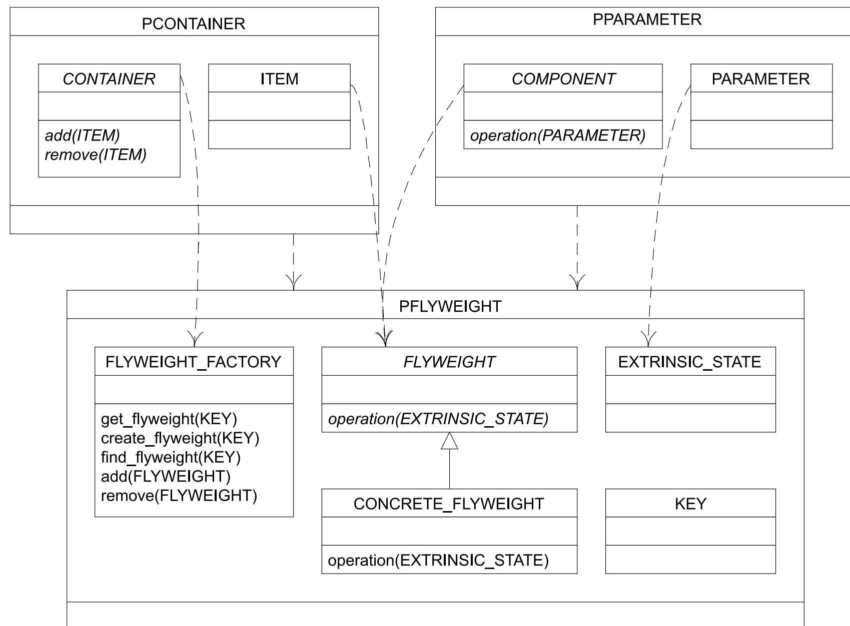


Abbildung 2.11: *Verfeinerung zum Design Pattern Fliegengewicht*

Implementierung

Das Pattern wird durch Verfeinerung des Hilfspatterns Container (11) implementiert, denn die Komponente `FLYWEIGHT_FACTORY` ist nicht nur Erzeuger, sondern auch Sammelbehälter für die Fliegengewichte.
Die abstrakte Komponente `FLYWEIGHT` enthält eine Methode `operation` über deren Parameter nichts weiter bekannt ist, als dass er den externen Zustand des Fliegengewichtes darstellt. Wie immer in solch einer Situation wird von dem Hilfspattern Parameter (15) verfeinert.
Die Komponente `KEY` ist der Typ des Schlüssels, mit dessen Hilfe die Fliegengewichte gesucht und erzeugt werden.
In dieser Standardimplementation wird noch nicht wie in [3] zwischen gemeinsam genutzten und getrennt genutzten Fliegengewichten unterschieden. Zum Einen ist noch nicht klar, ob es überhaupt getrennt genutzte Fliegengewichte geben wird, zum Anderen ist die Standardimplementation noch so abstrakt, dass es zwischen den beiden Typen noch keine bedeutenden Unterschiede gibt.

Die beiden folgenden Aspekte der Implementation werden in [3] angesprochen:

1. *Entfernen von extrinsischem Zustand.* Hier wird darauf hingewiesen, dass die Vorteile dieses Patterns dann am größten sind, wenn der extrinsische Zustand möglichst groß und einheitlich ist. Hier wird dieser Zustand in Objekten der Komponente `EXTRINSIC_STATE` festgehalten.

2. *Verwaltung gemeinsam genutzter Objekte.* Es wird empfohlen, dass die Komponente `FLYWEIGHT_FACTORY` die Fliegengewichte verwaltet. Diese Empfehlung ist bereits in die Standardimplementation eingeflossen. Die Verwaltung der Fliegengewichte wird durch den Container (11) angedeutet.

Code

```
pattern PFLYWEIGHT

    refine
        PCONTAINER
            rename
                CONTAINER as FLYWEIGHT_FACTORY,
                ITEM as FLYWEIGHT
        end

        PPARAMETER
            rename
                COMPONENT as FLYWEIGHT,
                PARAMETER as EXTRINSIC_STATE
        end

    component CONCRETE_FLYWEIGHT
        inherit
            FLYWEIGHT
    end – component CONCRETE_FLYWEIGHT

    component FLYWEIGHT_FACTORY

        feature get_flyweight(a_key: KEY): FLYWEIGHT is
            local
                temp_flyweight: CONCRETE_FLYWEIGHT
            do
                result := find_flyweight(a_key);
                if (result = void)
                then
                    result := create_flyweight(a_key)
                end
            end – get_flyweight

        feature create_flyweight(a_key: KEY): FLYWEIGHT is
            local
                temp_flyweight: CONCRETE_FLYWEIGHT
            do
                !!temp_flyweight.make(a_key);
                add(temp_flyweight);
                result := temp_flyweight
            end – create_flyweight

        feature find_flyweight(a_key: KEY): FLYWEIGHT is
            deferred
            end – find_flyweight

    end – component FLYWEIGHT_FACTORY

    component KEY
    end – component KEY

end – pattern PFLYWEIGHT
```

Konfigurationsmöglichkeiten

Die Konfigurationsmöglichkeiten gleichen denen des Design Patterns Kompositum (28). Auf die dort beschriebene Art lassen sich mehr konkrete Fliegengewichte und mehr Operationen erzeugen.
Zusätzlich lassen sich durch Vervielfältigung der Komponente `CONCRETE_FLYWEIGHT` auch getrennt genutzte Fliegengewichte erzeugen. In diesem Fall wird die Verwaltung aller Fliegengewichte komplexer. Soll es nur getrennt genutzte Fliegengewichte geben, ist das Design Pattern Kompositum (28) vorzuziehen.

Kombination mit anderen Design Patterns

Dieses Design Pattern ist mit dem Pattern Kompositum (28) verwandt. Ein Kompositum kann seine Kindobjekte mit Hilfe von Fliegengewichten verwalten. Es speichert dafür die

Schlüssel der Fliegengewichte im Container (11). Der extrinsische Zustand wird vom Kompositum berechnet oder bei den Schlüsseln gespeichert.
Die Verwaltung der Fliegengewichte in der Komponente `FLYWEIGHT_FACTORY` lässt sich durch die Kombination mit den Patterns Iterator (24) und Liste (12) konkretisieren.
Durch Kombination mit den Design Patterns Zustand (50) und Strategie (47) kann dieses Pattern effizient Zustände und Strategie-Objekte verwalten.

Anwendung in *DrawIt*

Fliegengewichte kommen in *DrawIt* nicht zum Einsatz. Sie könnten zur Speicherung von „knappen Gütern", wie *Pens*, *Brushes* und *Fonts* verwendet werden.

2.4.7 Dekorierer (Decorator)

Struktur

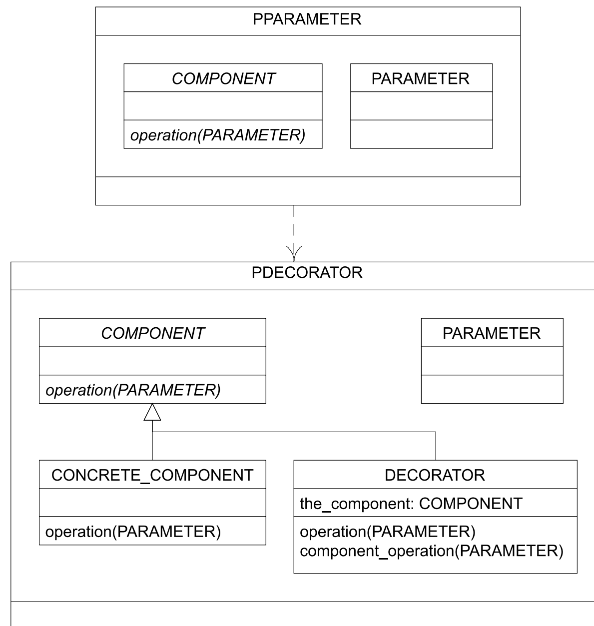


Abbildung 2.12: *Verfeinerung zum Design Pattern Dekorierer*

Implementierung

Wie schon durch die Abbildung 2.12 deutlich wird, weicht die Implementation des Dekorierer-Patterns in dieser Standardbibliothek von der Beschreibung in [3] ab. In der Implementation in [3] gibt es einen abstrakten Dekorierer, der eine Komponente verwalten kann und mehrere konkrete Dekorierer, die davon erben, um diese Implementation wiederzuverwenden. Der abstrakte Dekorierer bildet eine einheitliche Schnittstelle für die konkreten Dekorierer.

Durch die Möglichkeiten der Sprache *PaL* lässt sich dieses Pattern auch auf einfachere Art implementieren. Dabei werden die konkreten und abstrakten Dekorierer zu einer Komponente zusammengefasst. Die einheitliche Schnittstelle des abstrakten Dekorierers wird in einer konkreten Anwendung im Allgemeinen nicht gebraucht, hier genügt die Schnittstelle von `COMPONENT`. Die Wiederverwendung der Struktur und der Funktionalität des Dekorierers geschieht dann nicht durch Vererbung, sondern durch Mehrfachverfeinerung. Der Vorteil dieser Variante ist, dass die Struktur nicht dreistufig, sondern nur zweistufig ist. Somit lässt sich das Pattern leichter mit anderen Patterns mit zweistufiger Struktur kombinieren.

In [3] wird diese Variante ebenfalls diskutiert:

1. *Schnittstellenkonformanz.* Hier wird nur gefordert, dass der Dekorierer und die konkrete Komponente von einer gemeinsamen Klasse erben.

2. *Weglassen der abstrakten Dekoriererklasse.* In diesem Punkt wird genau die in dieser Bibliothek implementierte Variante besprochen. Bei objektorientierter Programmierung ist diese Variante nur bei genau einer Dekorierer-Klasse sinnvoll. Bei patternorientierter Programmierung ist dieser Ansatz generell der bessere.

Code

```
pattern PDECORATOR

    refine
        PPARAMETER
        end

    component CONCRETE_COMPONENT
        inherit
            COMPONENT
    end – component CONCRETE_COMPONENT

    component DECORATOR

        inherit
            COMPONENT

        creation
            make_decorator

        feature the_component: COMPONENT

        feature make_decorator(a_component: COMPONENT) is
            do
                the_component := a_component
            end

        feature operation(a_parameter: PARAMETER) is
            do
                component_operation(a_parameter)
            end

        feature component_operation(a_parameter: PARAMETER) is
            do
                the_component.operation(a_parameter)
            end

    end – component DECORATOR

end – pattern PDECORATOR
```

Konfigurationsmöglichkeiten

Zum Einen lassen sich die konkreten Komponenten vervielfältigen. Weiterhin kann man durch das Duplizieren der Operation mehr Funktionalität hinzufügen. Diese Erweiterungen gleichen den Erweiterungen, die beim Kompositum (28) vorgeschlagen wurden.
Zusätzlich kann man mehrere konkrete Dekorierer anlegen, indem man die Komponente `DECORATOR` durch Mehrfachverfeinerung vervielfältigt.
Diese drei Varianten der Erweiterung sind unabhängig voneinander und sollten in unterschiedlichen Verfeinerungsstufen durchgeführt werden. Durch das folgende Codefragment wird lediglich das Anlegen zweier konkreter Dekorierer angedeutet. Hiermit wird der gleiche Effekt, wie mit der in [3] beschriebenen Implementation erzielt.

```
pattern PDECORATOR2

    refine
        select PDECORATOR
            rename
                DECORATOR as CONCRETE_DECORATORA
        end

        PDECORATOR
            rename
                DECORATOR as CONCRETE_DECORATORB
        end

end – pattern PDECORATOR2
```

Kombination mit anderen Design Patterns

Durch die Ähnlichkeit der Strukturen lässt sich dieses Pattern hervorragend mit dem Kompositum (28) kombinieren. (Siehe auch Abschnitt 2.5.4.)
Wie beim Kompositum (28) lässt sich die Funktionalität durch die Kombination mit dem Design Pattern Besucher (44) auslagern. Diese Kombination gestaltet sich durch das Weglassen des abstrakten Dekorierers deutlich einfacher, da nun eine zweistufige Hierarchie mit einer anderen zweistufigen Hierarchie verknüpft wird.

Anwendung in *DrawIt*

In *DrawIt* kommen zwei unterschiedliche Dekorierer von Grafiken zum Einsatz. Zum Einen gibt es *Markierer*, die dazu dienen, die Markierung der Grafiken darzustellen, zum Anderen gibt es *Verbinder*, die eine Grafik so verändern, dass sie zum Verbinder zwischen anderen Grafiken wird.

2.4.8 Proxy (Proxy)

Struktur

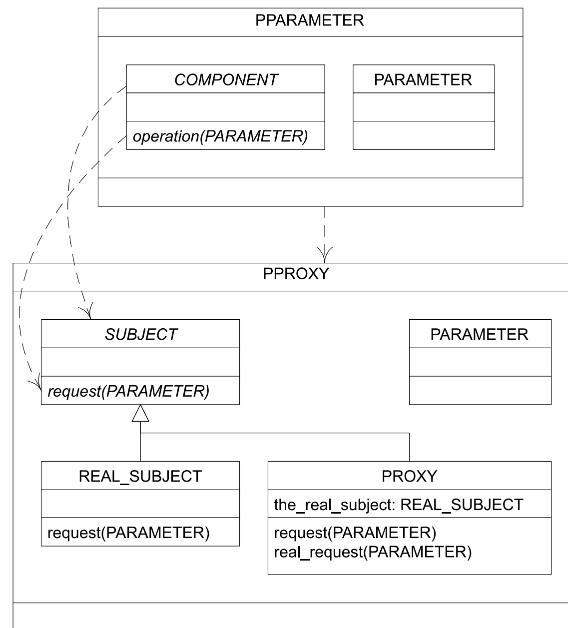


Abbildung 2.13: *Verfeinerung zum Design Pattern Proxy*

Implementierung

Das Pattern besteht aus den Komponenten `SUBJECT`, `REAL_SUBJECT` und `PROXY`. Die Methode `request` der Komponenten wird wieder durch das Hilfspattern Parameter (15) parametrisiert.
Zur Delegation der Anfrage wird eine weitere Methode `real_request` im `PROXY` implementiert. Das hat den Vorteil, dass die Methode `request` im `PROXY` überschrieben werden kann, ohne dass die Implementation der Delegation verloren geht.

Von den in [3] besprochenen Implementationsaspekten ist nur der Punkt 3. für die Sprache *PaL* interessant:

3. *Der Proxy muss nicht immer den dynamischen Typ des eigentlichen Subjekts kennen.* Hier wird die Variante vorgestellt, in der der `PROXY` nicht einen Verweis auf die Komponente `REAL_SUBJECT` hat, sondern auf die allgemeinere Oberkomponente `SUBJECT`. Die Implementation würde dann der des Design Patterns Dekorierer (36) gleichen. In *PaL* muss man sich bereits bei der Standardimplementation des Patterns für eine Variante entscheiden. Diese Variante wurde hier nicht gewählt und ist auch nicht durch Verfeinerung nachträglich realisierbar.

Code

```
pattern PPROXY

    refine
        PPARAMETER
            rename
                COMPONENT as SUBJECT
        end

    component SUBJECT
        cast
            rename
            operation as request
        end
    end – component SUBJECT

    component REAL_SUBJECT
        inherit
            SUBJECT
    end – component REAL_SUBJECT

    component PROXY

        inherit
            SUBJECT

        feature the_real_subject: REAL_SUBJECT

        feature request(a_parameter: PARAMETER) is
            do
                real_request(a_parameter)
            end – request

        feature real_request(a_parameter: PARAMETER) is
            do
                the_real_subject.request(a_parameter)
            end – real_request

    end – component PROXY

end – pattern PPROXY
```

Konfigurationsmöglichkeiten

Bei dieser Implementation lassen sich mehrere Proxys, aber nicht mehrere reale Subjekte hinzufügen. Zur Erweiterung des Patterns um einen weiteren Proxy wird von dem Pattern zweimal verfeinert. Die Komponente `PROXY` wird dabei jeweils so umbenannt, dass die beiden gewünschten Proxys entstehen. In der einen Verfeinerung wird die Komponente `REAL_SUBJECT` so umbenannt, dass sie auf den Proxy der anderen Verfeinerung abgebildet wird. Auf diese Weise entsteht eine Kette von Proxys.

```
pattern PPROXY2

    refine
        select PPROXY
            rename
                PROXY as PROXY1
        end

        PPROXY
            rename
                REAL_SUBJECT as PROXY1
                PROXY as PROXY2
        end

end – pattern PPROXY2
```

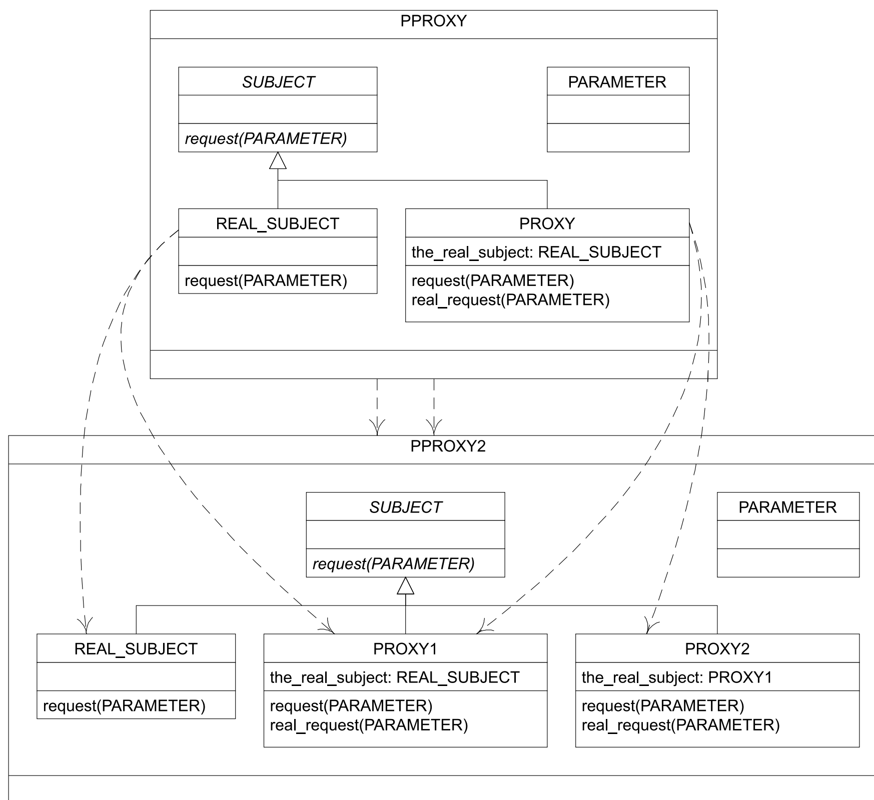


Abbildung 2.14: *Erweiterung des Design Patterns Proxy*

Kombination mit anderen Design Patterns

Das Pattern ist gut mit anderen zweistufig strukturierten Patterns kombinierbar.

Anwendung in *DrawIt*

In *DrawIt* kommt dieses Pattern nicht zur Anwendung.

2.4.9 Zuständigkeitskette (Chain of Responsibility)

Struktur

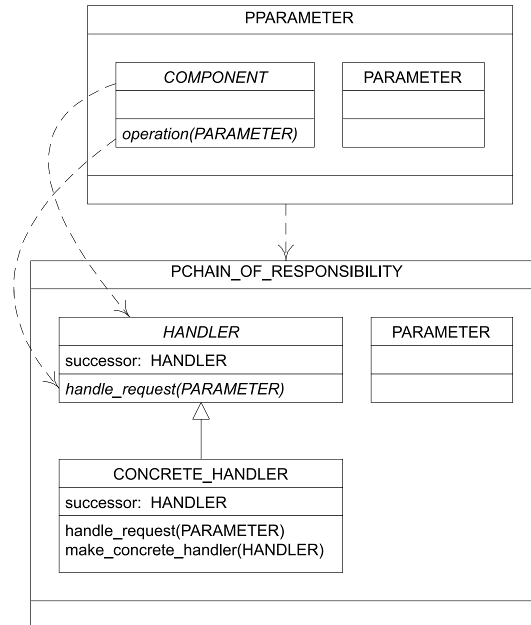


Abbildung 2.15: *Verfeinerung zum Design Patterns Zuständigkeitskette*

Implementierung

Dieses ist ein sehr einfaches Pattern. Es besteht aus den Komponenten `HANDLER` und `CONCRETE_HANDLER`. Da über die Methode `handle_request` noch nichts bekannt ist, wird sie durch Verfeinerung des Patterns Parameter (15) implementiert. In der Komponente `CONCRETE_HANDLER` bekommt die Methode `handle_request` die Standardimplementation, so dass alle Aufrufe an den Nachfolger weiterleitet werden. Für diesen Fall muss die Methode nicht überschrieben werden.

Die folgenden Aspekte werden in [3] behandelt:

1. *Implementierung der Nachfolgerkette.* Hier wird besprochen, in welchen Fällen die Nachfolgerkette neu implementiert werden muss und wann auf vorhandene Strukturen zurückgegriffen werden kann. Bei dieser Standardimplementation ist die Nachfolgerkette bereits implementiert. Man kann nur auf vorhandene Strukturen zurückgreifen, wenn sich die implementierte Nachfolgerkette durch Verfeinerung genau auf die vorhandene Struktur abbilden lässt.

2. *Verbinden von Nachfolgeobjekten.* Hier wird die Implementation der Methode `handle_request` vorgeschlagen, die standardmäßig alle Anfragen an den Nachfolger weiterleitet. Diese Variante wurde in der Bibliothek implementiert.

3. *Repräsentation von Anfragen.* In diesem Punkt wird die Repräsentation der Anfragen durch Objekte empfohlen, wie es hier durch die Komponente `PARAMETER` gelöst ist.

Code

```
pattern PCHAIN_OF_RESPONSIBILITY

    refine
        PPARAMETER
            rename
                COMPONENT as HANDLER
        end

    component HANDLER

        cast
            rename
                operation as handle_reqeuset
        end

        feature successor: HANDLER

    end – component HANDLER

    component CONCRETE_HANDLER

        inherit
            HANDLER

        creation
            make_concrete_handler

        feature handle_reqeuset(a_parameter: PARAMETER) is
            do
                if (successor /= void) then
                    successor.handle_reqeuset(a_parameter)
                end
            end

        feature make_concrete_handler(a_handler: HANDLER) is
            do
                successor := a_handler
            end

    end – component CONCRETE_HANDLER

end – pattern PCHAIN_OF_RESPONSIBILITY
```

Konfigurationsmöglichkeiten

Dieses Design Pattern erweitert man, indem man `CONCRETE_HANDLER` dupliziert.

Kombination mit anderen Design Patterns

Wenn die Zuständigkeitskette mit dem Pattern Kompositum (28) kombiniert wird, lassen sich so Anfragen in der Struktur nach oben weiterleiten.
Die Komponente `HANDLER` lässt sich gut mit den Items im Container (11) verknüpfen.

Anwendung in *DrawIt*

Dieses Design Pattern wird in *DrawIt* nicht verwendet.

2.4.10 Besucher (Visitor)

Struktur

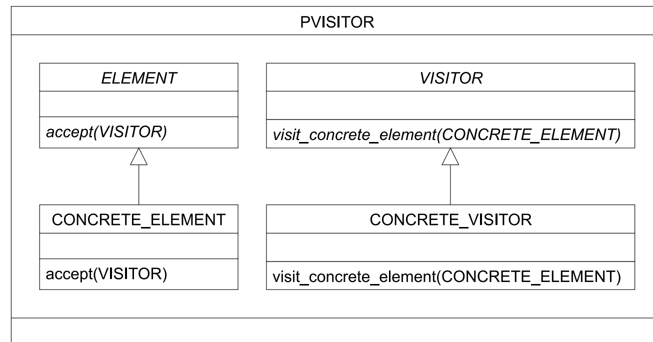


Abbildung 2.16: *Die Struktur des Design Patterns Besucher*

Implementierung

Das Besucher-Pattern wird ohne Verfeinerung anderer Patterns implementiert. Es besteht in der abstrakten Form aus den vier Komponenten `VISITOR`, `CONCRETE_VISITOR`, `ELEMENT` und `CONCRETE_ELEMENT`.

Die folgenden Implementierungsaspekte werden in [3] angesprochen:

1. *Double-Dispatch.* Eiffel und somit auch *PaL* bieten bieten von der Sprache her lediglich Single-Dispatch. Das Besucher-Pattern beschreibt, wie man Double-Dispatch erreichen kann.

2. *Zuständigkeit für die Traversierung.* Hier wird beschrieben, welche Möglichkeiten es gibt, dass der Besucher jedes konkrete Element besucht. Dafür kann der Besucher zuständig sein oder die Objektstruktur. Das Pattern Iterator (24) kann dabei helfen. In dieser Standardimplementation ist noch nichts über die Traversierung der Elemente festgelegt, mit ihr sind alle aufgezählten Varianten realisierbar.

Code

```
pattern PVISITOR

    component VISITOR
        feature visit_concrete_element(an_element: CONCRETE_ELEMENT) is
            deferred
            end – visit_concrete_element
    end – component VISITOR

    component CONCRETE_VISITOR
        inherit
            VISITOR
    end – component CONCRETE_VISITOR

    component ELEMENT
        feature accept(a_visitor: VISITOR) is
            deferred
            end – accept
    end – component ELEMENT
```

```
    component CONCRETE_ELEMENT

        inherit
            ELEMENT

        feature accept(a_visitor: VISITOR) is
            do
                a_visitor.visit_concrete_element(current)
            end – accept

    end – component CONCRETE_ELEMENT

end – pattern PVISITOR
```

Konfigurationsmöglichkeiten

Dieses Pattern bietet zwei Schnittstellen zur Erweiterung. Es lassen sich mehr konkrete Elemente anlegen und mehr konkrete Besucher. Die Erweiterung zu mehr Funktionalität durch mehr Besucher geschieht einfach durch Duplikation der Komponente `CONCRETE_VISITOR`. Das Anlegen weiterer konkreter Elemente wird im Folgenden demonstriert.

```
pattern PVISITOR2

    refine
        select PVISITOR <Ref_A>
            rename
                CONCRETE_ELEMENT as CONCRETE_ELEMENTA
        end

        PVISITOR <Ref_B>
            rename
                CONCRETE_ELEMENT as CONCRETE_ELEMENTB
        end

    component VISITOR
        cast
            rename
                visit_concrete_element from <Ref_A> as visit_concrete_elementa,
                visit_concrete_element from <Ref_B> as visit_concrete_elementb
        end
    end – component VISITOR

end – pattern PVISITOR2
```

Kombination mit anderen Design Patterns

Die Element-Struktur des Patterns lässt sich gut mit zweistufig strukturierten Design Patterns, wie z.B. dem Kompositum (28), dem Interpreter (31) und dem Dekorierer (36) kombinieren.
Zur Traversierung der Objektstruktur ist die Kombination mit dem Pattern Iterator (24) möglich.

Anwendung in *DrawIt*

In *DrawIt* wird die Funktionalität der Grafik-Struktur, die aus den Patterns Kompositum (28) und Dekorierer (36) zusammengesetzt ist, größtenteils durch Besucher ausgelagert. Die Traversierung der Struktur geschieht durch das Pattern Iterator (24), das mit dem Kompositum (28) verknüpft ist.
Die Besucher werden durch die in der Befehlskette enthaltenen Nutzerbefehle (beruhend auf dem Pattern Befehl (55)) durch eine Fabrikmethode (17) erzeugt.

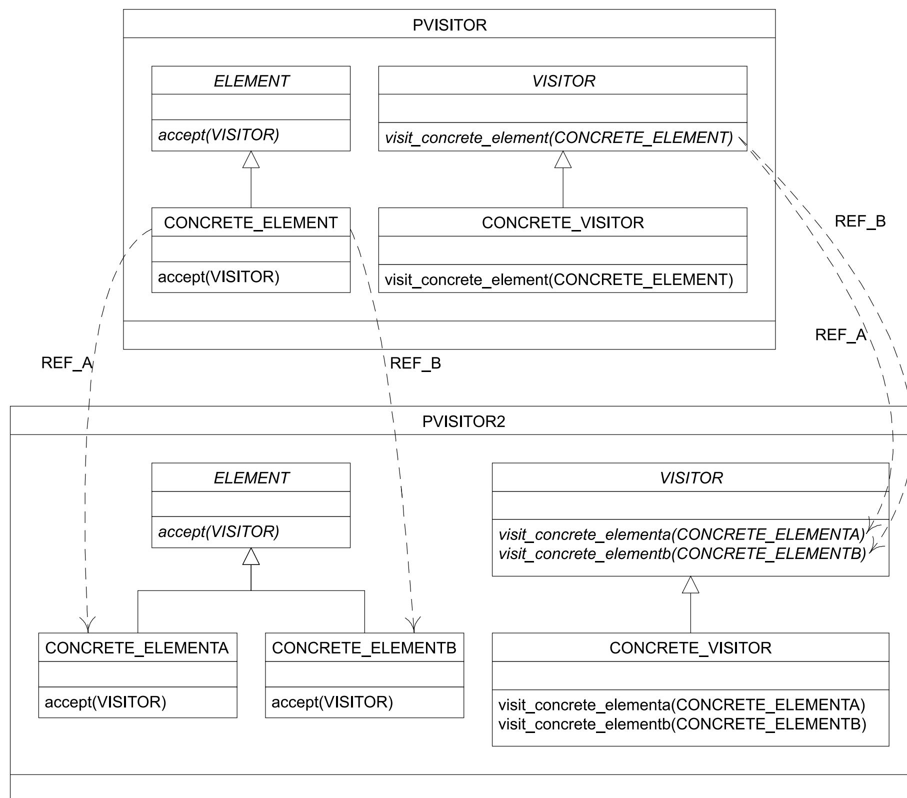


Abbildung 2.17: *Erweiterung des Patterns Besucher um mehrere konkrete Elemente*

2.4.11 Strategie (Strategy)

Struktur

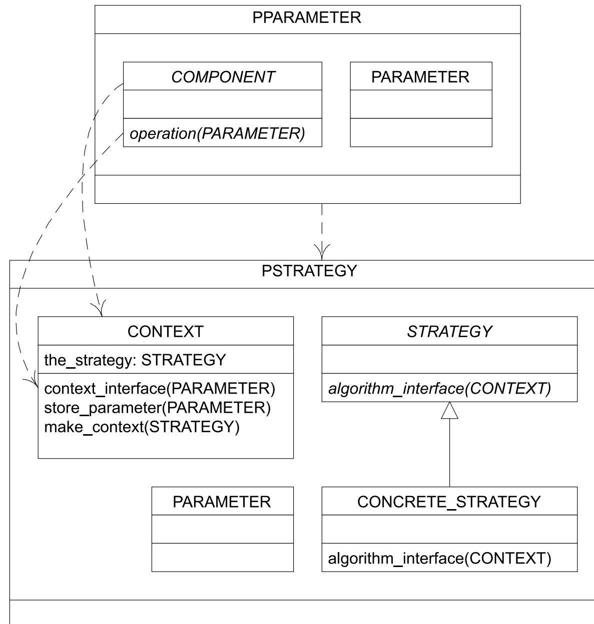


Abbildung 2.18: *Verfeinerung zum Design Patterns Strategie*

Implementierung

Das Pattern wird durch die Komponenten `CONTEXT`, `STRATEGY` und `CONCRETE_STRATEGY` implementiert. Bei den Operationen `context_interface` und `algorithm_interface` ist wieder nicht klar, welche Parameter sie haben.
In der Praxis können beide Methoden unterschiedliche Parameter verwenden. In diesem allgemeinsten Fall lässt sich für die Methode `context_interface` der Komponente `CONTEXT` keine Standardimplementation angeben, da auch nicht bekannt ist, wie die Parameter voneinander abhängig sind. Bei solch einer abstrakten Implementation hätte das implementierte Strategie-Pattern sehr wenig Substanz und wäre sehr aufwendig anzuwenden. Beim Einsatz des Patterns müssten zwei Parameter-Komponenten konkretisiert und die Methode `context_interface` in jedem Fall implementiert werden.
Da bei diesem allgemeinsten Fall der Aufwand zur Anwendung des Patterns größer ist als der Nutzen, wird in dieser Patternbibliothek eine in [3] vorgestellte Variante implementiert. Dabei speichert die Komponente `CONTEXT` den Parameter der Methode `context_interface` und ruft `algorithm_interface` mit sich selbst als Parameter auf. Zum Speichern des Parameters wird zusätzlich die Methode `store_parameter` implementiert, die zunächst einen leeren Rumpf besitzt. So ist diese Standardimplemention sofort für parameterlose Strategien geeignet. Unter Verwendung dieser Methode kann auch eine standardmäßige Delegation in der Methode `context_interface` implementiert werden.

Für die Sprache *PaL* sind die in [3] diskutierten Punkte 1. und 3. relevant:

1. *Definition der Strategie- und Kontextschnittstellen.* Hier werden drei Varianten zur Parametrisierung der Methoden vorgestellt. In der ersten Variante wird der Parameter direkt an die Strategieoperation übergeben. Diese Variante wäre in *PaL* einfach implementierbar, könnte in der Praxis aber eine Einschränkung darstellen. Die zweite

Variante ist die hier implementierte. In der letzten Variante hat die Strategie einen Verweis auf den Kontext, so dass die Strategieoperation keinen Parameter benötigt. Diese Variante ist ungeeignet, wenn ein Strategieobjekt für mehrere Kontexte verwendet werden soll (wie beim Design Pattern Fliegengewicht (33)).

3. *Strategieobjekte optionieren.* Diese Variante schlägt eine Implementation vor, bei der nicht immer ein Strategieobjekt benötigt wird. In dem Fall, dass keines vorhanden ist, wird eine Standardimplementation aufgerufen. Für diese Variante müsste bei einer Verfeinerung die Methode `context_interface` überschrieben werden.

Code

```
pattern PSTRATEGY
    refine
        PPARAMETER
            rename
                COMPONENT as CONTEXT
        end
    component CONTEXT
        cast
            rename
                operation as context_interface
        end
        creation
            make_context
        feature the_strategy: STRATEGY
        feature context_interface(a_parameter: PARAMETER) is
            do
                store_parameter((a_parameter);
                the_strategy.algorithm_interface(current)
            end
        feature store_parameter(a_parameter: PARAMETER) is
            do
            end
        feature make_context(a_strategy: STRATEGY) is
            do
                the_strategy := a_strategy
            end
    end – component CONTEXT
    component STRATEGY
        feature algorithm_interface(a_context: CONTEXT) is
            deferred
            end
    end – component STRATEGY
    component CONCRETE_STRATEGY
        inherit
            STRATEGY
    end – component CONCRETE_STRATEGY
end – pattern PSTRATEGY
```

Konfigurationsmöglichkeiten

Dieses Pattern bietet nur eine Möglichkeit der Erweiterung. Dafür wird die Komponente `CONCRETE_STRATEGY` einfach dupliziert.

Kombination mit anderen Design Patterns

Die Auslagerung von Funktionalität ist bei den meisten Design Patterns denkbar. So lässt sich die Komponente `CONTEXT` mit den Komponenten vieler anderer Patterns sinnvoll kombinieren.
In Kombination mit dem Pattern Fliegengewicht (33) lassen sich konkrete Strategien als Fliegengewichte verwalten.
Das Pattern Zustand (50) und dieses Pattern sind nahezu identisch. Es wird durch Verfeinerung dieses Patterns implementiert.

Anwendung in *DrawIt*

Das Pattern kommt in *DrawIt* nicht zum Einsatz. Dort geschieht die Auslagerung von Funktionalität mit Hilfe des Design Patterns Besucher (44).

2.4.12 Zustand (State)

Struktur

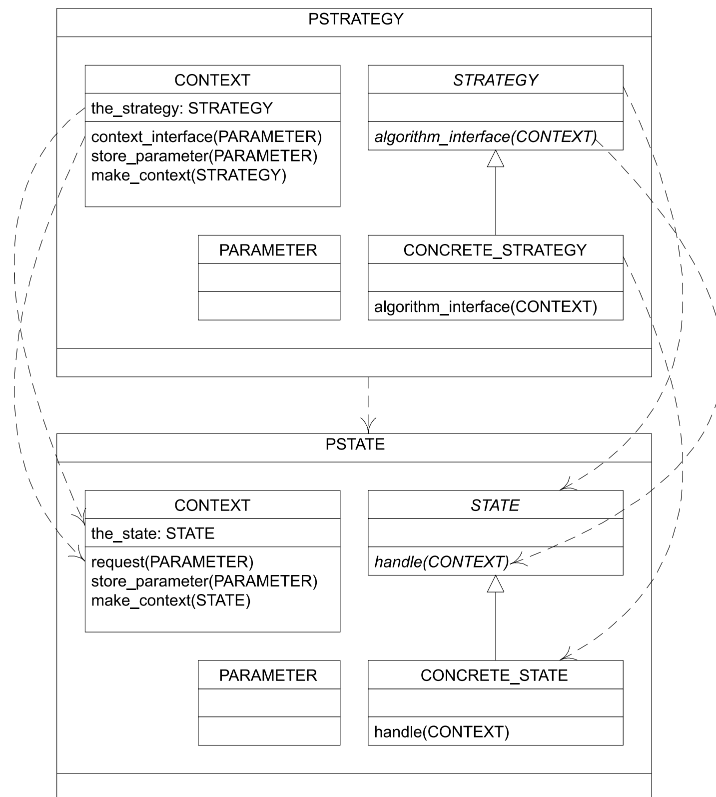


Abbildung 2.19: *Verfeinerung zum Design Patterns Zustand*

Implementierung

Auch wenn die Idee eine etwas andere ist, ist dieses Pattern bis auf die Bezeichner absolut identisch mit dem Design Pattern Strategie (47). Die einfachste Implementation ist die Verfeinerung des Patterns Strategie (47) und die Umbenennung der Bezeichner.

In [3] werden nur Implementationsaspekte aufgezählt, die nicht relevant für die Standardimplementation in *PaL* sind.

Code

```
pattern PSTATE

    refine
        PSTRATEGY
            rename
                STRATEGY as STATE,
                CONCRETE_STRATEGY as CONCRETE_STATE
        end

    component CONTEXT
        cast
            rename
                context_interface as request,
                the_strategy as the_state
        end
    end – component CONTEXT

    component STATE
        cast
            rename
                algorithm_interface as handle
        end
    end – component STATE

end – pattern PSTATE
```

Konfigurationsmöglichkeiten

Die Anzahl der konkreten Zustände lässt sich durch das Duplizieren der Komponente `CONCRETE_STATE` erhöhen.

Kombination mit anderen Design Patterns

Es sind die gleichen Kombinationen wie beim Design Pattern Strategie (47) möglich.

Anwendung in *DrawIt*

Das Pattern kommt in *DrawIt* nicht zum Einsatz.

2.4.13 Brücke (Bridge)

Struktur

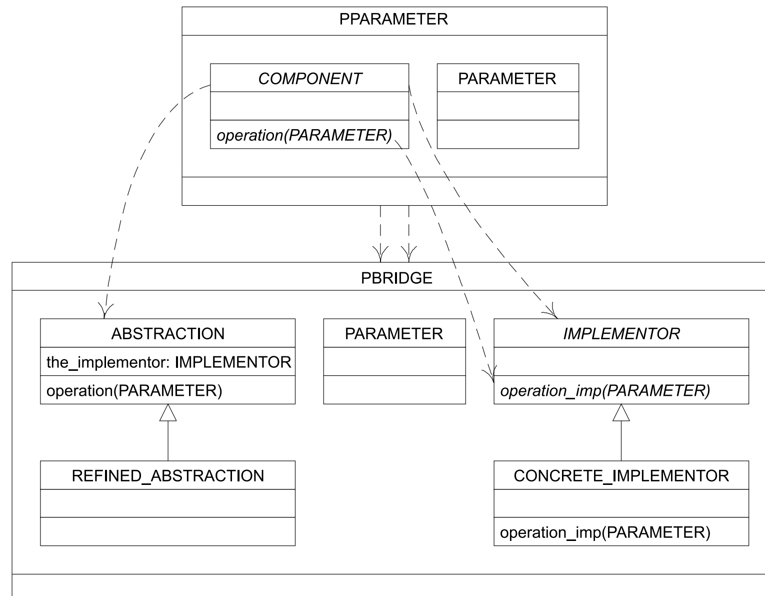


Abbildung 2.20: *Verfeinerung zum Design Pattern Brücke*

Implementierung

Dieses Pattern ähnelt stark dem Design Pattern Strategie (47). Es lagert ebenfalls eine Implementation aus. Bezüglich der Übergabeparameter treten die gleichen Probleme auf, wie bei der Strategie (47). Die Parameter der Operationen der `ABSTRACTION`-Komponenten und der `IMPLEMENTOR`-Komponenten sind nicht bekannt und müssen nicht identisch sein.
Die allgemeinste Lösung mit jeweils unterschiedlichen Parametertypen ist nur sehr aufwendig wiederzuverwenden. Eine Implementation der Delegation ist in dieser allgemeinen Variante nicht möglich. So ist das Ergebnis dieser umständlichen Wiederverwendung ein leeres Klassengerüst.
Mit der Implementation des Patterns Strategie (47) wird eine für die Sparche *PaL* sinnvolle Variante des Patterns vorgestellt. Da es mit dem Pattern Zustand (50) schon ein identisches Pattern dazu gibt, soll hier eine andere Variante implementiert werden.
Die Parameter der Methoden `operation` und `operation_imp` der Komponenten `ABSTRACTION` bzw. `IMPLEMENTOR` sollen vom gleichen Typ sein. Diese Variante lässt eine Implementation der Delegationsmethode `operation` zu. So bietet die Implementation mehr Substanz zur Wiederverwendung, stellt aber auch eine Einschränkung in den Anwendungsmöglichkeiten dar. Wenn diese Beschränkung den Einsatz des Patterns in einem bestimmten Anwendungsfall verhindert, kann auf des Pattern Strategie (47) zurückgegriffen werden oder es muss von Hand implementiert werden.

Die folgenden Punkte werden in [3] angesprochen:

1. *Nur ein Implementor.* Wenn es nur eine einzige Implementierung gibt, wäre es denkbar, die abstrakte Implementor-Komponente wegzulassen. Bei einer Implementation für eine Bibliothek muss aber der allgemeinste Fall vorgesehen werden. Daher ist hier die Vereinfachung auf eine Implementor-Komponente nicht möglich.

2. *Erzeugen des richtigen Implementorobjekts.* Durch die Kombination mit dem Design Pattern Abstrakte Fabrik (20) kann die Erzeugung der Implementorobjekte geregelt werden.

3. *Gemeinsame Nutzung von Implementierungsobjekten.* Die Gemeinsame Nutzung von Implementierungsobjekten ist mit dieser Implementation möglich. Sie kann z.B. durch die Kombination mit dem Design Pattern Fliegengewicht (33) realisiert werden.

4. *Verwenden von Mehrfachvererbung.* Dieser Ansatz ist weniger flexibel, da die Implementation nicht austauschbar ist. Die Variante wird durch das Design Pattern Adapter (73) implementiert.

Code

```
pattern PBRIDGE
    refine
        PPARAMETER
            rename
                COMPONENT as ABSTRACTION
        end

        PPARAMETER
            rename
                COMPONENT as IMPLEMENTOR
            end

    component ABSTRACTION

        feature the_implementor: IMPLEMENTOR

        feature operation(a_parameter: PARAMETER) is
            do
                the_implementor.operation_imp(a_parameter)
            end – operation

    end – component ABSTRACTION

    component REFINED_ABSTRACTION
        inherit
            ABSTRACTION
    end – component REFINED_ABSTRACTION

    component IMPLEMENTOR
        cast
            rename
                operation as operation_imp
        end
    end – component IMPLEMENTOR

    component CONCRETE_IMPLEMENTOR
        inherit
            IMPLEMENTOR
    end – component CONCRETE_IMPLEMENTOR

end – pattern PBRIDGE
```

Konfigurationsmöglichkeiten

Mit dem üblichen Mittel der Mehrfachverfeinerung lassen sich die Komponenten `REFINED_ABSTRACTION` und `CONCRETE_IMPLEMENTOR` in zwei voneinander unabhängigen Schritten duplizieren.

Kombination mit anderen Design Patterns

Wie schon bei der Diskussion der Implementierung angesprochen, kann die Erzeugung der Implementorobjekte durch die Kombination mit dem Design Pattern Abstrakte Fabrik (20) realisiert werden.

Die Anzahl der Implementorobjekte kann durch die Kombination mit dem Pattern Fliegengewicht (33) minimal gehalten werden.

Anwendung in *DrawIt*

Dieses Design Pattern kommt in *DrawIt* nicht zur Anwendung.

2.4.14 Befehl (Command)

Struktur

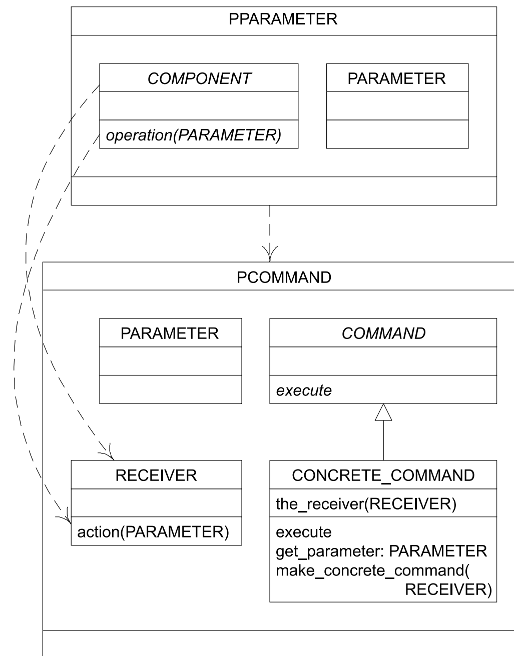


Abbildung 2.21: *Verfeinerung zum Design Patterns Befehl*

Implementierung

Diese Implementation übernimmt aus der Beschreibung in [3] nur die Komponenten `COMMAND`, `CONCRETE_COMMAND` und `RECEIVER`. Auf die Implementation der Teilnehmerkomponenten *Client* und *Invoker* wird verzichtet, da diese noch keine Funktionalität besitzen und auch nicht als Typ für die Schnittstellen anderer Komponenten benötigt werden. Die Anwendung des Patterns wird auf diese Art einfacher.

Auch bei diesem Pattern muss ein optimale Parametrisierung gefunden werden. Die Befehle sollen einheitlich behandelt werden können. Üblich ist, dass sie durch einen Menüpunkt oder einen Button ausgelöst werden. Die Methode `execute` der Komponente `COMMAND` bekommt daher keinen Parameter.
Bei der Aktion der Komponente `RECEIVER` ist dagegen nicht bekannt, wie die Parametrisierung geschehen soll. Daher wird hier das Hilfspattern Parameter (15) verwendet. Damit für die Methode `execute` von `CONCRETE_COMMAND` eine Standardimplementation angegeben werden kann, wird der Komponente zusätzlich die Methode `get_parameter` zugefügt. Diese Methode kann den Parameter aus dem konkreten Befehl extrahieren. Als Standardimplementation gibt sie `void` zurück und ist somit für parameterlose `RECEIVER` geeignet.

In [3] werden die folgenden Aspekte diskutiert:

1. *Intelligenz von Befehlsobjekten.* Die Befehlsobjekte können viel Funktionalität besitzen und alle Operationen auf dem Empfänger selbst ausführen. Andererseits kann das Befehlsobjekt die Aufgaben weitesgehend an den Empfänger delegieren. Beide Extreme sind mit dieser Implementation möglich und werden in *DrawIt* realisiert.

2. *Unterstützung von Undo und Redo.* Für diese Funktionalität muss die Befehlsgeschichte in einer Befehlskette gespeichert werden. In [3] wird für die Undo-Funktion eine Umkehrfunktion für den Befehl oder die Speicherung des Zustands vor Ausführung des Befehls vorgeschlagen. In *DrawIt* wurde eine andere Möglichkeit gewählt. Hier wird die Anwendung bei einem Undo zurückgesetzt und die Befehlskette bis zum vorhergehenden Befehl abgespielt. Die Undo- und Redo-Funktionalität ist auf diese Weise sehr einfach zu realisieren.

3. *Vermeiden von Fehlerlawinen im Undo-Prozess.* Zur Vermeidung von sich potenzierenden Fehlern beim Undo und Redo wird die Speicherung von Zuständen mit dem Pattern Memento (58) vorgeschlagen. Bei der in *DrawIt* verwendeten Lösung können solche Fehler erst gar nicht auftreten.

Code

```
pattern PCOMMAND

    refine
        PPARAMETER
            rename
                COMPONENT as RECEIVER
        end

    component COMMAND
        feature execute is
            deferred
            end – execute
    end – component COMMAND

    component CONCRETE_COMMAND

        inherit
            COMMAND

        creation
            make_concrete_command

        feature the_receiver: RECEIVER

        feature make_concrete_command(a_receiver: RECEIVER) is
            do
                the_receiver := a_receiver
            end

        feature execute is
            do
                the_receiver.action(get_parameter)
            end

        feature get_parameter: PARAMETER is
            do
                result := void
            end

    end – component CONCRETE_COMMAND

    component RECEIVER
        cast
            rename
                operation as action
        end
    end – component RECEIVER

end – pattern PCOMMAND
```

Konfigurationsmöglichkeiten

Durch einfache Duplizierung der Komponente `CONCRETE_COMMAND` lassen sich mehrere konkrete Befehle anlegen.

Kombination mit anderen Design Patterns

Durch Kombination mit dem Pattern Kompositum (28) lassen sich Makrobefehle realisieren.
Mit dem Design Pattern Memento (58) lassen sich Zwischenzustände für die Undo-Redo-Funktionalität speichern.
Die Kombination mit vielen anderen zweistufig strukturierten Patterns ist denkbar.

Anwendung in *DrawIt*

In *DrawIt* wird dieses Pattern verwendet, um Nutzerinteraktionen auszuführen. Die Nutzerbefehle sind durch eine Liste (12) zu einer Befehlskette aneinandergereiht. Jeder Befehl kann mittels einer Fabrikmethode (17) einen Besucher (44) erzeugen, der diesen Befehl auf der Dokumentstruktur, bestehend aus Kompositum (28) und Dekorierer (36), ausführt.

2.4.15 Memento (Memento)

Struktur

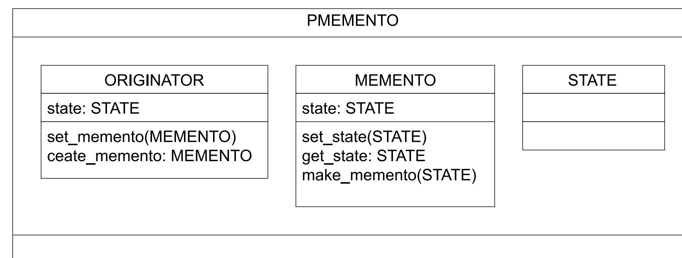


Abbildung 2.22: *Die Struktur des Design Patterns Memento*

Implementierung

Das Pattern Memento wurde hier mit den drei Komponenten `ORIGINATOR`, `MEMENTO` und `STATE` implementiert. In [3] wird zusätzlich noch die Klasse *Caretaker* aufgeführt. Über diese ist aber so wenig bekannt, dass es sich nicht lohnt, sie in die Patternimplementation mit aufzunehmen. Ist in der Praxis mehr über den *Caretaker* bekannt, so kann er z.B. durch die Kombination mit einem Container (11) später hinzugefügt werden.

Von der Implementationsdiskussion in [3] ist nur der Punkt 2. für die Sprache *PaL* von Bedeutung.

2. *Speicherung inkrementeller Änderungen.* In diesem Punkt wird anstatt der Speicherung des gesamten Zustands nur die Speicherung inkrementeller Änderungen vorgeschlagen. Auf diese Weise lässt sich eine relativ speichereffiziente Undo-Redo-Funktionalität zusammen mit dem Pattern Befehl (55) implementieren.

Code

```
pattern PMEMENTO

    component ORIGINATOR

        feature set_memento(a_memento: MEMENTO) is
            do
                state := a_memento.get_state
            end – set_memento

        feature create_memento: MEMENTO is
            local
                temp_memento: MEMENTO
            do
                !!temp_memento.make_memento(state);
                result := temp_memento
            end – create_memento

        feature state: STATE

    end – component ORIGINATOR
```

```
component MEMENTO
    creation
        make_memento
    feature get_state: STATE is
        do
            result := state
        end
    feature set_state(a_state: STATE) is
        do
            state := a_state
        end
    feature make_memento(a_state: STATE) is
        do
            state := a_state
        end
    feature state: STATE
end – component MEMENTO
component STATE
end – component STATE
end – pattern PMEMENTO
```

Konfigurationsmöglichkeiten

Bei diesem Pattern gibt es keine Erweiterungsschnittstellen, wie bei den meisten anderen Design Patterns. Es wird ohne Vervielfältigung einer Komponente angewendet. Durch Vererbung lassen sich aber auch unterschiedliche konkrete Komponenten anlegen.

Kombination mit anderen Design Patterns

Dieses Pattern kann zusammen mit dem Design Pattern Befehl (55) Zustände für die Undo-Redo-Funktionalität speichern.
Die Mementos können in einem Container (11) abgelegt werden und mit einem Iterator (24) traversiert werden.

Anwendung in *DrawIt*

Dieses Design Pattern wird in *DrawIt* nicht verwendet.

2.4.16 Beobachter (Observer)

Struktur

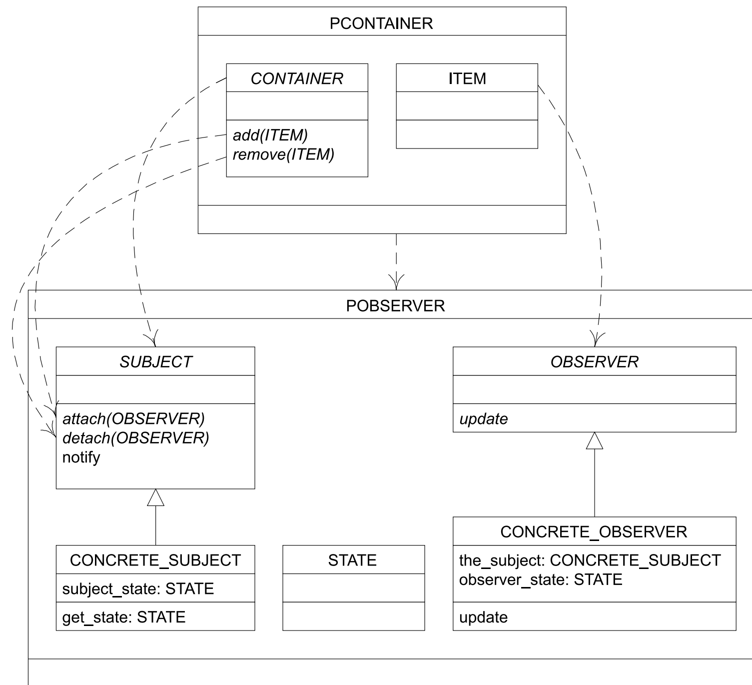


Abbildung 2.23: *Verfeinerung zum Design Pattern Beobachter*

Implementierung

Diese Implementation geht davon aus, dass ein Subjekt beliebig viele Beobachter verwalten können muss. Daher entstehen die Komponenten `SUBJECT` und `OBSERVER` durch Verfeinerung des Hilfspatterns Container (11). Davon erben die Komponenten `CONCRETE_SUBJECT` und `CONCRETE_OBSERVER`, wobei `CONCRETE_OBSERVER` mit `the_subject` eine direkte Referenz auf `CONCRETE_SUBJECT` hat.
Die Methoden `notify` und `update` werden ohne Parameter implementiert, da der konkrete Beobachter den Zustand des konkreten Subjektes erfragen kann. Die Methode `notify` besitzt noch keine konkrete Implementation, da bei dem abstrakten Pattern Container (11) noch nicht klar ist, wie über die Items iteriert wird.

In [3] werden zahlreiche Aspekte der Implementation diskutiert:

1. *Abbildung von Subjekten auf ihre Beobachter.* Hier werden Möglichkeiten aufgeführt, wie Subjekte ihre Beobachter verwalten. Darunter werden direkte Referenzen und Hash-Tabellen aufgezählt. Die Implementation mit dem Hilfspattern Container (11) ist abstrakt und damit vielseitig genug, um diese Möglichkeiten realisieren zu können.

2. *Beobachten von mehr als einem Subjekt.* In *DrawIt* beobachtet ein Verbinder die beiden Grafiken, die er verbindet. Dafür wird im konkreten Beobachter die Referenz `the_subject` dupliziert. In der Methode `update` werden beide Subjekte abgefragt. Andere Varianten sind mit dieser Standardimplementation ebenfalls möglich.

3. *Auslösen der Aktualisierung.* Die Methode `notify` kann durch das Subjekt selbst oder durch die zustandsändernden Operationen angestoßen werden. In *DrawIt* sind die Operationen, die die Subjekte bearbeiten, der Auslöser.

4. *Fehlerhafte Referenzen auf gelöschte Objekte.* Durch die Garbage Collection von Eiffel, die auch in *PaL* wirkt, gibt es keine fehlerhaften Referenzen auf gelöschte Objekte. Ein Objekt ist nicht gelöscht, wenn es noch Referenzen darauf gibt. Um ein Objekt richtig zu löschen, muss die `detach`-Operation korrekt angewendet werden.

5. *Sicherstellen, dass der Subjektzustand vor der Benachrichtigung konsistent ist.* Hier wird die Anwendung des Design Patterns Schablonenmethode (75) vorgeschlagen, um auch bei Subjekt-Hierarchien den genauen Zeitpunkt der Benachrichtigung der Beobachter kontrollieren zu können.

6. *Vermeiden von beobachterspezifischen Aktualisierungsschnittstellen: Das Push- und das Pull-Modell.* Beim Push-Modell werden mit der Methode `update` über Parameter dem Beobachter alle notwendigen und evtl. auch unwichtigen Funktionen übergeben. Im Pull-Modell besitzt die Operation `update` keine Parameter. Der Beobachter holt sie sich vom Subjekt. In dieser Bibliothek wurde das Pull-Modell implementiert, da das Push-Modell schwerer wiederverwendbar ist.

7. *Explizites Festlegen der interessierenden Änderungen.* Durch Mehrfachverfeinerung des Patterns kann ein Subjekt mehrere Container von Beobachtern verwalten, von denen jeder für eine andere Art der Änderung zuständig ist. Beobachter können sich so nur für die interessierenden Änderungen anmelden.

8. *Kapseln komplexer Aktualisierungssemantik.* Bei einer komplexen Semantik der Aktualisierung wird der Einsatz eines Änderungsmanagers vorgeschlagen. In [3] ist dafür eine erweiterte Patternstruktur angegeben. Durch Kombination des Patterns Vermittler (63) mit dem hier implementierten Beobachter-Pattern lässt sich eine ähnliche, gleichmächtige Struktur erzeugen.

9. *Kombinieren von Subjekt- und Beobachterklassen.* Durch eine Verfeinerung kann man die Subjekt- und Beobachterklassen miteinander verschmelzen lassen.

Code

```
pattern POBSERVER

    refine
        PCONTAINER
            rename
                CONTAINER as SUBJECT,
                ITEM as OBSERVER
        end

    component OBSERVER
        feature update is
            deferred
            end – feature update
    end – component OBSERVER

    component CONCRETE_OBSERVER

        inherit
            OBSERVER

        feature the_subject: CONCRETE_SUBJECT

        feature observer_state: STATE

        feature update is

            do
                observer_state := the_subject.get_state
            end

    end – component CONCRETE_OBSERVER

    component STATE
    end – component STATE
```

```
    component SUBJECT

        cast
            rename
                add as attach,
                remove as detach
        end

        feature notify is
            deferred
            end – notify

    end – component SUBJECT

    component CONCRETE_SUBJECT

        inherit
            SUBJECT

        feature get_state: STATE is
            do
                result := subject_state
            end

        feature subject_state: STATE

    end – component CONCRETE_SUBJECT

end – pattern POBSERVER
```

Konfigurationsmöglichkeiten

Dieses Pattern bietet zahlreiche Möglichkeiten der Konfiguration. Wie bei anderen Patterns lassen sich die konkreten Komponenten vervielfältigen. Dabei gibt es hier mehrere Variationen. Die Komponenten `CONCRETE_SUBJECT` und `CONCRETE_OBSERVER` lassen sich abhängig voneinander in einem Verfeinerungsschritt oder unabhängig voneinander in unterschiedlichen Verfeinerungsschritten vervielfältigen.
Bei der abhängigen Vervielfältigung existiert zu jeder konkreten Implementation eines Subjekts eine konkrete Implementation eines Beobachters und umgekehrt.
Wird nur das konkrete Subjekt vervielfältigt, so hat ein konkreter Beobachter gleichzeitig mehrere Referenzen auf konkrete Subjekte und kann diese gleichzeitig beobachten.
Durch Duplizierung der konkreten Beobachter können unterschiedliche Ausprägungen dieser Komponente erzeugt werden, die den gleichen Typ eines konkreten Subjektes beobachten.

Kombination mit anderen Design Patterns

Durch die zweistufige Komponentenstruktur des Patterns ist es leicht mit anderen zweistufigen Patterns kombinierbar.
Durch Kombination mit dem Design Pattern Vermittler (63) lässt sich eine komplexe Aktualisierungssemantik kapseln.
Wird das Pattern mit dem Iterator (24) kombiniert, so lassen sich die beim Subjekt angemeldeten Besucher leicht traversieren. Eine Implementation der Methode `notify` ist dann möglich.
Die Liste (12) bietet eine konkrete Implementation der Container-Funktionalität.
(Siehe dazu auch Abschnitt 2.5.5.)

Anwendung in *DrawIt*

In *DrawIt* wird dieses Pattern für Verbinder verwendet. Diese beobachten die verbundenen Grafiken und passen gegebenenfalls ihre Position und Größe an. Um jeden Grafiktyp in einen Vebinder verwandeln zu können, wird dieses Pattern mit dem Dekorierer (36) kombiniert. Die Verwaltung der beim Subjekt angemeldeten Beobachter wird durch Kombination mit den Patterns Iterator (24) und Liste (12) implementiert.

2.4.17 Vermittler (Mediator)

Struktur

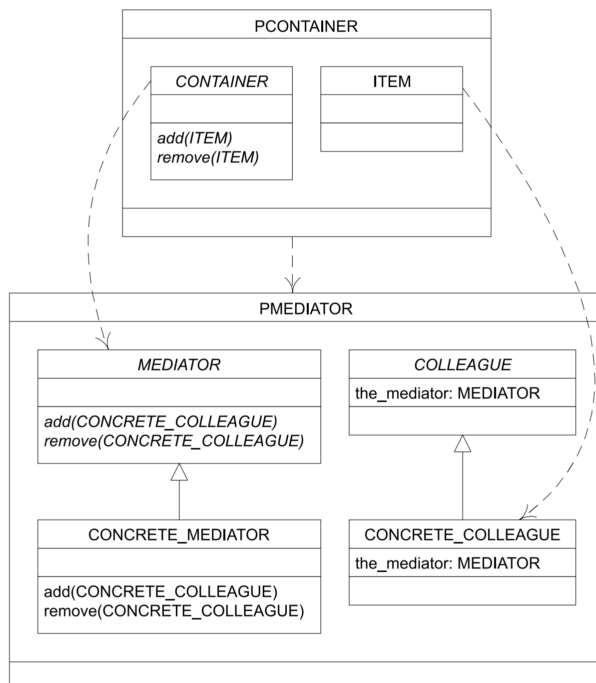


Abbildung 2.24: *Verfeinerung zum Design Pattern Vermittler*

Implementierung

Das Pattern besteht aus den vier Grundkomponenten `MEDIATOR`, `CONCRETE_MEDIATOR`, `COLLEAGUE` und `CONCRETE_COLLEAGUE`. Die Komponente `COLLEAGUE` hat mit `the_mediator` eine direkte Referenz auf den `MEDIATOR`. Die Komponente `CONCRETE_MEDIATOR` kann mit Komponenten vom Typ `CONCRETE_COLLEAGUE` umgehen. Wieviele das sind, ist von Anwendung zu Anwendung unterschiedlich. Auf Grund der Tatsache, dass es mehrere sein können, wird hier das Pattern Container (11) verwendet. `ITEM` wird dabei auf `CONCRETE_COLLEAGUE` und `CONTAINER` auf `MEDIATOR` abgebildet. So können durch `CONCRETE_MEDIATOR` unterschiedliche konkrete Container implementiert werden.

Die folgenden Aspekte der Implementation werden in [3] diskutiert:

1. *Weglassen der abstrakten Vermittlerklasse.* Da oft nur ein konkreter Vermittler gebraucht wird, schlägt man hier vor, die abstrakte Vermittlerklasse wegzulassen. In dieser Bibliothek muss der allgemeinste Fall implementiert werden. Daher kann der abstrakte Vermittler nicht weggelassen werden.

2. *Die Interaktionen von Kollegen- und Vermittlerklassen.* In diesem Pattern ist noch keine Interaktion implementiert. Durch Kombination mit dem Pattern Beobachter (60) kann die Benachrichtigung der Komponenten untereinander realisiert werden.

Code

```
pattern PMEDIATOR

    refine
        PCONTAINER
            rename
                CONTAINER as MEDIATOR,
                ITEM as CONCRETE_COLLEAGUE
            end

    component MEDIATOR
    end – component MEDIATOR

    component CONCRETE_MEDIATOR
        inherit
            MEDIATOR
    end – component CONCRETE_MEDIATOR

    component COLLEAGUE
        feature the_mediator: MEDIATOR
    end – component COLLEAGUE

    component CONCRETE_COLLEAGUE
        inherit
            COLLEAGUE
    end – component CONCRETE_COLLEAGUE

end – pattern PMEDIATOR
```

Konfigurationsmöglichkeiten

Für unterschiedliche Ausprägungen des konkreten Vermittlers wird die Mehrfachverfeinerung so angewendet, dass die Komponente `CONCRETE_MEDIATOR` dupliziert wird.
Für mehrere Ausprägungen der Komponente `CONCRETE_COLLEAGUE` wird diese Komponente dupliziert. Dabei muss darauf geachtet werden, dass die Listenfunktionalität in den Vermittler-Komponenten ebenfalls dupliziert wird.

Kombination mit anderen Design Patterns

Die Container-Funktionalität dieses Patterns kann durch Verknüpfung mit den Patterns Iterator (24) und Liste (12) implementiert werden.
Durch die Kombination mit dem Pattern Beobachter (60) kann die Vermittler-Komponente einen Änderungsmanager implementieren.
Da dieses Pattern wie viele andere Design Patterns eine zweistufige Struktur besitzt, ist es leicht mit diesen Design Patterns kombinierbar.

Anwendung in *DrawIt*

Dieses Design Pattern kommt in *DrawIt* nicht zur Anwendung.

2.4.18 Erbauer (Builder)

Struktur

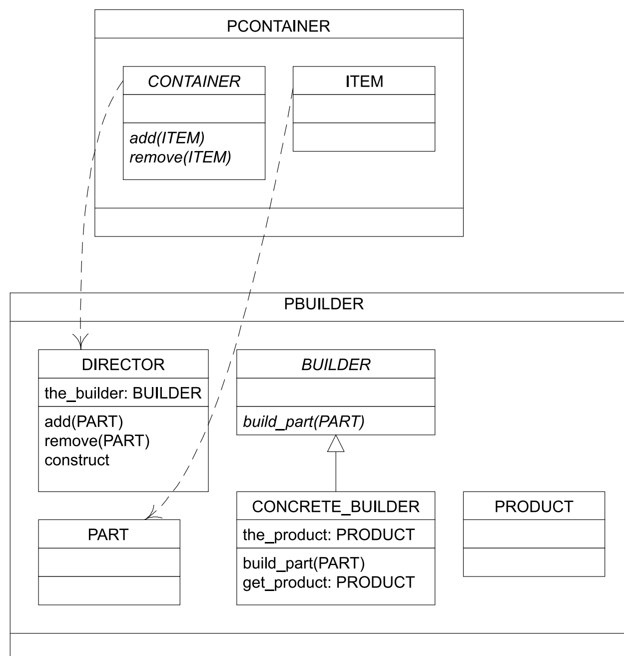


Abbildung 2.25: *Verfeinerung zum Design Pattern Erbauer*

Implementierung

Die Hauptkomponenten des Patterns sind `DIRECTOR`, `BUILDER` und `CONCRETE_BUILDER`. Der konkrete Erbauer erzeugt ein `PRODUCT`, in dem er vom `DIRECTOR` übergebene Objekte vom Typ `PART` zusammensetzt. Der `DIRECTOR` ist für die Verwaltung der Parts zuständig. Daher entstehen diese beiden Komponenten durch Verfeinerung des Hilfspatterns Container (11).
Die Methode `construct` wurde hier parameterlos implementiert. Im üblichen Fall werden durch diese Methode alle vorhandenen Parts an den Erbauer übergeben, wobei keine zusätzlichen Parameter benötigt werden. Sollten dennoch bei einer speziellen Anwendung zusätzliche Informationen erforderlich sein, so müssen diese in den Objekten der Komponenten `DIRECTOR` oder `CONCRETE_BUILDER` gespeichert werden.

Die folgenden Aspekte der Implementation werden in [3] angesprochen:

1. *Konstruktionsschnittstelle.* In diesem Abschnitt wird darauf hingewiesen, dass die Schnittstelle des Erbauers allgemein genug für alle konkreten Erbauer sein muss. Die hier implementierte Schnittstelle stellt keine Einschränkung dar.

2. *Keine abstrakte Produktklasse.* Es wird davon ausgegangen, dass sich die Produkte der unterschiedlichen konkreten Erbauer sehr unterscheiden können. Daher wird keine abstrakte Produktklasse eingeführt.

3. *Leere Methoden als Standardimplementation der Erbaueroberklasse.* Diese Standardimplementation wird auch in dieser Bibliothek benutzt. In einigen Fällen bleibt dem Programmierer dadurch das überflüssige Überschreiben unwichtiger Methoden erspart.

Code

```
pattern PBUILDER

    refine
        PCONTAINER
            rename
                CONTAINER as DIRECTOR,
                ITEM as PART
        end

    component DIRECTOR

        feature the_builder: BUILDER

        feature construct is
            deferred
            end

    end – component DIRECTOR

    component BUILDER
        feature build_part(a_part: PART) is
            do
            end
    end – component BUILDER

    component CONCRETE_BUILDER

        inherit
            BUILDER

        feature the_product: PRODUCT

        feature get_result: PRODUCT is
            do
                result := the_product
            end – get_result

    end – component CONCRETE_BUILDER

    component PRODUCT
    end – component PRODUCT

end – pattern PBUILDER
```

Konfigurationsmöglichkeiten

Durch Mehrfachverfeinerung können unterschiedliche konkrete Erbauer erzeugt werden. Die damit konstruierten Produkte müssen in der gleichen Verfeinerung dupliziert werden.
Durch Vererbung lassen sich unterschiedliche konkrete Parts erzeugen. Denkbar ist eine Vervielfältigung der Methode `build_part` entsprechend der Anzahl der konkreten Parts.

Kombination mit anderen Design Patterns

Durch die zweistufige Erbauerstruktur ist eine Kombination mit anderen zweistufigen Design Patterns denkbar.
Das Produkt des konkreten Erbauers kann ein Kompositum (28) sein.

Anwendung in *DrawIt*

Dieses Design Pattern kommt in *DrawIt* nicht zur Anwendung. Das Pattern könnte in diesem Programm zum Einsatz kommen, um die Befehlskette in einer Textdatei abzuspeichern.

2.4.19 Prototyp (Prototype)

Struktur

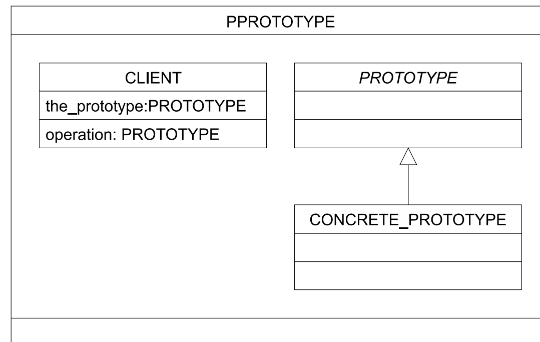


Abbildung 2.26: *Die Struktur des Design Patterns Prototyp*

Implementierung

Eiffel bietet die Funktionen `clone` und `deep_clone`. Da diese auch in *PaL* verwendbar sind, ist die Implementation dieses Patterns sehr einfach. Die Komponenten `PROTOTYPE` und `CONCRETE_PROTOTYPE` benötigen keine Funktionalität. Die Komponente `CLIENT` hat mit `the_prototype` eine Referenz auf ein Prototypobjekt. Die Methode `operation` löst mit Hilfe der Funktion `deep_clone` das Klonen des Objektes aus. Diese Methode benötigt dazu keinen Parameter. Sollen in einer speziellen Anwendung Prototypen in Abhängigkeit von Parametern geklont werden, so kann dafür eine neue Methode geschrieben werden, die die Methode `operation` aufruft.

Die folgenden Punkte werden in [3] diskutiert:

1. *Verwendung eines Prototypenverwalters.* Statt der Speicherung der Prototypen im Klienten, wird hier die Verwendung eines Prototypenverwalters vorgeschlagen. Diese Implementation ist auf die Speicherung im Klienten festgelegt und somit für die vorgeschlagene Alternative eher ungeeignet.

2. *Implementierung der Klone-Operation.* Der in [3] als am schwierigsten beschriebene Aspekt ist unter Eiffel kein Problem. Hierfür wird die Funktion `deep_clone` verwendet, mit der man einen Prototypen und alle dort referenzierten Objekte klont.

3. *Initialisierung geklonter Objekte.* Wenn der erzeugte Klon noch initialisiert werden muss, kann das über eine Methode mit zusätzlicher Funktionalität im Klienten realisiert werden. Diese Methode verwendet dann die bereits implementierte Methode `operation`.

Code

```
pattern PPROTOTYPE

    component CLIENT

        feature the_prototype: PROTOTYPE

        feature operation: PROTOTYPE is
            do
                result := deep_clone(the_prototype)
            end

    end – component CLIENT

    component PROTOTYPE
    end – component PROTOTYPE

    component CONCRETE_PROTOTYPE
        inherit
            PROTOTYPE
    end – component CONCRETE_PROTOTYPE

end – pattern PPROTOTYPE
```

Konfigurationsmöglichkeiten

Mit dem Mittel der Mehrfachverfeinerung lassen sich unterschiedliche konkrete Prototypen erzeugen.

Kombination mit anderen Design Patterns

Das Pattern kann zur Erzeugung von Objekten in Patterns, wie Kompositum (28) und Dekorierer (36), verwendet werden.

Anwendung in *DrawIt*

Prototypen werden in *DrawIt* in dieser Form nicht verwendet. Zur Erzeugung von Linien, Rechtecken und Ellipsen könnten Prototypen verwendet werden.

2.4.20 Singleton (Singleton)

Struktur

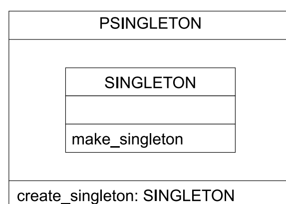


Abbildung 2.27: *Die Struktur des Design Patterns Singleton*

Implementierung

Es lässt sich in *PaL* nicht verhindern, dass mehrere Objekte einer Klasse erzeugt werden. Es kann aber eine Methode implementiert werden, über die nur ein Objekt erzeugt wird. Bei ausschließlicher Verwendung dieser Methode zum Erzeugen der Objekte erhält man ein Singleton-Objekt.
In Eiffel und somit auch in *PaL* kann eine Klasse die Erzeugung ihrer Objekte nicht selbst unterdrücken. Die Verwendung des Konstruktors ist immer möglich. Daher wird die Erzeugung der Singletons über die Patternmethode `create_singleton` implementiert. Die Implementation beginnt mit dem Schlüsselwort `once`, was bedeutet, dass das Ergebnis dieser Funktion nur ein einziges Mal berechnet wird. Daher wird nur bei dem ersten Aufruf von `create_singleton` ein Singleton-Objekt erzeugt, bei jedem weiteren Aufruf wird dieses Objekt zurückgegeben.

In [3] werden die folgenden Implementationsaspekte behandelt:

1. *Garantie eines einzigen Exemplars.* Diese Garantie kann in *PaL* nur gegeben werden, wenn zur Erzeugung von Singleton-Objekten ausschließlich die vorgegebene Funktion `create_singleton` verwendet wird.

2. *Ableiten der Singletonklasse.* Das Ableiten der Singletonkomponente ist bei dieser Implementation kein Problem, da die Erzeugung des einzigen Objektes vom Pattern aus gesteuert wird.

Code

```
pattern PSINGLETON

    component SINGLETON

        creation
            make_singleton

        feature make_singleton is
            do
            end

    end – component SINGLETON

    intern feature create_singleton: SINGLETON is
        once
            !!result.make_singleton
        end

end – pattern PSINGLETON
```

Konfigurationsmöglichkeiten

Bei diesem Pattern gibt es die klassischen Konfigurationsmöglichkeiten nicht.

Kombination mit anderen Design Patterns

Dieses Pattern ist so einfach strukturiert, dass der Kombination mit allen anderen Patterns nichts im Wege steht.
Denkbar ist eine Kombination mit den Erzeugungsmustern Fabrikmethode (17), Abstrakte Fabrik (20) und Erbauer (65).
In Kombination mit dem Design Pattern Fliegengewicht (33) kann die unnötige Erzeugung von Fliegengewicht-Objekten verhindert werden.

Anwendung in *DrawIt*

In dieser Anwendung wird von der Implementation des Singleton-Patterns kein Gebrauch gemacht. Diese Implementation eignet sich ausschließlich für einzigartige Komponenteninstanzen.

2.4.21 Fassade (Facade)

Struktur

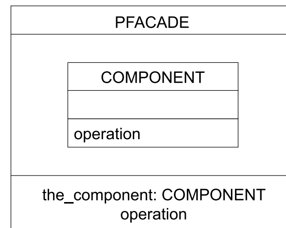


Abbildung 2.28: *Die Struktur des Design Patterns Fassade*

Implementierung

Das Pattern Fassade lässt sich auch mit der Sprache *PaL* nicht gut wiederverwenden. Das begründet sich damit, dass dieses Pattern eine Idee beschreibt, die sich schlecht durch eine Klassenstruktur ausdrücken lässt. Bei diesem Pattern ist es nicht klar, wie es bei der Anwendung aussehen wird.
Dennoch kann man dieses Pattern in *PaL* besser implementieren, als in klassischen objektorientierten Programmiersprachen. Bei dem Pattern Fassade kapselt eine Klasse viele andere Klassen. Diese Idee lässt sich hevorragend auf ein Pattern mit seinen gekapselten Komponenten übertragen. Der Quellcode wird dabei durch die Sprache *PaL* nicht nur strukturierter, sondern auch sicherer. Auf die gekapselten Komponenten kann nur über die externe Schnittstelle des Patterns zugegriffen werden. Komponenten anderer Patterns haben keinen Zugriff auf die gekapselten Komponenten.
Die hier angegebene Implementation ist zwar generell auch wiederverwendbar, der Aufwand steigt aber mit zunehmender Anzahl der gekapselten Komponenten und der Anzahl der Kopplungen zwischen Pattern und Komponenten erheblich. Diese Implementation ist daher mehr als Vorschlag zur Implementation des Fassade-Patterns zu verstehen.

Die folgenden beiden Punkte werden in [3] besprochen:

1. *Reduzieren der Kopplung zwischen Klient und Subsystem.* Im übertragenen Sinne ist die Kopplung zwischen dem Pattern und seinen Komponenten gemeint. Je geringer sich diese Kopplung realisieren lässt, desto realistischer ist die Wiederverwendung dieser Implementation.

2. *Öffentliche versus private Subsystemklassen.* In *PaL* sind automatisch alle Komponenten durch das Pattern gekapselt. Es gibt also nur private Komponenten. Öffentliche Subsystemklassen ließen sich aber durch Patterns realisieren, die den Komponenten unterschiedlicher Patterns bekannt sein dürfen.

Code

```
pattern PFACADE

    component COMPONENT
        feature operation is
            deferred
            end
    end – component COMPONENT

    feature the_component: COMPONENT

    feature operation is
        do
            the_component.operation
        end

end – pattern PFACADE
```

Konfigurationsmöglichkeiten

Für die Wiederverwendung dieser Implementation gibt es zwei Erweiterungsmöglichkeiten. Zum Einen lassen sich mehr Komponenten hinzufügen. Gleichzeitig muss das Attribut `the_component` und die Zugriffsmethode `operation` in der Patternschnittstelle dupliziert werden.
Zum Anderen kann die Kommunikation mit einer Komponente erweitert werden. Dafür wird die Methode `operation` des Patterns und der Komponente vervielfältigt.

Kombination mit anderen Design Patterns

Eine Kombination ist mit allen anderen Design Patterns denkbar. Die Komponenten des kombinierten Patterns sind dann ein Teil des Subsystems.

Anwendung in *DrawIt*

Ein Fenster mit seiner gesamten Funktionalität bildet in *DrawIt* eine Fassade. Die hier vorgeschlagene Implementation wurde dafür jedoch nicht wiederverwendet.

2.4.22 Adapter (Adapter)

Struktur

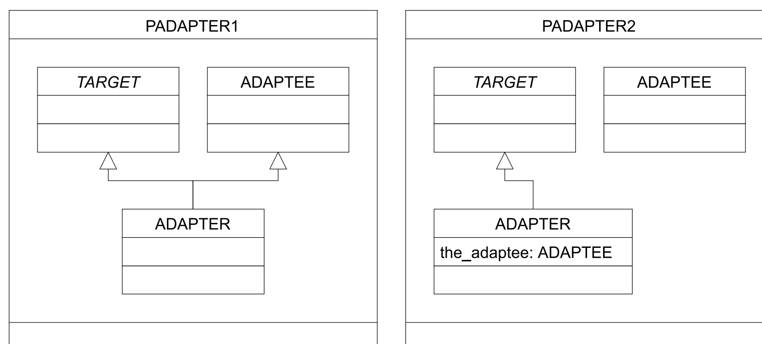


Abbildung 2.29: *Zwei Versionen des Design Patterns Adapter*

Implementierung

Für dieses Pattern werden in [3] zwei unterschiedliche Implementationen vorgeschlagen. Eine macht von Mehrfachvererbung Gebrauch, die andere verwendet Delegation. Da in *PaL* beide Implementationen möglich sind, werden hier auch beide aufgeführt.
Über die Funktionalität der beiden zu verbindenden Komponenten `TARGET` und `ADAPTEE` ist noch nichts bekannt. Daher werden hier keine Standardimplementationen der Methoden angegeben. Diese wären aufgrund der Parameter zunächst umständlich zu implementieren und später nur sehr umständlich wiederzuverwenden.

Die Diskussion der Implementierung in [3] betrachtet keine neuen, für *PaL* relevanten Aspekte.

Code

```
pattern PADAPTER1

    component TARGET
    end -- component TARGET

    component ADAPTEE
    end -- component ADAPTEE

    component ADAPTER
        inherit
            TARGET;
            ADAPTEE
    end -- component ADAPTER

end -- pattern PADAPTER1
```

```
pattern PADAPTER2

    component TARGET
    end -- component TARGET

    component ADAPTEE
    end -- component ADAPTEE

    component ADAPTER

        inherit
            TARGET

        feature the_adaptee: ADAPTEE

    end -- component ADAPTER

end -- pattern PADAPTER2
```

Konfigurationsmöglichkeiten

Bei diesem Pattern gibt es die klassischen Erweiterungsmöglichkeiten nicht. Man kann lediglich zwischen den beiden Implementationen wählen.

Kombination mit anderen Design Patterns

Mit Hilfe dieses Patterns lassen sich andere Design Patterns miteinander verknüpfen, indem deren Schnittstellen angepasst werden.

Anwendung in *DrawIt*

In *DrawIt* wird dieses Pattern nicht angewendet.

2.4.23 Schablonenmethode (Template Method)

Struktur

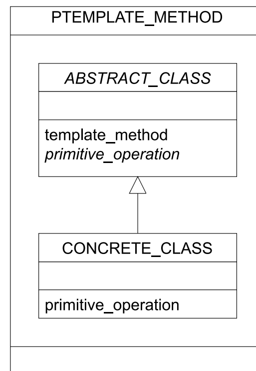


Abbildung 2.30: *Beispiel für das Design Pattern Schablonenmethode*

Implementierung

Bei diesem Design Pattern bringt die Sprache *PaL* keine Vorteile. Es ist nichts über die Schablonenmethode und über die primitiven Operationen bekannt. Man kann nicht voraussehen, wann und wieviele primitive Operationen aufgerufen werden. Ein weiteres Problem ist die Parametrisierung.
Dieses Design Pattern lässt sich daher nicht in die Patternbibliothek integrieren.

Die Abbildung 2.30 und der folgende Quelltext zeigen eine Beispielimplementation in der Sprache *PaL*.

Code

```
pattern PTEMPLATE_METHOD

    component ABSTRACT_CLASS

        feature template_method is
            do
                ...
                primitive_operation
                ...
            end

        feature primitive_operation is
            deferred
            end

    end – component ABSTRACT_CLASS

    component CONCRETE_CLASS
        feature primitive_operation is
            do
                ...
            end
    end – component CONCRETE_CLASS

end – pattern PTEMPLATE_METHOD
```

Konfigurationsmöglichkeiten

Da die Implementation nicht wiederverwendbar ist, kann sie auch nicht durch Verfeinerung konfiguriert werden.

Im Progammcode implementiert, kann es auf zwei Arten angewendet werden. Bei der objektorientierten Anwendung wird die primitive Operation in einer erbenden Komponente implementiert. Eine neue Variante ist die patternorientierte Verwendung der Schablonenmethode. Dabei wird die primitive Operation erst in einer Verfeinerung implementiert.

Kombination mit anderen Design Patterns

Die patternorientierte Anwendung der Schablonenmethode kann bei vielen anderen Design Patterns den Grad der Wiederverwendung erhöhen, da die Operation, die wie eine Schablonenmethode implementiert ist, nicht stets neu implementiert werden muss.
Die Idee der Schablonenmethode wird in dieser Bibliothek z.B. bei den Design Patterns Dekorierer (36), Strategie (47), Zustand (50) und Befehl (55) verwendet.
Eine weitere Anwendungsmöglichkeit der Schablonenmethode ist die Implementation der Operation zur vollständigen Iteration im Design Pattern Iterator (24).

Anwendung in *DrawIt*

In *DrawIt* wird die Schablonenmethode vielfach indirekt angewendet. So tritt sie häufig bei der Kombination des Patterns Besucher (44) mit anderen Patterns auf.

2.5 Beispiele für die Kombination von Patterns

Bei einigen Design Patterns stellten sich Kombinationsmöglichkeiten heraus, die für das Pattern besonders geeignet sind. Meist ergänzen sich die kombinierten Patterns so, dass die gemeinsame Struktur und Funktionalität ein nahezu vollständiges und gut wiederverwendbares System bilden.
Im Folgenden werden einige Kombinationen von Patterns, wie sie auch in *DrawIt* zur Anwendung kommen, demonstriert. Die Komponenten in den Grafiken werden dabei in verkürzter Form dargestellt.

2.5.1 Liste + Iterator

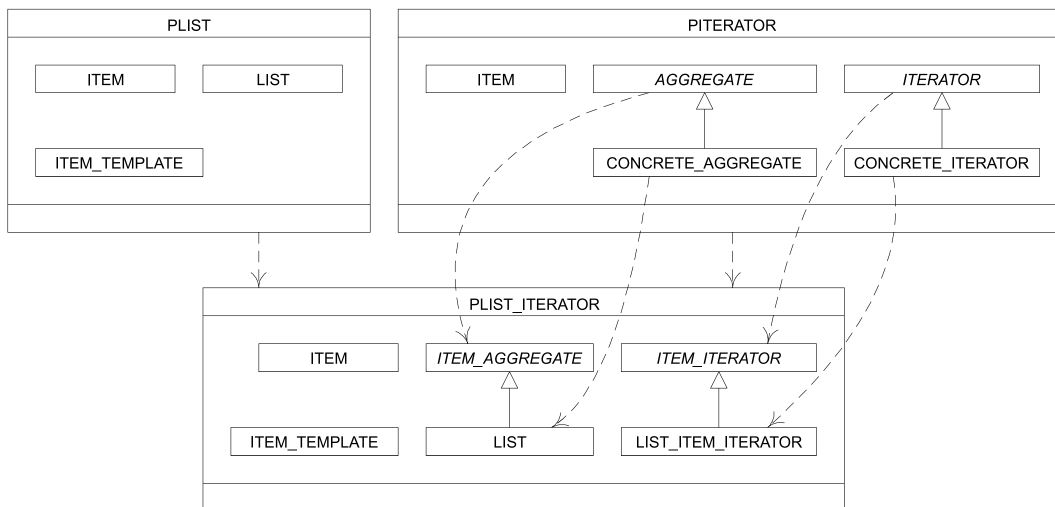


Abbildung 2.31: *Die Kombination der Patterns Liste und Iterator*

Die in dieser Bibliothek implementierte Liste (12) hat selbst noch keine Funktionalität zum Traversieren der Elemente. Durch die Kombination mit dem Design Pattern Iterator (24) wird die Liste um diese Funktionalität erweitert.

Die Abbildung 2.31 veranschaulicht die Kombination und Umbenennung der Komponenten. Bei der Verschmelzung dieser beiden Patterns wird die Komponente `CONCRETE_AGGREGATE` des Iterator-Patterns zur Komponente `LIST` des neuen Patterns.
Zusätzlich werden noch einige Umbenennungen vorgenommen, um die Wiederverwendbarkeit zu erhöhen und Fehler zu vermeiden. Wie beim Design Pattern Iterator (24) beschrieben, eignet sich die abstrakte Komponente `ITERATOR` zwar für unterschiedliche konkrete Typen von Aggregaten, aber nur für einen Item-Typ. In *DrawIt* werden Listen für viele unterschiedliche Item-Typen benötigt. Alle diese Listen benötigen unterschiedliche abstrakte Iteratoren. Damit es nicht zu Namenskonflikten bei der Kombination unterschiedlicher Iteratoren kommt, müssen die Komponenten des Iterator-Patterns umbenannt werden. Ein konsistentes Namenskonzept erreicht man, wenn vor die Namen der Komponenten des Iterator-Patterns der Name des Item-Typs gesetzt wird. Bei der Anwendung des Patterns müssen entsprechend der Umbenennung der Komponente `ITEM` die anderen Komponenten des Iterator-Patterns ebenfalls umbenannt werden.
In diesem speziellen Fall bedeutet das, `AGGREGATE` wird zu `ITEM_AGGREGATE`, `ITERATOR` wird zu `ITEM_ITERATOR` und `CONCRETE_ITERATOR` zu `LIST_ITEM_ITERATOR`.
In diesem Verfeinerungsschritt werden alle Operationen in der Komponente `LIST_ITEM_ITERATOR`, die im Design Pattern Iterator (24) noch nicht implementiert werden konnten, vollständig implementiert.

2.5.2 Kompositum + Iterator

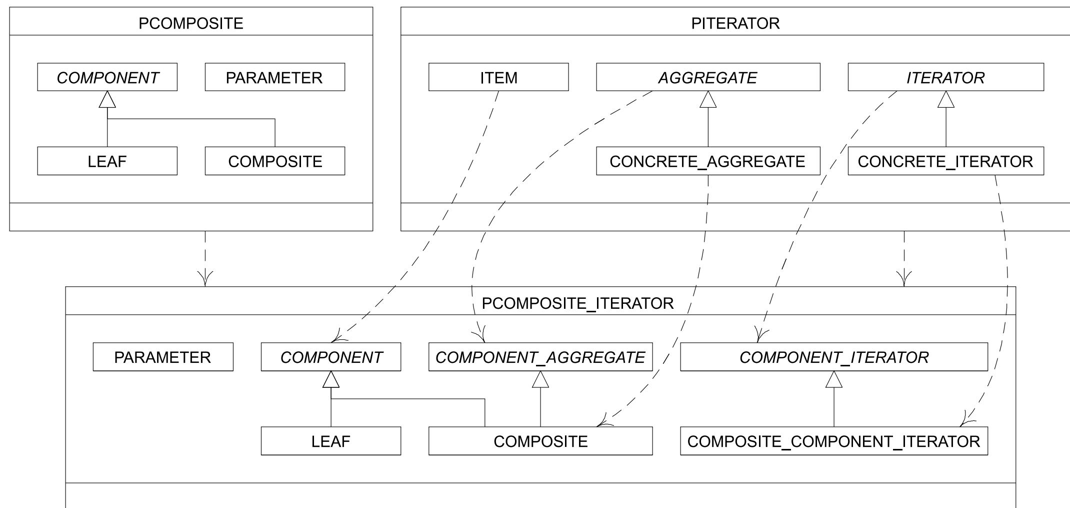


Abbildung 2.32: *Die Kombination der Patterns Kompositum und Iterator*

Bei der abstrakten Implementation des Design Patterns Kompositum (28) ist noch nicht klar, wie das Kompositum seine Komponenten verwaltet. Daher ist auch noch nicht bekannt wie das Kompositum eine Operation an alle seine Komponenten weiterleiten kann. Das Pattern Iterator (24) bietet eine Schnittstelle zum Traversieren der Kindobjekte, ohne dass man sich auf die Art des Containers festlegen muss. Mit Hilfe der Iterator-Funktionalität kann die Delegation einer Operation an die Kindobjekte in dem resultierenden Pattern implementiert werden.

Wie schon im vorhergehenden Abschnitt 2.5.1 beschrieben, werden auch hier die Namen der Komponenten des Iterator-Patterns um die Bezeichnung des Item-Typs (hier `COMPONENT`) erweitert.

2.5.3 Kompositum + Liste + Iterator

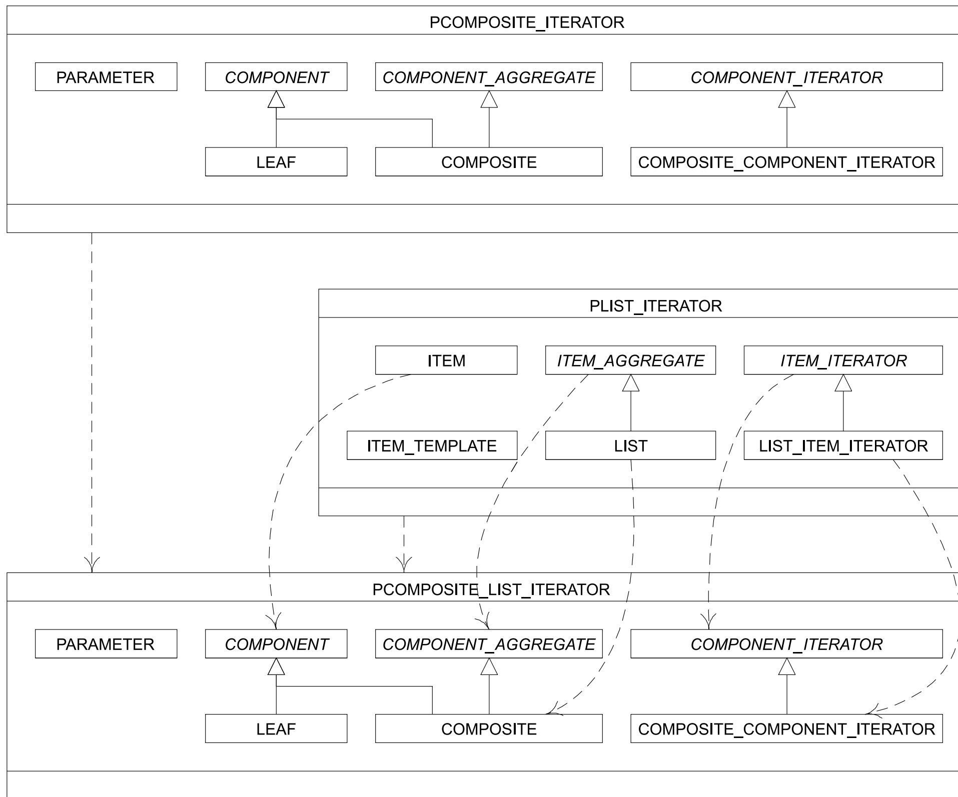


Abbildung 2.33: *Die Kombination der beiden vorhergehenden Patterns*

Im vorhergehenden Abschnitt 2.5.2 wurde das Pattern Kompositum (28) um die Funktionalität zum Traversieren der Komponenten erweitert. Zur zeitsparenden und praktischen Wiederverwendung fehlt dem Kompositum aber noch eine konkrete Implementation der Containerfunktionalität. Diese ist mit der Kombination von Liste (12) und Iterator (24) aus Abschnitt 2.5.1 gegeben.
Zur Generierung eines vollständig implementierten Kompositums werden daher die komplexen Patterns der beiden vorhergehenden Abschnitte miteinander kombiniert.

Bei der Kombination der beiden Patters (siehe Abbildung 2.33) bleiben die Bezeichnungen der Komponenten des Patterns Kompositum erhalten, die Bezeichnungen der Komponenten des Iterator-Patterns werden angepasst. Das Pattern erhält seine komplette Struktur und Funktionalität aus den beiden verfeinerten Patterns. Die Implementation oder das Überschreiben von Methoden ist nicht erforderlich.

2.5.4 Kompositum + Dekorierer

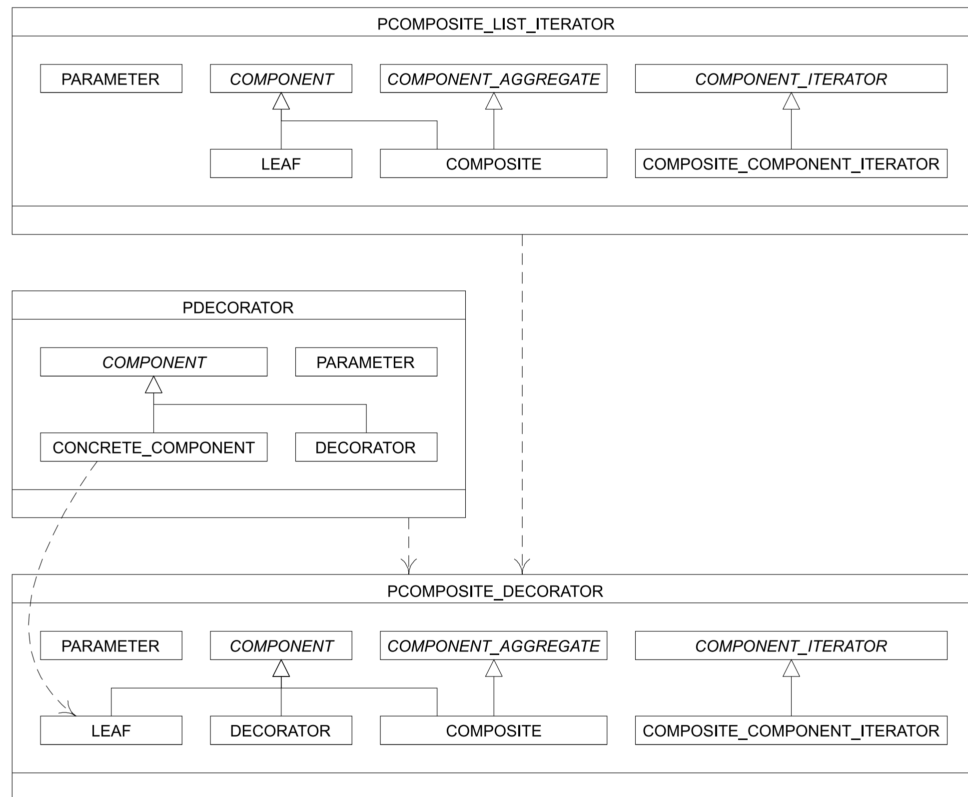


Abbildung 2.34: *Die Kombination eines Dekorierers mit dem vorhergehenden Pattern*

Bei der Beschreibung der Design Patterns Kompositum (28) und Dekorierer (36) wurde schon darauf hingewiesen, dass diese Patterns durch ihre ähnliche Struktur sehr einfach zu kombinieren sind. Die Abbildung 2.34 zeigt die Verknüpfung des vollständig implementierten Kompositums aus dem vorhergehenden Abschnitt 2.5.3 mit dem Pattern Dekorierer (36). Die einzige Anpassung, die bei dieser Kombination erforderlich ist, ist die Umbenennung der Komponente `CONCRETE_COMPONENT` aus dem Dekorierer-Pattern in `LEAF`.

2.5.5 Beobachter + Liste + Iterator

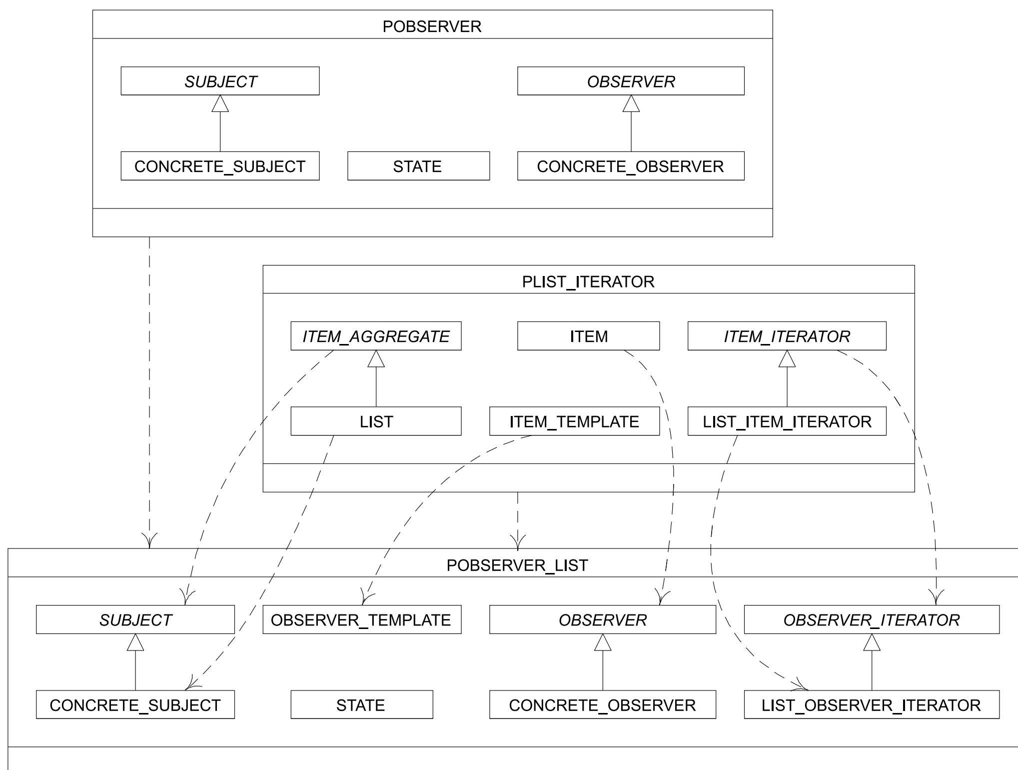


Abbildung 2.35: *Die Kombination der Patterns Beobachter und iterierbare Liste*

Im Design Pattern Beobachter (60) können sich bei einem Subjekt mehrere Beobachter an- und abmelden. Diese Container-Funktionalität wird in der abstrakten Implementation in dieser Bibliothek nur durch die Beschreibung der Schnittstelle angedeutet. Dadurch ist auch eine Implementation der Methode `notify`, die alle Beobachter bei einem Ereignis benachrichtigt, noch nicht möglich.

Damit das Beobachter-Pattern bei der Programmierung schnell zum Einsatz kommen kann, wird es hier mit der iterierbaren Liste aus Abschnitt 2.5.1, wie in der Abbildung 2.35 veranschaulicht, kombiniert. Die Synthese aus Liste (12) und Iterator (24) bietet die vollständige Implementation eines Containers und die einfache Traversierung der Elemente. In Kombination mit dem Pattern Beobachter (60) entsteht so die komplette Verwaltung der Beobachter im Subjekt. Die Methode `notify` kann durch diese Kombination unter Nutzung der Iterator-Funktionalität leicht implementiert werden.

Da der Iterator in dieser Patternkombination Elemente vom Typ `OBSERVER` verwaltet, wird vor die Bezeichner der Komponenten des Iterator-Patterns der Bezeichner `OBSERVER` gesetzt. So werden Konflikte mit anderen Iteratoren, wie z.B. dem `COMPONENT_ITERATOR` aus dem vorhergehenden Abschnitt 2.5.3, vermieden. Die Iterator-Komponenten heißen nun `OBSERVER_ITERATOR` und `LIST_OBSERVER_ITERATOR`. Die Aggregat-Komponenten werden auf die Subjekt-Komponenten `SUBJECT` und `CONCRETE_SUBJECT` abgebildet.

Kapitel 3

Eine Zeichenapplikation als Anwendungsstudie

In diesem Kapitel wird eine Anwendungsstudie in Form der einfachen Zeichenapplikation *DrawIt* vorgestellt. *DrawIt* bietet gewisse Grundfunktionalität, die es mit kommerziellen Produkten gemein hat, verzichtet allerdings auf Verzierungen, die keinen oder nur geringen demonstrativen Nutzen besitzen. So muss man z.B. auf die Verwendung von Farben gänzlich verzichten.

Die Anwendung *DrawIt* wird nun im Folgenden schrittweise entwickelt. Basierend auf der Analyse der geforderten Funktionalität (Abschnitt 3.1) wird der patternorientierte Entwurf durchgeführt. Diese Art des Entwurfs umfasst die folgenden Schritte:

1. *Entwurf der Design Patterns der Applikation* — Ein patternorientiertes Programm besteht im Normalfall aus mehreren Design Patterns. Einige dieser Design Patterns werden nur zur Wiederverwendung implementiert. Andere werden zur Laufzeit des Programms instanziiert. Diese Design Patterns werden auch *Design Patterns der Applikation* oder auch kurz *Applikationspatterns* genannt. Dieser Entwurfsschritt beinhaltet das Festlegen der Methoden und Attribute der Applikationpatterns und den Entwurf der Komponenten und deren Struktur auf objektorientierter Ebene[1].

2. *Herleitung der Design Patterns der Applikation* – Die zu benutzenden Design Patterns der Standardbibliothek und der notwendigen Verfeinerungsschritte werden identifiziert. Unter Benutzung des Baukastenprinzips werden grundlegende Design Patterns aus Kapitel 2 kombiniert und spezialisiert und damit wiederverwendet.

Dieses Kapitel schließt mit einer zusammenfassenden Betrachtung des patternorientierten Entwurfs. Der gewählte Ansatz wird dabei im Besonderen mit dem objektorienten Ansatz des Softwareentwurfs verglichen.

3.1 Die Funktionalität von *DrawIt*

Die Zeichenapplikation *DrawIt* soll die folgende grundlegende Funktionalität bieten:

1. *Repräsentation der grafischen Elemente* — *DrawIt* soll ein vektororientiertes Zeichenprogramm sein. Das beinhaltet die Repräsentation der Grafiken, wie z.B. Ellipsen, Linien oder Rechtecke, als Einheiten und nicht als Ansammlung von Punkten. Dies soll sich dann auch in der Benutzungsschnittstelle widerspiegeln. Grafiken können als

[1] Für eine Einführung in die benutzte Terminologie wird der Leser auf [1] und [4] verwiesen.

solche ausgewählt und Operationen auf ihnen ausgeführt werden. Grafiken besitzen bestimmte Eigenschaften, wie z.B. Position und Größe. Folgende Aktionen sollen auf den Grafiken definiert werden.

(a) *Allgemeine Operationen* — Hierzu gehört das Erstellen, das Löschen und das Ändern der Eigenschaften von Objekten.

(b) *Gruppieren* — Es soll möglich sein, ausgewählte Grafiken zu gruppieren. Das Resultat einer *Gruppieren*-Aktion ist eine Grafik, welche die vorher ausgewählten Grafiken enthält und von nun an als logische Einheit agiert. Die neuerstellte Gruppe von Grafiken soll sich homogen zu allen anderen Grafiken verhalten, d.h. dass man z.B. die Position eines Rechtecks und die Position einer Gruppe durch das *Drag & Drop* mit der Maus verändern kann.
Eine Gruppe von Grafiken soll auch wieder aufgelöst (degruppiert) werden können.

(c) *Verbinden* — Zwei Grafiken (*Grundgrafiken*) sollen über eine dritte Grafik (*Verbundgrafik*) verbunden werden können. Wird dann eine der Grundgrafiken verschoben oder ihre Größe geändert, so verändert die Verbundgrafik ihre eigene Größe und Position, um sich an die neue Position der Grundgrafiken anzupassen.
Verbindungen sollen auch wieder gelöst werden können.

(d) *Sperren von Eigenschaften gegen Veränderungen* — Es soll möglich sein, z.B. die horizontale Ausdehnung oder das Seitenverhältnis einer Grafik zu fixieren. Nachfolgende Änderungsanforderungen dieser Eigenschaften bleiben dann wirkungslos. Bedeutung bekommt diese Möglichkeit, wenn Operationen auf Gruppen von Grafiken ausgeführt werden.

2. *Undo- und Redomechanismen* — Aktionen bzw. Operationen auf den Grafiken in *DrawIt* sollen rückgängig gemacht werden können. Sollte sich eine Aktion, die rückgängig gemacht wurde, trotzdem im Nachhinein als nützlich erweisen, so soll diese Aktion wiederherstellbar sein.

3. *Benutzungsschnittstelle* — *DrawIt* soll eine moderne Benutzungsschnittstelle bieten, die intuitiv und funktional zu bedienen ist.

3.2 Entwurf der Benutzungsschnittstelle

Dieses Kapitel kann nicht alle Aspekte des Entwurfs und der Implementation von *DrawIt* abdecken. So ist die Rolle der Benutzungsschnittstelle unter Berücksichtigung der Aufgabenstellung als solche eher von untergeordnetem Interesse. Um das Verständnis für die folgenden Abschnitte zu erleichtern, wird hier die im Folgenden beschriebene Benutzungssschnittstelle vorgegeben. Sie kann z.B. durch ein *RAD* (*rapid application development*) - Toolkit erstellt worden sein.

Im Folgenden soll kurz auf die Komponenten der Benutzungsschnittstelle eingegangen werden. Generell ist das Applikationsfenster in Menüleiste, Werkzeugleiste, Arbeitsfläche und Statusleiste unterteilt (siehe Abbildung 3.1). Die Arbeitsfläche enthält das zu bearbeitende Dokument, d.h. hier werden Grafiken dargestellt und bearbeitet. Die Statusleiste wird für Ausgaben, wie z.B. Fehlermeldungen und Hinweise, verwendet. Mittels der Menü- und Werkzeugleiste kann der Benutzer Aktionen anstoßen. Menü- und Werkzeugleiste sind dabei gleichmächtig, d.h. jede Aktion, die im Menü ausgewählt werden kann, ist auch über die Werkzeugleiste zu erreichen und umgekehrt.

Es folgt eine Zusammenfassung der Arbeitsweise der einzelnen Aktionen, die über die Werkzeugleiste aufrufbar sind (beschriftet in Abbildung 3.1):

- *Programm beenden* — *DrawIt* wird beendet.

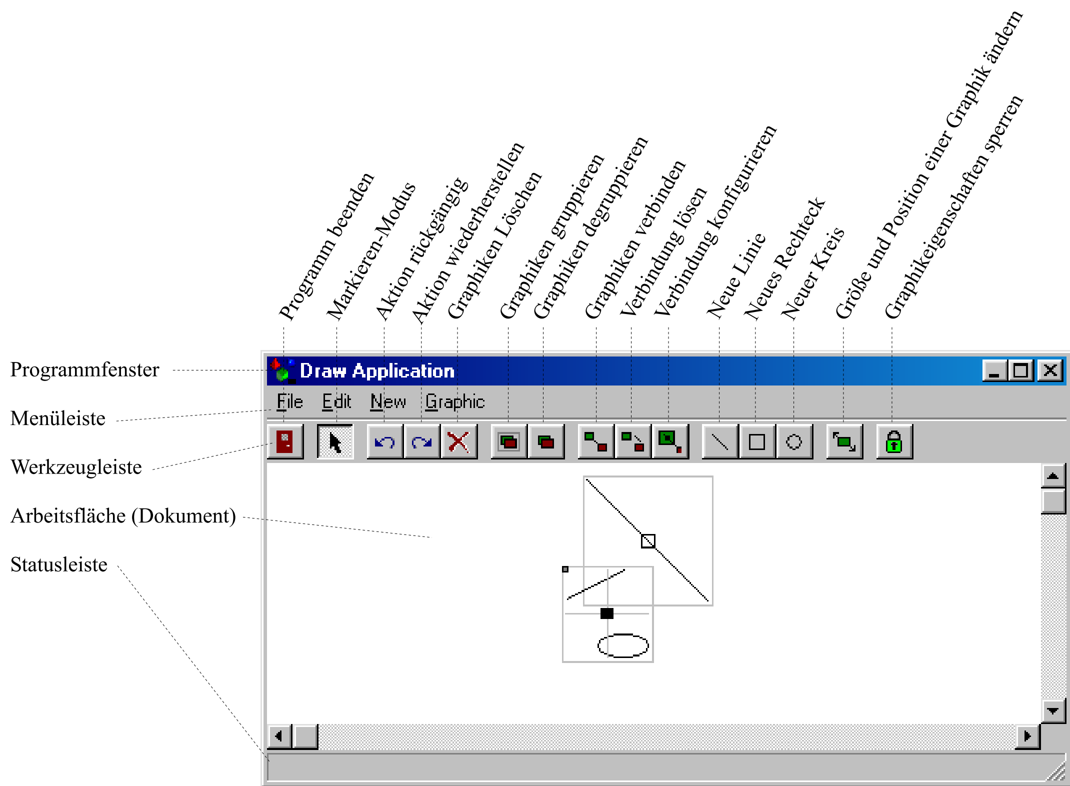


Abbildung 3.1: Die Benutzungsschnittstelle von *DrawIt*

- *Markieren - Modus* — Wenn dieser Knopf gedrückt ist, kann der Benutzer Grafiken der Maus markieren oder demarkieren. Dazu klickt er jeweils in das Rechteck in der Mitte einer Grafik, um eine nicht-markierte Grafik zu markieren bzw. um eine markierte Grafik zu demarkieren.

- *Aktion rückgängig* — Die letzte Aktion wird rückgängig gemacht. Auf diese Weise können beliebig viele Aktionen rückgängig gemacht werden.

- *Aktion wiederherstellen* — Eine rückgängig gemachte Aktion kann wiederhergestellt werden. Auf diese Weise können beliebig viele Aktionen wiederhergestellt werden. Dies kann jedoch nur dann geschehen, wenn seit dem letzten *Aktion rückgängig* keine weitere Aktion ausgeführt wurde.

- *Grafiken löschen* — Alle gegenwärtig markierten Grafiken werden aus dem Dokument gelöscht.

- *Grafiken gruppieren* (nur bei mindestens zwei markierten Grafiken) — Die gegenwärtig markierten Grafiken werden in einer Gruppe zusammengefasst.

- *Grafiken degruppieren* — Alle gegenwärtig markierten Gruppen werden degruppiert.

- *Grafiken verbinden* (nur bei drei markierten Grafiken) — Die beiden zuerst markierten Grafiken werden mittels der dritten markierten Grafik verbunden (siehe auch 3.1).

- *Verbindung lösen* — Alle gegenwärtig markierten Verbinder werden von ihren Grundgrafiken getrennt.

- *Verbindung konfigurieren* (nur bei einem markierten Verbinder) — Nach dem Starten dieser Aktion wird temporär der Markieren-Modus ausgeschaltet. Nun werden durch Anklicken die Verbindungspunkte des Verbinders neu gesetzt.

- *Neue Linie* — Nach dem Starten dieser Aktion wird der Markieren-Modus ausgeschaltet. Der Benutzer legt durch das Drücken der linken Maustaste den Startpunkt der Linie fest. Bei gedrückter Maustaste wird die Maus zum Endpunkt der Linie geführt. Durch das Loslassen der Maustaste wird der Endpunkt der Linie festgelegt. Der Benutzer kann nun noch weitere Linien erzeugen, da nicht automatisch in den Markieren-Modus zurückgeschaltet wird.
- *Neues Rechteck* — Nach dem Starten dieser Aktion wird der Markieren-Modus ausgeschaltet. Der Benutzer legt durch das Drücken der linken Maustaste die erste Ecke des Rechtecks (gewöhnlich die linke, obere) fest. Bei gedrückter Maustaste wird die Maus zur gegenüberliegende Ecke des Rechtecks geführt. Durch das Loslassen der Maustaste wird diese Ecke festgelegt. Der Benutzer kann nun noch weitere Rechtecke erzeugen, da nicht automatisch in den Markieren-Modus zurückgeschaltet wird.
- *Neuer Kreis* — Nach dem Starten dieser Aktion wird der Markieren-Modus ausgeschaltet. Analog zu *Neues Rechteck* kann nun ein Kreis gezeichnet werden. Der Benutzer kann nun noch weitere Kreise erzeugen, da nicht automatisch in den Markieren-Modus zurückgeschaltet wird.
- *Größe und Position einer Grafik ändern* (nur bei einer markierten Grafik) — Analog zur Aktion *Neues Rechteck* wird ein Rechteck festgelegt, in dem die markierte Grafik positioniert werden soll.
- *Grafikeigenschaften sperren* — Es wird eine Dialogbox geöffnet, die es dem Benutzer erlaubt, bestimmte Eigenschaften der markierten Grafiken zu sperren bzw. eine vorher verhängte Sperre wieder aufzuheben. Zu diesen Eigenschaften zählen:
  - Ausdehnung in X - und/oder Y - Richtung,
  - Seitenverhältnis X/Y bzw. Y/X.

3.3 Entwurf der Design Patterns der Applikation

Applikationspatterns werden der Regel nicht zur Verfeinerung benutzt und stellen somit das Ende einer Hierarchie von Verfeinerungsschritten dar. Der patternorientierte Entwurf beschreibt hier den umgekehrten Weg. Hier ist es zunächst wichtig, die Struktur, Attribute und Methoden der Design Patterns der Applikation zu entwerfen. Im konkreten Fall der Applikation *DrawIt* gibt es genau ein relevantes Applikationspattern, welches im Folgenden `PDOCUMENT_WINDOW` genannt wird.[2]. Eine Instanz dieses Design Patterns steht für ein Dokument und seine enthaltenen Grafiken. Die Grafiken selbst werden dabei durch Instanzen von Komponenten dieses Design Patterns repräsentiert. Es soll nun in den folgenden Unterabschnitten entworfen werden.

3.3.1 Entwurfsprobleme

Aufgrund der geforderten Funktionalität (siehe Abschnitt 3.1) von *DrawIt* lassen sich die folgenden Entwurfsprobleme identifizieren:

1. *Interne Repräsentation der Grafiken* — Wenn der Benutzer Ellipsen, Linien, etc. zeichnet, werden dafür Objekte entsprechender Komponenten angelegt. Welche Eigenschaften müssen diese Objekte haben? Wie kann eine angemessene Komponentenstruktur aussehen, damit im Nachhinein auch Operationen, wie z.B. das Gruppieren von Grafiken, einfach ausgeführt werden können. Dieser Teil des Entwurfs beeinflusst viele andere Aspekte des Entwurfs und kann deshalb als Kernstück in `PDOCUMENT_WINDOW` angesehen werden.

2. *Hinzufügen von Funktionalität* — Der entworfenen Komponentenstrukur muss nun Funktionalität hinzugefügt werden. Welche Ansätze gibt es, welche Vor- bzw. Nachteile haben sie und wie kann man sie in `PDOCUMENT_WINDOW` implementieren?

3. *Realisierung von Markierern* — Grafiken können mit der Maus markiert und auf diese Weise für folgende Operationen ausgewählt werden. Die Darstellung dieser Markierung und die Verwaltung des Markierungszustandes ist eigentlich der Grafik selbst zuzuordnen. Es ist jedoch vorzuziehen, diese Funktionalität von der der allgemeinen Grafik zu trennen. Wie kann dies adäquat erreicht werden?

4. *Realisierung von Verbindern* — In *DrawIt* ist es möglich, zwei Grafiken mittels einer dritten Grafik zu verbinden. Wie kann diese Funktionalität in den Entwurf integriert werden?

5. *Kommandos* — Aktionen, die der Benutzer anstößt, und unterliegende Operationen (Kommandos) müssen in einer Weise repräsentiert werden, dass sie rückgängig gemacht werden können und wiederherstellbar sind.

6. *Benutzungsschnittstelle* — Wie kann der Code für die Benutzungsschnittstelle so in `PDOCUMENT_WINDOW` integriert werden, dass spätere Erweiterungen einfach hinzugefügt werden können? Außerdem ist es wünschenswert, dass die Implementation der Benutzungsschnittstelle, die abhängig vom verwendeten Betriebssystem ist, von der systemunabhängigen Implementation anderer Teile von *DrawIt* getrennt werden kann.

[2]Es gibt zwei weitere Applikationspatterns `PDRAW_APPLICATION` und `PMAIN_WINDOW`, die allerdings nur aus technischen Gründen in Abschnitt 3.3.7 eingeführt werden.

3.3.2 Interne Repräsentation der Grafiken

Ein *Dokument* in *DrawIt* besteht aus *Grafiken*. Grafiken sind entweder Ausprägungen der Grundelemente Ellipse, Linie und Rechteck, oder komplexe Strukturen, die Gruppierungen anderer Grafiken sind. Auf der Ebene von *DrawIt* werden diese Grafiken durch Objekte repräsentiert. Diese Objekte sind dann Instanzen der entsprechenden Komponenten `CIRCLE`, `LINE`, `RECTANGLE` für die Grundelemente bzw. der Komponente `PICTURE` für Gruppierungen. Die Komponenten ihrerseits sind Bestandteil der Komponentenstruktur von `PDOCUMENT_WINDOW`.

Die interne Repräsentation der Grafiken muss nun folgenden Verantwortlichkeiten genügen:

1. Eine Grafik muss gewisse allgemeine Operationen zur Verfügung stellen (z.B. um sich selbst darstellen zu können) und allgemeine Daten, wie z.B. Position und Größe, verwalten.

2. Eine Grafik muss zusätzlich Operationen zur Verfügung stellen und Informationen speichern, die speziell diesen Typ Grafik betrifft (eine Gruppengrafik muss z.B. die Grafiken der Gruppe verwalten).

Eine gemeinsame Schnittstelle für die Objekte aller Grafiken

Zusätzlich zu den genannten Punkten ist es wünschenswert, dass die Objekte aller Grafiken uniform ansprechbar sind, d.h. auf Programmebene sollte es eine gemeinsame Schnittstelle für alle Grafiken geben. In solch einer Schnittstelle werden Operationen und Attribute aufgenommen, die auf alle Grafiken anwendbar sind.

In der objektorientierten Programmierung kann man diese Forderung durch das Mittel der Vererbung von Komponenten erfüllen. Es wird nun eine Komponente `GRAPHIC` als Oberkomponente der Komponenten `ELLIPSE`, `LINE`, `RECTANGLE` und `PICTURE` eingeführt. `GRAPHIC` definiert eine gemeinsame Schnittstelle für die Unterkomponenten, stellt jedoch noch keine Implementationen bzw. nur Standardimplementationen der Methoden zur Verfügung. In den Unterkomponenten werden diese Methoden überschrieben und somit spezielle Funktionalität hinzugefügt. Damit verhalten sich Methodenaufrufe zur Laufzeit polymorph. Der hier gewählte Ansatz ist gut erweiterbar, da neue Unterkomponenten auch im Nachhinein problemlos hinzugefügt werden können.

Rekursive Komposition

Die Gruppe ist eine spezielle Grafik, die andere Grafiken enthält. Dies wird auch auf Programmierebene deutlich. Die Gruppe wird durch die Komponente `PICTURE` implementiert, die eine Spezialisierung von `GRAPHIC` ist und einen Container für Grafiken implementiert. Es ist dabei unwichtig, ob es sich bei einer dieser Grafik nun um eine Ellipse, eine Linie, etc. oder sogar eine weitere Gruppe handelt. Deshalb wird die Komponente `PICTURE` im Folgenden so erweitert, dass sie Objekte vom Typ `GRAPHIC` aggregiert. Da auch ein Objekt der Komponente `PICTURE` vom Typ `GRAPHIC` ist, können auf diese Weise komplexe dynamische Strukturen beliebiger Tiefe erzeugt werden. Diese Entwurfsmethode wird auch *rekursive Komposition* genannt (siehe [3]). Abbildung 3.2 stellt die beschriebene Komponentenstruktur grafisch dar.

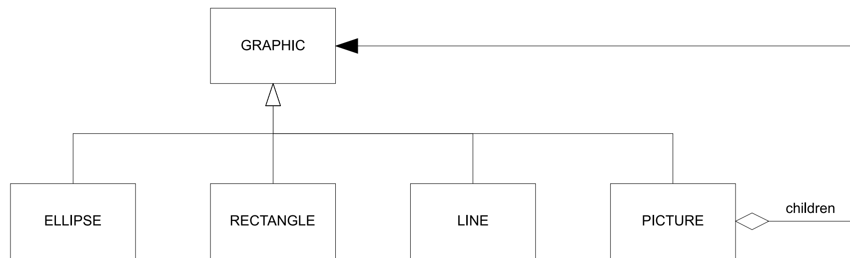


Abbildung 3.2: Das grafische Kompositum

Das Design Pattern *Kompositum*

Die vorgestellte Komponentenstruktur wird auch als *grafisches Kompositum* bezeichnet. Es handelt sich hierbei um eine Anwendung des Design Patterns Kompositum (28), welches in [3] eingeführt wurde. Das Dokument, also das oberste Kompositum, wird durch das Patternattribut `the_toplevel_picture` referenziert.

Koordinierungsmechanismen für Grafiken

Nachdem nun eine Komponentenstruktur für die Repräsentation von Grafiken in *DrawIt* entworfen wurde, ist es Ziel dieses Unterabschnitts, den Grafiken Eigenschaften basierend auf den vergebenen Verantwortlichkeiten zuzuordnen.

Die wichtigsten Eigenschaften einer Grafik sind ihre Position und Größe. Es ist also notwendig, diese Information in der Komponente `GRAPHIC` zu verwalten. Grundsätzlich verwendet *DrawIt* zwei logische Koordinatensysteme, um Grafiken zu positionieren:

- *ein lokales Koordinatensystem* — Dieses Koordinatensystem wird verwendet, um eine Grafik relativ zur direkt übergeordneten Gruppe zu positionieren. Der Vorteil des lokalen Koordinatensystem liegt darin, dass die Koordinaten der Kindgrafiken einer Gruppe nicht verändert werden müssen, wenn sich die Position der Gruppe ändert.

- *ein globales Koordinatensystem* — Dieses Koordinatensystem wird aus Effizienzgründen verwendet, um eine Grafik absolut im Dokument zu positionieren. Dadurch ist es nicht notwendig, das grafische Kompositum zu traversieren, um die absolute Position einer Grafik zu bestimmen. Das globale Koordinatensystem wird z.B. beim Neuzeichnen des Dokuments benutzt, da hier die Position im Dokument und nicht die in der übergeordneten Gruppe wichtig ist. Die Position einer Grafik im globalen Koordinatensystem verhält sich redundant zu der im jeweiligen lokalen Koordinatensystem.

Zusätzlich wird festgelegt, dass sich der gültige Bereich, der mit den Koordinatensystemen beschrieben wird, zwischen -1 und 1 auf den jeweiligen Koordinatenachsen befindet. Abbildung 3.3 zeigt in *a)* eine Dokumentstruktur, die eine Gruppe besitzt und in *b)* dessen Darstellung in *DrawIt* mit eingezeichneten Koordinatensystemen. Die Grafiken verwalten ihre Koordinaten sowohl im lokalen als auch im globalen Koordinatensystem. Bei den Grafiken auf oberster Ebene fallen beide Systeme zusammen, so dass die Koordinaten im lokalen mit den Koordinaten im globalen System übereinstimmen. Die Gruppe in Abbildung 3.3 definiert jedoch zusätzlich ein neues lokales System, so dass sich die Koordinatenpaare einer gruppierten Grafik effektiv unterscheiden.

Die Komponente `GRAPHIC` definiert nun Attribute für Position und Größe einer Grafik mittels einer Rechteckstruktur. Eine Rechteckstruktur speichert den Mittelpunkt und die Ausdehnung eines rechteckigen Bereichs in Koordinaten des jeweilig benutzten Koordina-

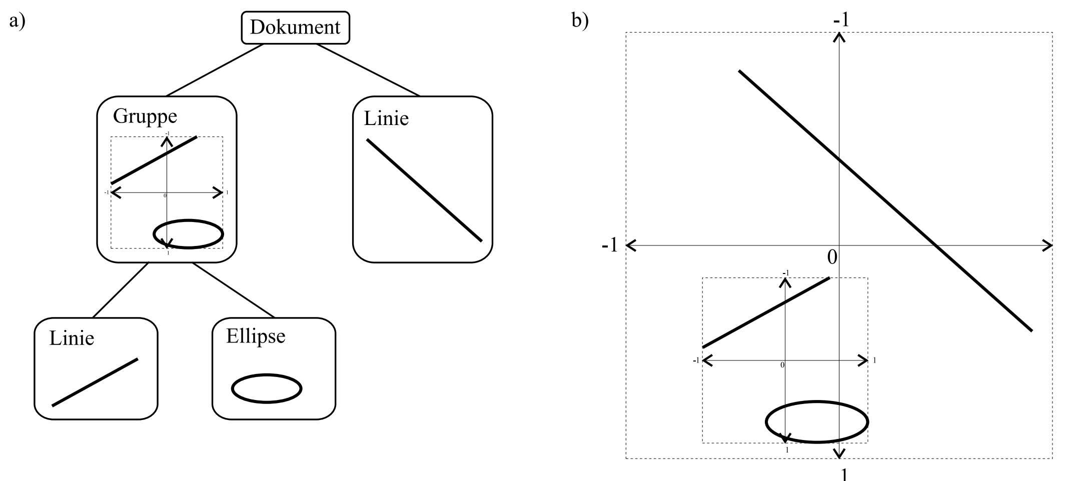


Abbildung 3.3: Die Koordinatensysteme in *DrawIt*

tensystems. Es werden folgende Attribute eingeführt.

Attribut	**Koordinatensystem**	**Bedeutung**
`client_extent`	lokal	definiert den Mittelpunkt und die Ausdehnung einer Grafik in der übergeordneten Gruppe
`extent`	global	definiert den Mittelpunkt und die Ausdehnung einer Grafik; ist redundant zu `client_extent`
`mbr`	global	definiert das miniale umschließende Rechteck einer Grafik (*Minimum Bounding Rectangle*); kann jedoch von `extent` abweichen (siehe Text und Abbildung 3.4)

Tabelle 3.1: Attribute zur Positionierung in `GRAPHIC`.

Aus Effizienzgründen werden in `GRAPHIC` drei Attribute für die Positionierung einer Grafik benutzt. Dies ist redundant. So können `extent` und `mbr` aus `client_extent` von einem komplizierten Algorithmus berechnet werden. Da diese beiden Attribute jedoch sehr häufig benötigt werden (z.B. für das Zeichnen der Grafiken), ist es sehr unwirtschaftlich, `extent` und `mbr` dynamisch zu berechnen. Da Redundanz immer die Gefahr der Inkonsistenz einschließt, muss im Verlauf der Implementierung auf die sorgfältige Behandlung der Attribute geachtet werden.

In einem Objekt einer Grafik müssen `extent` und `mbr` nicht notwendigerweise das gleiche Rechteck beschreiben. Da der Nutzer die Möglichkeit hat, Eigenschaften von Grafiken gegen Veränderungen zu sperren, kann es durchaus vorkommen, dass `extent` und `mbr` variieren. Dabei kann man dann von einer wirklichen (`extent`) und einer scheinbaren Größe (`mbr`) sprechen. Abbildung 3.4 demonstriert diesen Effekt. Im Bild *a)* ist eine Gruppe von Grafiken dargestellt, bei der die enthaltene Linie gegen eine Veränderung der Größe geschützt ist. Zunächst überdecken sich `extent` und `mbr`. Im Bild *b)* wurde diese Gruppe insgesamt mit

dem Faktor 1.5 skaliert. Die Linie ist unverändert geblieben, der Kreis ist jetzt jedoch, wie auch die gesamte Gruppe, um den Faktor 1.5 größer. Dadurch ist `mbr` nun kleiner als `extent`.

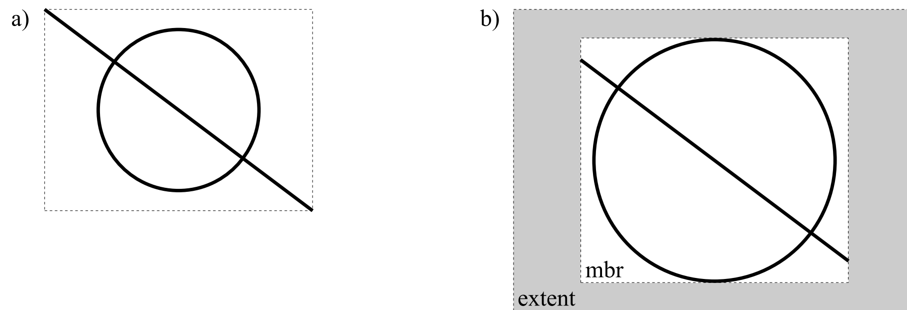


Abbildung 3.4: Der Unterschied zwischen Ausdehnung (`extent`) und dem umschließenden Rechteck (`mbr`) nach einer Skalierung mit dem Faktor 1.5.

3.3.3 Funktionalität für das grafische Kompositum

Das grafische Kompositum im Allgemeinen und die Komponente GRAPHIC im Speziellen besitzt momentan noch keinerlei Funktionalität. In diesem Unterabschnitt werden zwei Ansätze vorgestellt, den Entwurf um diese Notwendigkeit zu erweitern. In PDOCUMENT_WINDOW wird von beiden Möglichkeiten Gebrauch gemacht, da jeweils Vor- und Nachteile aufzeigbar sind, die sich aus dem speziellen Anwendungsfall ergeben.

Das Hinzufügen von Funktionalität bedeutet konkret, dass Methoden deklariert und implementiert werden. Ziel ist es, diese Methoden so zu definieren, dass neben der Effektivität auch Eigenschaften wie Wiederverwendbarkeit, Erweiterbarkeit, Kombinationsfähigkeit und Lesbarkeit eine Rolle spielen.

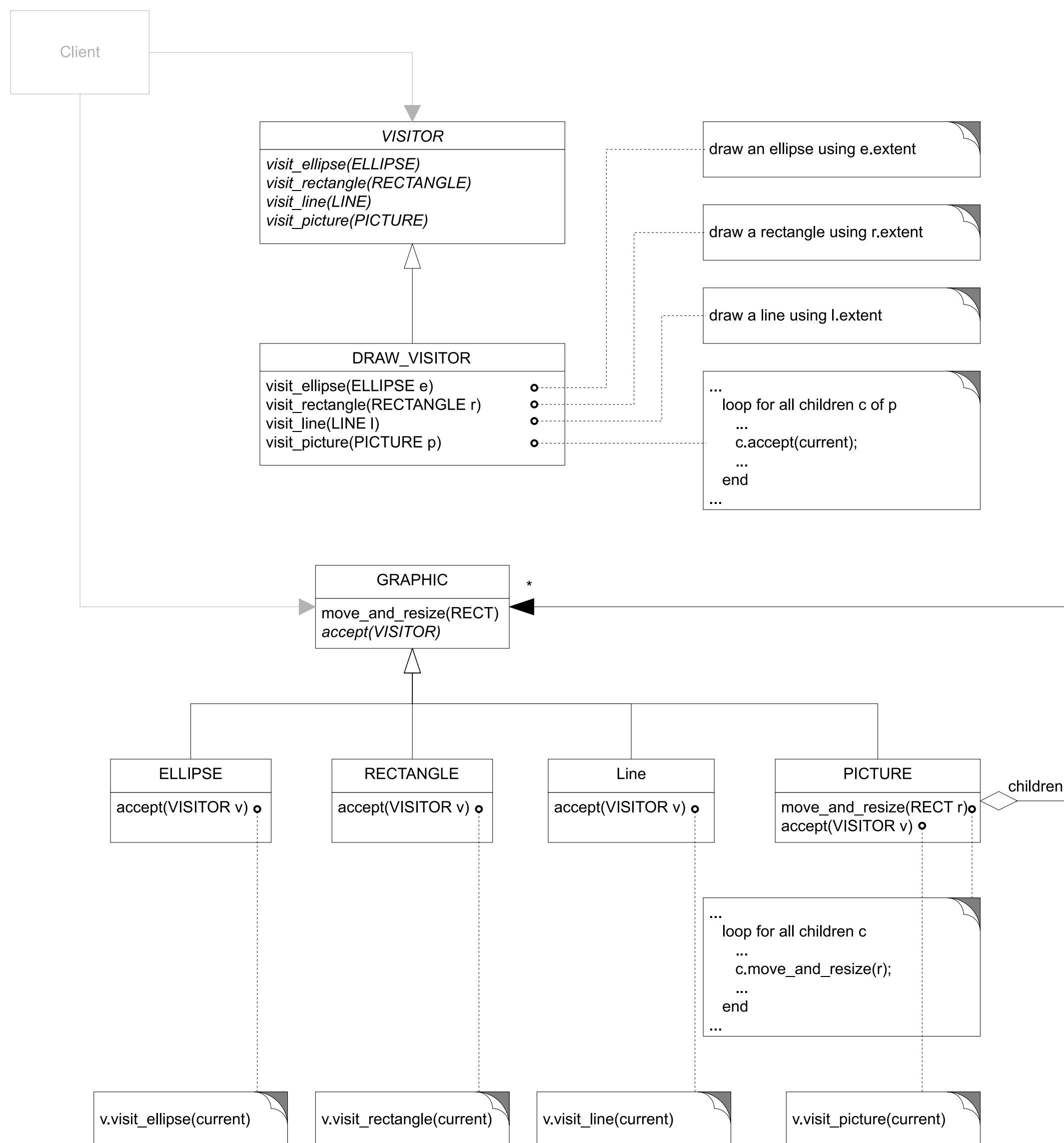


Abbildung 3.5: Funktionalität für das grafische Kompositum

Einlagern von Funktionalität

Im Design Pattern *Kompositum* wird neben der oben beschriebenen Komponentenstruktur zusätzlich eine Methode namens `operation` definiert. Diese Methode ist Teil der gemeinsamen Schnittstelle der Komponenten des Kompositums und wird je nach Gebrauch für bestimmte Komponenten überschrieben. In vielen Fällen ist es z.B. notwendig, `operation` für die Komponente *COMPOSITE* so zu implementieren, dass `operation` auch auf allen Kindobjekten ausgeführt wird. Methoden dieser Art werden auch im grafischen Kompositum benötigt. Im Folgenden wird das Beispiel der Methode `move_and_resize` diskutiert, um die Verwendung der Komponentenstruktur zu demonstrieren.

Die Methode `move_and_resize` wird für die Positionierung der Grafiken verwendet. Sie wird immer dann aufgerufen wenn der Benutzer die Position oder die Größe einer Grafik verändert. Im Falle einfacher Grafiken wird die geforderte Aktion auf lokaler Ebene ausgeführt. Eine Gruppe reagiert jedoch komplexer. In diesem Fall müssen dann auch die enthaltenen Grafiken in Position und Größe angepasst werden. Dazu wird `extent` und `mbr` aus `client_extent` berechnet.

Technisch gesehen, wird `move_and_resize` in `GRAPHIC` eine Standardimplementation vorgegeben und in `PICTURE` überschrieben, um auch enthaltene Grafiken einer Gruppe zu aktualisieren.

Auslagern von Funktionalität

Unter bestimmten Umständen ist es günstiger, die Funktionalität und damit Methoden und deren Implementationen aus den Komponenten auszulagern. Dies wird am Beispiel des Neuzeichnens der Grafiken demonstriert.

Generell sollte man Methoden und deren Implementationen aus den eigentlichen Komponenten auslagern wenn

- viele Unterkomponenten ihre eigene, spezielle Funktionalität benötigen, d.h. die Methode müsste im Einlagerungsfall häufig überschrieben werden.
- Implementationen von Methoden wiederverwendet werden sollen. Dies ist der Fall, wenn die Implementation einer Methode einer Unterkomponente die Implementation der Oberkomponente aufrufen muss[3].
- die Flexibilität erhöht werden soll, um z.B. *multiple dispatch* zu simulieren oder unsicheres *downcasting* zu vermeiden. Auf diesen Probleme trifft man, wenn sich ein Methodenaufruf eigentlich auch polymorph zu einem oder mehreren Übergabeparametern verhalten sollte.
  Beispiel zum multiple dispatch*:* Es soll eine Methode implementiert werden, die berechnet, ob sich zwei Grafiken überschneiden. Diese Methode muss so speziell sein, dass sie für jede mögliche Kombination von Grafiktypen korrekt arbeitet. Dies kann nicht mittels normalem *dynamic binding* von sogenannten *selfish methods*, wohl aber durch das Auslagern von Methoden, erreicht werden.
- allgemein die Lesbarkeit und Wartbarkeit im Einlagerungsfall leiden würde.

Das Auslagern von Funktionalität wird technisch realisiert, indem Methoden und deren Implementationen nicht mit in die eigentlichen, sondern in spezielle, für die Auslagerung

[3] An dieser Stelle is einzuwenden, dass Programmiersprachen wie z.B. *C++* ein *super*-Konstrukt besitzen, welches genau diesem Zweck dient. Da jedoch weder *Eiffel* noch *PaL* dieses Konstrukt bietet, wird hier auf die Auslagerung zurückgegriffen.

vorgesehene Komponenten aufgenommen werden. Zu diesem Zweck werden die Design Patterns Strategie (47) und Besucher (44) in [3] eingeführt und in Kapitel 2 implementiert. Im Folgenden wird jedoch nur auf den Besucher (44) eingegangen, da in `PDOCUMENT_WINDOW` auch nur dieser angewendet wird.

Es wird der abstrakte Besucher als abstrakte Komponente `VISITOR` eingeführt. `VISITOR` enthält die gemeinsame Schnittstelle aller Besucher. Im Wesentlichen enthält diese Schnittstelle Methodendeklarationen der Form `visit_xxxx(e: XXXX)` wobei `XXXX` jeweils für eine konkrete Komponente des grafischen Kompositums steht. Der konkrete Besucher `DRAW_VISITOR` wird als Unterkomponente von `VISITOR` eingeführt. Er implementiert die Schnittstelle von `VISITOR` und stellt somit Funktionalität zur Verfügung. Des Weiteren wird in `GRAPHIC` eine Methode `accept(v: VISITOR)` deklariert, die in allen konkreten Unterkomponenten überschrieben wird und zur entsprechenden `visit_xxxx`-Methode im übergebenen Besucher `v` delegiert. Das Verhalten des Besuchers wird genauer in Besucher (44)beschrieben.

Durch die Auslagerung der Methoden aus dem grafischen Kompositum wird zusätzlich erreicht, dass Funktionalität in konkrete Besucher gekapselt wird, d.h. ein konkreter Besucher enthält eine gesamte Methodenhierarchie. So enthält `DRAW_VISITOR` alle Methoden zum Neuzeichnen, für jede konkrete Komponente des grafischen Kompositums eine Spezielle.

Der aktuelle Entwurf wird in Abbildung 3.5 dargestellt.

3.3.4 Realisierung von Markierern

Nachdem eine Grafik erstellt wurde, muss es möglich sein, sie für weitere Aktionen auszuwählen. In *DrawIt* sowie in gängigen kommerziellen Programmen wird hierfür die Maus verwendet. Durch das Anklicken einer Grafik mit der linken Maustaste wird die Grafik markiert. Durch nochmaliges Anklicken wechselt der Markierungsstatus wieder zu *nicht markiert*. Es können mehrere Grafiken gleichzeitig markiert sein. Eine nachfolgende Aktion, wie z.B. *Löschen*, löscht dann alle markierte Grafiken. Per Konvention wird festgelegt, dass eine Grafik nur dann markiert werden kann, wenn sie sich auf oberster Ebene im grafischen Kompositum befindet. In diesem Abschnitt wird diskutiert, wie man den gegenwärtigen Entwurf um die Möglichkeit erweitern kann, Grafiken mit einem Markierungszustand auszustatten und diesen darzustellen.

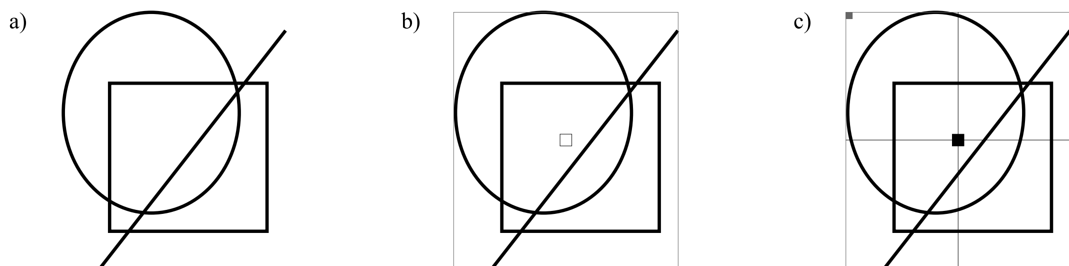


Abbildung 3.6: Darstellung von Markierungen: *a)* Markierung unsichtbar *b)* Markierung sichtbar, nicht markiert *c)* Markierung sichtbar, markiert.

Jegliche Art Markierung ist zunächst ausgeschaltet (*invisible*; siehe Abbildung 3.6 a)). Durch das Wechseln in den *Markieren-Modus* (siehe dazu auch Unterabschnitt 3.3.7) werden alle auswählbaren Grafiken mit einem Rahmen und einem kleinen Rechteck in der Mitte dargestellt (siehe Abbildung 3.6 b). Wird danach eine Grafik durch das Klicken der Maus in eines dieser kleinen Rechtecke markiert, so ändert sich die Darstellung der gesamten Grafik wie in Abbildung 3.6 c) dargestellt.

Die Verantwortlichkeit, die jeweilige Markierung darzustellen und den Markierungszustand zu verwalten, ist der Komponente `GRAPHIC` zuzuordnen. Dennoch ist es auf Entwurfsebene vorzuziehen, die entsprechende Funktionaliät in eine zusätzliche Komponente aufzunehmen, da Grafiken nur auf oberster Ebene des grafischen Kompositums markiert werden können. Grafiken die sich unterhalb dieser Ebene befinden, benötigen diese Funktionalität nicht.

Das Design Pattern *Dekorierer*

Es wird die Komponente `HIGHLIGHTER` eingeführt. Ein Objekt dieser Komponente wird dynamisch mit einem Objekt vom Typ `GRAPHIC` assoziiert. Das Objekt der Komponente `HIGHLIGHTER` wird als *Markierer* bezeichnet. Der Markierer ist ein Spezialfall eines *Dekorierers*, da er das mit ihm verbundene Objekt *verziert* oder auch *dekoriert*. Der Markierer speichert zusätzlich zur Assoziation den aktuellen Markierungszustand und stellt die entsprechende Markierung grafisch dar. Er sollte selbst vom Typ `GRAPHIC` sein, `HIGHLIGHTER` sollte also von `GRAPHIC` erben. Damit können sowohl Markierer als auch andere Objekte vom Typ `GRAPHIC` uniform angesprochen werden. Methodenaufrufe werden dann entweder zum dekorierten Objekt delegiert oder vom Markierer selbst bearbeitet (z.B. das Zeichnen der Markierung). Hierdurch verhält sich der Markierer als spezieller Dekorierer in weiten Teilen völlig transparent zum dekorierten Objekt.

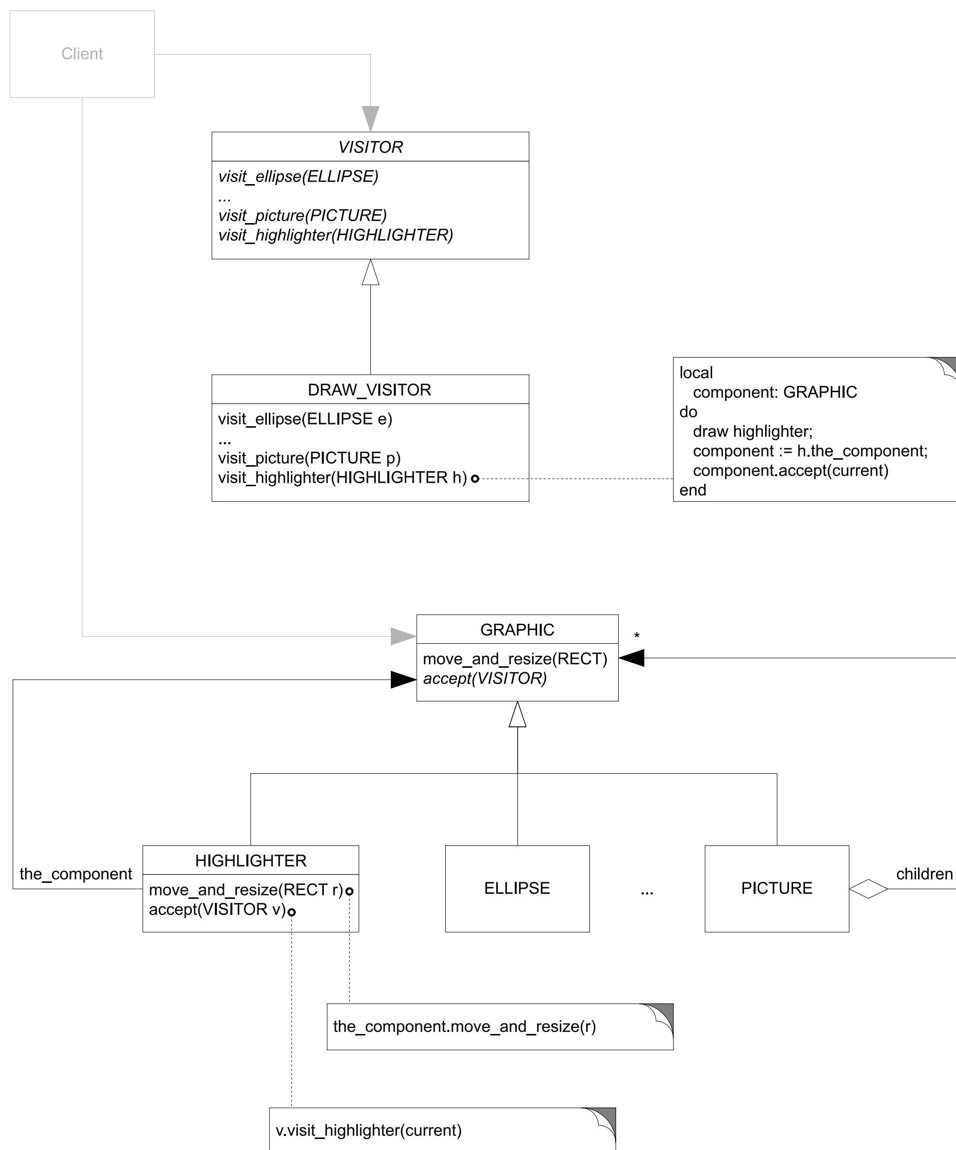


Abbildung 3.7: Das grafische Kompositum mit Dekorierer

Die beschriebenen Komponenten mit den Verantwortlichkeiten der entsprechenden Objekte werden in [3] im Design Pattern *Dekorierer* zusammengefasst (siehe auch Dekorierer (36)). Abbildung 3.7 zeigt die Kombination aus grafischem Kompositum und *Dekorierer*. Am Beispiel der Methode `move_and_resize` wird demonstriert, wie Methodenaufrufe zum dekorierten Objekt delegiert werden. Zusätzlich wird in der Komponente `VISITOR` die Methode `visit_highlighter` deklariert und in erbenden Besuchern entsprechend implementiert. Zum Beispiel ist die Implementation von `visit_highlighter` in der Komponente `DRAW_VISITOR` für das eigentliche Zeichnen der Markierung einer Grafik zuständig. Damit auch die Grafik selbst gezeichnet wird, muss der Besucher nach dem Zeichnen der Markierung auch auf das Objekt der Grafik angewandt werden. Analog zum Ein- bzw. Auslagern von Funktionalität kann man hier von *innerer* (`move_and_resize`) bzw. *äußerer* (via Besucher) Delegation sprechen.

Es ist somit möglich, Dekorierer in das grafische Kompositum aufzunehmen. Eine solche Situation wird in Abbildung 3.8 dargestellt. Teilbild *a)* zeigt eine typische Dokumentstruktur,

Teilbild *b)* stellt das Dokument grafisch dar. Die Dokumentstruktur enthält zwei Markierer auf oberster Ebene, welche ihrerseits zwei Grafiken markieren.

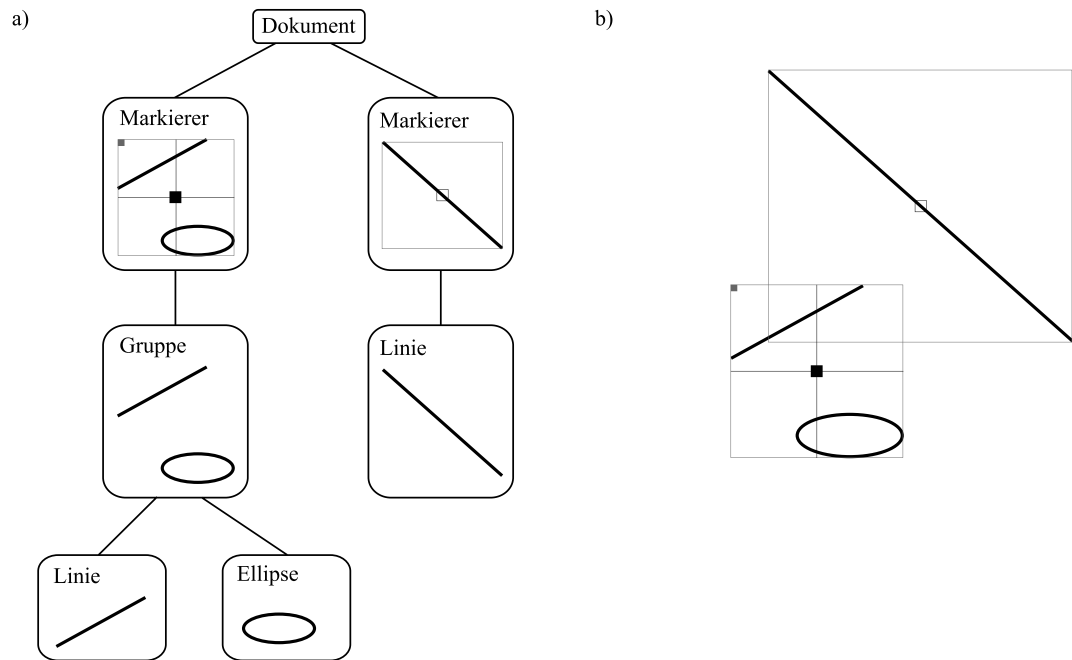


Abbildung 3.8: Markierer im grafischen Kompositum

3.3.5 Realisierung von Verbindern

Beim Zeichnen komplexerer Grafiken ist es oftmals von Vorteil, Grafiken logisch zusammenfassen zu können, da sich hierdurch nachfolgende Aktionen einfacher anwenden lassen und so eher der Intention des Benutzers folgen. Neben dem Gruppieren von Grafiken ist es in *DrawIt* auch möglich, zwei Grafiken (*Grundgrafiken*) über eine dritte Grafik (*Verbundgrafik*) miteinander zu *verbinden.*

Im Gegensatz zur Gruppierung ist das Ergebnis des Verbindens keine Gruppe, d.h. die teilnehmenden Grafiken bleiben als solche erhalten. Das Wesen einer Verbindung von Grafiken besteht nun für *DrawIt* darin, auf nachfolgende Positions- und Größenänderungen sinnvoller und intelligenterer reagieren zu können. Zur Demonstration soll Abbildung 3.9 dienen. In Teilbild *a)* ist eine Situation dargestellt, die häufig beim Zeichnen von Klassendiagrammen auftritt. Drei Rechtecke, als Sinnbild von Klassen, stehen in einer Beziehung zueinander, was durch die jeweiligen senkrechten Linien ausgedrückt werden soll. Der Benutzer entscheidet sich an dieser Stelle, jeweils zwei Rechtecke durch die entsprechende Linie logisch zu verbinden. In der weiteren Entwicklung der Zeichnung stellt der Benutzer nun fest, dass es sinnvoller wäre, das mittlere Rechteck weiter rechts anzuordnen. Da er dieses Rechteck vorher über die Linien mit den anderen beiden Rechtecken verbunden hat, werden bei der Positionänderung auch die Verbundlinien so angepasst, dass sie noch immer die drei Rechtecke verbinden (Teilbild *b)*). Dies funktioniert natürlich auch, wenn das unterste Rechteck ebenfalls bewegt wird (Teilbild *c)*).

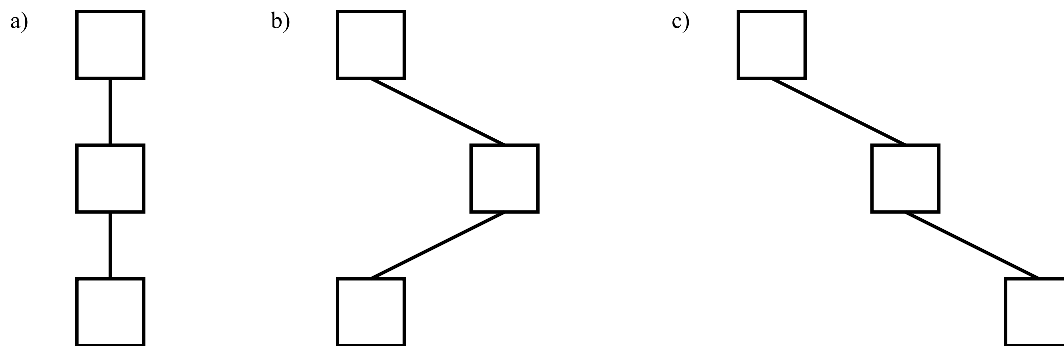


Abbildung 3.9: Verbinder in *DrawIt*

Zunächst wird die Komponente `CONNECTER` eingeführt. `CONNECTER` implementiert die Funktionalität des Verbinders. Objekte dieser Komponente dekorieren Objekte vom Typ `GRAPHIC`, d.h. `CONNECTER` erbt von `GRAPHIC` und enthält eine Referenz auf die dekorierte Grafik. Im konkreten Fall dekoriert ein Verbinderobjekt das Objekt der Verbundgrafik.

Der gewählte Ansatz zur Implementation des Verbinders stellt neben der Realisierung von Markierern (siehe Abschnitt 3.3.4) somit die zweite Anwendung des Design Patterns *Dekorierer* dar. Das Design Pattern *Dekorierer* ist ein sogenanntes *Strukturmuster*, d.h. das Wesen des Design Patterns liegt in seiner Komponentenstruktur. Das Verhalten spielt im *Dekorierer* hingegen eine untergeordnete Rolle. Das Verhalten des Markierers wurde gänzlich durch die Anwendung selbst und nicht durch das Design Pattern *Dekorierer* bestimmt.

Das Verhalten des Verbinders ist jedoch weitaus komplexer. Im Folgenden soll dies anhand der Interaktionen mit seinen Grundgrafiken und der Verbundgrafik begründet werden. Auf Objektebene besitzt ein Objekt vom Typ `GRAPHIC` einen Zustand, der Position und Größe beinhaltet. Wird dieser Zustand nun durch den Benutzer indirekt oder direkt verändert, so müssen Aktionen ausgelöst werden, die für die Anpassung von Position

und Form der mit dieser Grafik assoziierten Verbundgrafiken zuständig sind. Sowohl die Grundgrafiken als auslösende Elemente, als auch die Verbundgrafiken als angestoßene Elemente einer Aktualisierung müssen auf Implementierungsebene entsprechend erweitert werden.

Das Design Pattern *Dekorierer*

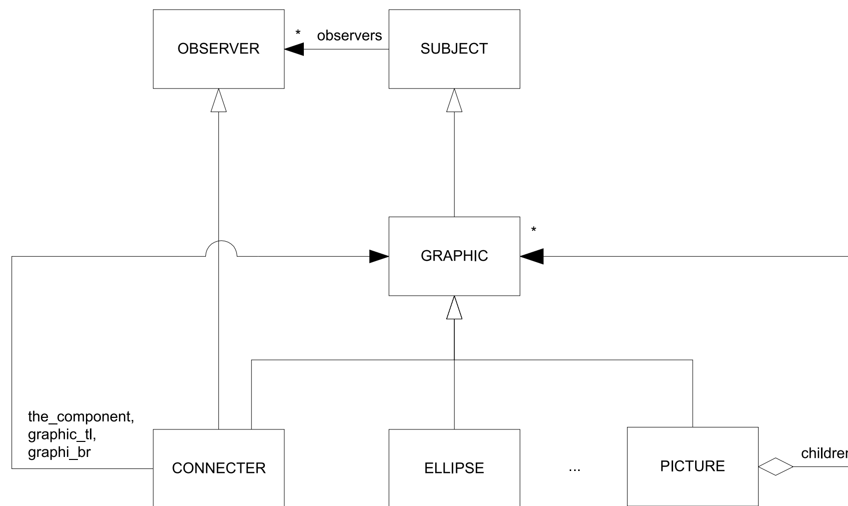


Abbildung 3.10: Verbinder im grafischen Kompositum

Die skizzierte Funktionalität wird nun durch das Design Pattern *Beobachter* (siehe auch Beobachter (60)) implementiert. Der *Beobachter* gehört zur Klasse der *Verhaltensmuster*, d.h. neben einer bestimmten Komponentenstruktur ist zusätzlich das Verhalten der Teilnehmer von Bedeutung. Das Design Pattern *Beobachter* wird nun folgendermaßen in die Anwendung übertragen. Das Verhalten des Verbinders sollte vom Verhalten der Komponente `CONCRETE_OBSERVER` im *Beobachter* geprägt sein, die Komponente `GRAPHIC` sollte sich analog zu `CONCRETE_SUBJECT` im *Beobachter* verhalten. Die hieraus resultierende Erweiterung der Struktur des grafischen Kompositums ist in Abbildung 3.10 dargestellt.

Abbildung 3.11 zeigt die entsprechende Objektstruktur der Grafiken in Abbildung 3.9 als dynamische Ausprägung der oben beschriebenen Komponentenstruktur. Neben der Struktur sind außerdem die Assoziationen eingezeichnet, die die Verbinderobjekte mit den Grafikobjekten eingehen (gestrichelte Linie). Wird die Position und/oder die Größe einer Grafik geändert, so benachrichtigt das entsprechende Grafikobjekt alle Verbinderobjekte in seiner `observers` Assoziation. Ein Verbinderobjekt reagiert nun, indem es zunächst den Zustand der Grafikobjekte der Grundgrafiken erfragt (über `graphic_tl` und `graphic_br`) und anschließend die Position und Größe der Verbundgrafik anpasst.

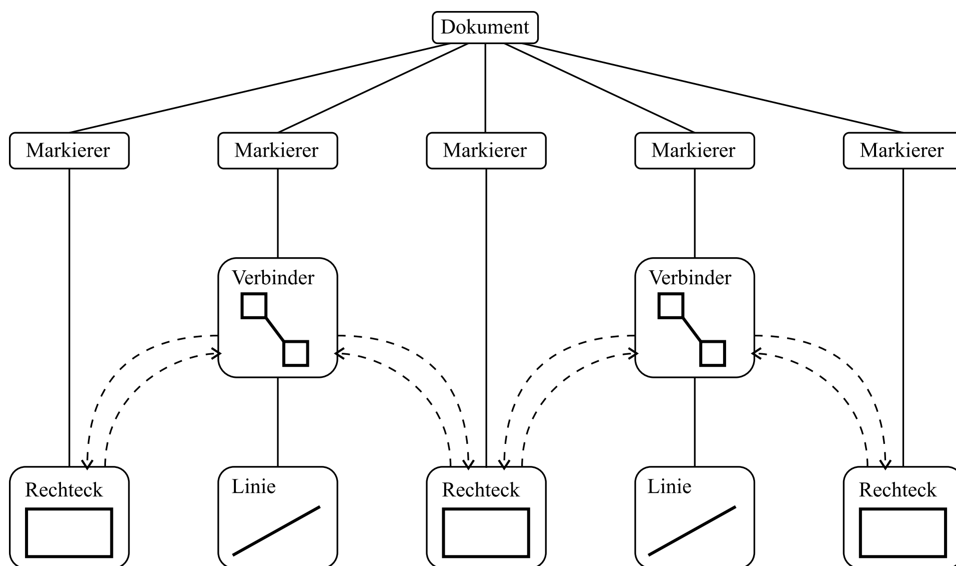


Abbildung 3.11: Dokumentstruktur aus Abbildung 3.9

3.3.6 Kommandos

Ein Dokument in *DrawIt* wird durch die Interaktionen zwischen dem Benutzer und *DrawIt* ständig verändert. Grafiken werden hinzugefügt oder gelöscht, Eigenschaften von Grafiken werden verändert, etc. Wenn der Benutzer mit der Maus auf einen Knopf in der Werkzeugleiste klickt oder einen Menüpunkt auswählt, also eine Aktion anstößt, so reagiert *DrawIt* entsprechend. Abstrakt betrachtet, formuliert der Klient ein *Kommando*, welches *DrawIt* dann ausführt.

Kommandos sollen den folgenden Verantwortlichkeiten genügen:

1. Ein Kommando muss ausgeführt werden können. Außerdem sollen alle relevanten Daten eines Kommandos in adequater Weise im Kommando selbst verwaltet werden.

2. Ein Kommando wird vom Klienten formuliert (oder auch erstellt) und danach ausgeführt. Erst das Ausführen des Kommandos bewirkt die gewünschte Änderung im Dokument. Nach dem Erstellen eines Kommandos muss es möglich sein, dieses Kommando bei gleichem Systemzustand beliebig oft auszuführen. Dies bedeutet, dass die Ausführung des Kommandos das Kommando selbst nicht verändern darf.

3. Kommandos verändern den Zustand der im Dokument enthaltenen Grafiken in beliebiger Weise. Darüber hinaus sollen sie jedoch keine Wirkung haben. Zum Beispiel soll es für ein Kommando nicht möglich sein, den aktuellen Bildausschnitt zu scrollen oder das Programmfenster zu bewegen.

Das Design Pattern *Kommando*

Hieraus lässt sich folgende grundlegende Komponentenstruktur ableiten, die der des Design Patterns *Kommando* entspricht. Kommandos werden durch eine konkrete Komponente implementiert die von der abstrakten Komponente `COMMAND` erbt. `COMMAND` repräsentiert eine gemeinsame Schnittstelle für alle Kommandos. Diese Schnittstelle beinhaltet die Methode `execute`, die das Kommando letztendlich ausführt. Hierdurch können alle Kommandos homogen behandelt werden. Zusätzlich definiert `COMMAND` das Attribut `the_receiver`, welches den Empfänger, in diesem Fall ein Grafikobjekt, referenziert. Auf diese Weise kann das konkrete Kommando einfach auf das Dokument und die darin enthaltenen Grafiken zugreifen.

Die Kombination von *Kommando* und dem grafischen Kompositum

Kommandos sollen nun möglichst effektiv mit dem grafischen Kompositum gekoppelt werden. Das grafische Kompositum stellt Funktionalität zur Verfügung, die von Kommandos genutzt werden sollen. Prinzipiell existieren zwei alternative Ansätze zum Erreichen dieses Zieles:

1. Die Funktionalität des grafischen Kompositums wird im Kommando benutzt. Die Methode `execute` wird so implementiert, dass von dort Methoden des grafischen Kompositums aufgerufen werden. Diese Vorgehensweise ist geeigneter, wenn das Kommando nicht oder nur auf einfache Art und Weise auf bestehende Grafiken zugreift.

2. Das Kommando erzeugt in `execute` dynamisch einen Besucher für das grafische Kompositum (siehe auch Abschnitt 3.3.3) über das Design Pattern *Fabrikmethode* (siehe auch Fabrikmethode (17)). Dieser Besucher ist speziell auf das Kommando zugeschnitten. Durch die Nutzung des Besucherkonzepts ist es möglich, während der Kommandoausführung die Elemente des grafischen Kompositums in entsprechender Weise zu traversieren.
   Beispiel: Soll ein Kommando nur auf markierten Elemente einen Effekt besitzen, so kann im Besucher die Methode `visit_highlighter` so implementiert werden, dass der

Besucher nur bei markierter Grafik an die Grafik selbst weitergerreicht wird.

In *DrawIt* werden beide Ansätze benutzt. Dabei wird im speziellen Fall nach den oben genannten Kriterien entschieden, da beide Varianten verträglich miteinander sind. Tabelle 3.2 listet alle verwendeten Kommandos auf. Neben einer Bedeutung des jeweiligen Kommandos wird außerdem ein Indikator über die Verwendung eines entsprechenden Besuchers geführt.

Kommando	**Besucher**	**Bedeutung**
`MARK_COMMAND`	ja	markiert Grafiken
`GROUP_COMMAND`	ja	gruppiert markierte Grafiken
`UNGROUP_COMMAND`	ja	degruppiert markierte Gruppen
`DELETE_COMMAND`	ja	löscht markierte Grafiken
`NEW_LINE_COMMAND`	nein	erstellt eine neue Linie
`NEW_RECTANGLE_COMMAND`	nein	erstellt ein neues Rechteck
`NEW_ELLIPSE_COMMAND`	nein	erstellt eine neue Linie
`RESIZE_COMMAND`	nein	verändert Position und Größe der markierten Grafik
`CONNECT_COMMAND`	nein	verbindet markierte Grafiken
`DISCONNECT_COMMAND`	ja	löst markierte Verbinder von deren Grundgrafiken
`CONF_CONNECT_COMMAND`	nein	konfiguriert den markierten Verbinder
`LOCK_GRAPHIC_COMMAND`	ja	sperrt / entsperrt Eigenschaften der markierten Grafik

Tabelle 3.2: Kommandos in *DrawIt*

Kommandos, also Objekte vom Typ `COMMAND`, werden in einer Befehlskette verwaltet (*Chain of Command*). Diese Befehlkette wird durch das Patternattribut `the_chain_of_command` referenziert. Führt der Benutzer entweder *Aktion rückgängig* oder *Aktion wiederherstellen* aus, so navigiert *DrawIt* auf dieser Befehlskette rückwärts bzw. vorwärts.

Kommerzielle Systeme bedienen sich zweier Strategien, um eine Aktion (in diesem Kontext ein Kommando) rückgängig machen zu können:

1. Für jedes Kommando wird der vorherige Zustand des Systems gespeichert. Wird nun eine Aktion rückgängig gemacht, so wird jeweilige abgespeicherte Zustand wieder aktiviert und so zum aktuellen Zustand. Diese Vorgehensweise ist in der Regel einfach zu implementieren, sie ist jedoch nicht effizient, da oftmals unnütze Information abgespeichert wird (unveränderte Grafiken).

2. Jedes Kommando besitzt eine Methode `undo`, die diese Aktion rückgängig macht. `undo` kann die erstgenannte Strategie implementieren. In vielen Fällen ist es jedoch adäquater, mittels eines im Kommando abgespeicherten Deltas die Aktion rückgängig zu machen.

In *DrawIt* wird jedoch ein, zugegebenermaßen, ineffizienter Weg beschritten, der keine der beiden Strategien benutzt. Ein *Aktion rückgängig* wird nun wie folgt realisiert. Zunächst wird das gesamte Dokument gelöscht wird. Danach wird die gesamte Befehlskette bis zum vorletzten Befehl *abgespielt*[4]. Da kompliziertere Algorithmen keinen demonstrativen Nutzen im Sinne dieser Arbeit besitzen, wird hier auf diesen Ansatz, dessen Implementationsaufwand am geringsten ist, zurückgegriffen.

Abbildung 3.12 zeigt die Kombination aus Befehlskette, Kommandos und grafischem Kompositum.

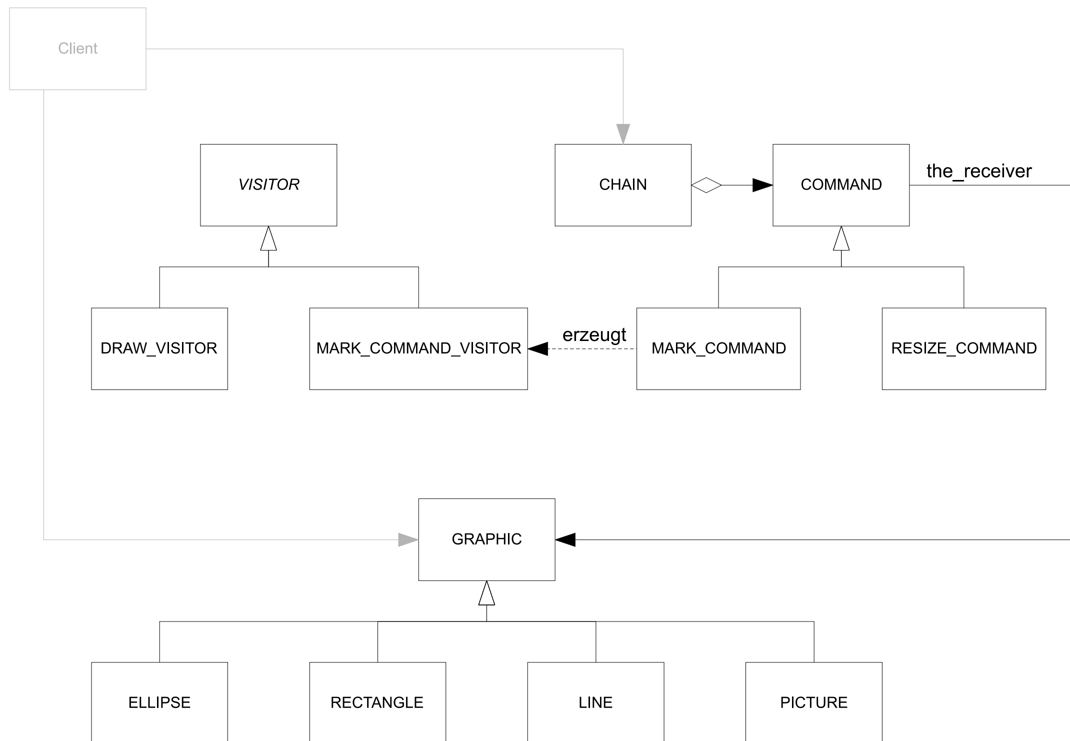


Abbildung 3.12: Kombination aus Befehlskette, Kommandos und grafischem Kompositum

[4] *Abspielen* bedeutet, dass die Kommandos eines Teils oder der gesamten Befehlskette sequentiell ausgeführt werden.

3.3.7 Benutzungsschnittstelle

Dieser Unterabschnitt befasst sich im Folgenden mit der Integration der Benutzungsschnittstellenimplementierung in `PDOCUMENT_WINDOW`. Dabei soll im Besonderen die Kombination von objektorientierter Benutzungsschnittstellenprogrammierung und patternorientierter Programmierung im Vordergrund stehen.

`PDOCUMENT_WINDOW` als Fensterklasse

Zur Programmierung der Benutzungsschnittstelle von *DrawIt* in *PaL* müssen zunächst Mittel und Mechanismen zur Verfügung stehen, die es ermöglichen, mit einem Fenstersystem, wie z.B. *Microsoft Windows*, zu kommunizieren. Diese Mittel werden oft unter dem Namen *API (application programming interface)* zusammengefasst. In modernen objektorientierten Programmiersprachen wird diese (imperative) API von einer Klassenbibliothek gekapselt. Die gesamte Programmierung der Benutzungsschnittstelle kann nun unter Ausnutzung des objektorientierten Methoden erfolgen.

Die Implementation von *DrawIt* erfolgt in *PaL*. Der *PaL*-Compiler übersetzt den *PaL*-Quelltext nach *Eiffel*. Für *Eiffel* ist eine Klassenbibliothek zur Benutzungsschnittstellenprogrammierung namens *WEL (windows eiffel library)* erhältlich. Natürlich gibt es keine solche eine Bibliothek für *PaL*. Die Sprache *PaL* besitzt jedoch einige Zusatzkonstrukte, mit denen der Benutzer im *PaL*-Quelltext Eiffel-Klassen benutzen kann (siehe dazu Abschnitt A.1).

In *WEL* wird das Aussehen und das Verhalten (*Look & Feel*) einer Klasse von Fenstern von einer Eiffelklasse implementiert. Die Klasse `WEL_WINDOW` ist dabei der Ausgangspunkt für eigene Erweiterungen. Ein Objekt dieses Typs repräsentiert ein Fenster auf Programmebene.

WEL ist ereignisorientiert konzipiert. Ereignisse des Fenstersystems werden aufbereitet und in Methodenaufrufe des entsprechenden Fensterobjektes umgesetzt. `WEL_FRAME_WINDOW` stellt zu diesem Zweck eine Schnittstelle zur Verfügung, die z.B. `on_paint` zum Neuzeichnen des Fensters oder auch `on_mouse_move` zur Behandlung von Mausbewegungen enthält. Diese Methoden müssen in den Klassen der Anwendung (also *DrawIt*) entsprechend überschrieben werden, um so die Reaktionen auf Ereignisse des Fenstersystems zu definieren.

Bei der in [1] und [4] patternorientiertem Modell kann man eine Klasse im konventionellen objektorientiertem Sinn als Design Pattern ohne Komponenten auffassen. Die umgekehrte Implikation gilt ebenfalls. Es ist also konzeptionell möglich, das Applikationspatterns `PDOCUMENT_WINDOW` von der Klasse `WEL_FRAME_WINDOW` erben zu lassen. Auf Implementationsebene kann hierfür nicht auf das `refines`-Konstrukt zurückgegriffen werden, da `WEL_FRAME_WINDOW` eine Eiffel-Klasse ist. `WEL_FRAME_WINDOW` muss in der Implementation von `PDOCUMENT_WINDOW` mittels der `import`-Klausel dem *PaL*-Compiler bekannt gemacht werden. Dadurch erbt `PDOCUMENT_WINDOW` automatisch von `WEL_FRAME_WINDOW`. Methoden in `WEL_FRAME_WINDOW` können nun von Patternmethoden in `PDOCUMENT_WINDOW` überschrieben werden. Nach der Übersetzung verhalten sich dann die Eiffel-Klassen des Design Patterns, dessen Methoden und deren Implementationen entsprechend dem Subclassing-Grundgedanken der Anwendung von *WEL*.

Die Behandlung von Mausereignissen

Eine spezielle Anwendung des obigen Ansatzes in *DrawIt* ist die Behandlung von Mausereignissen. `WEL_FRAME_WINDOW` definiert die Methoden `on_left_button_down`, `on_left_button_up` und `on_mouse_move` durch eine Standardimplementation, die dann in `PDOCUMENT_WINDOW`

überschrieben wird.

Zur Behandlung von Mausereignissen wird ein *Listener*-Konzept[5] implementiert. Es wird eine abstrakte Komponente `MOUSE_LISTENER` eingeführt, die ebenfalls die Methoden `on_left_button_down`, `on_left_button_up` und `on_mouse_move` definiert. Konkrete Listener implementieren dann ein spezielles Verhalten. Zusätzlich wird das Patternattribut `the_mouse_listener` eingeführt. Die Mausbehandlungsmethoden des Design Patterns werden so implementiert, dass sie alle Aufrufe zu dem *Listener*-Objekt delegiert, der momentan durch `the_mouse_listener` referenziert wird. Durch diese Vorgehensweise kann man die Reaktion des Programms sehr effizient an Zustandsänderungen anpassen. In *DrawIt* wird in der Mausbehandlung z.B. zwischen den Zuständen *momentan wird markiert* und *momentan wird Rechteck gezeichnet* unterschieden. Wird nun in den Markierungsmodus gewechselt, so wird auf Programmierebene ein neuer aktueller Listener gesetzt.

Der beschriebene Ansatz ist eine Anwendung des Design Patterns *Zustand* (siehe auch Zustand (50)). Leider kann hier die Implementation dieses Design Patterns nicht durch Verfeinerung wiederverwendet werden, da die Implementation nur auf Komponentenebene arbeiten. Die Komponente `CONTEXT` ist im Fall des Listener jedoch das Design Pattern selbst.

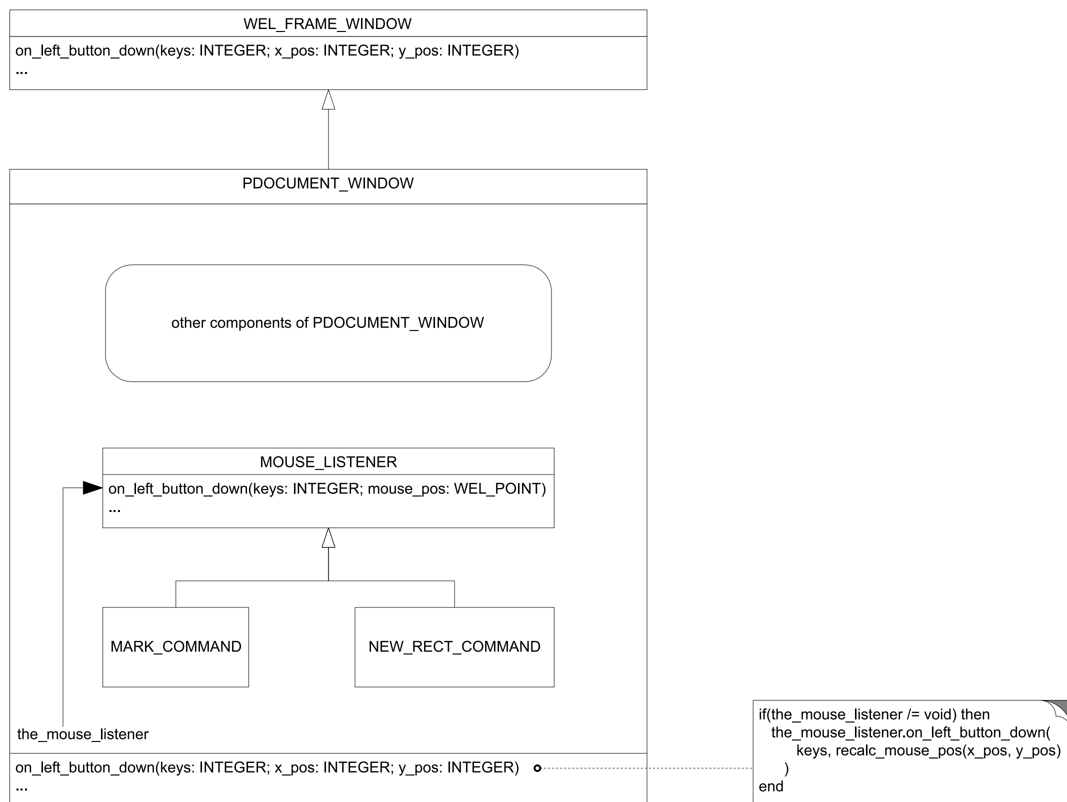


Abbildung 3.13: Die Verwendung der *WEL* am Beispiel der Mausereignisbehandlung

Abbildung 3.13 zeigt die Verwendung der *WEL* am Beispiel der Mausereignisbehandlung. An dieser Stelle angemerkt, dass einige der eingeführten Kommandos aus Abschnitt 3.3.6, wie z.B. `MARK_COMMAND` und `NEW_RECT_COMMAND`, gleichzeitig konkrete Listener sind. Wenn der Benutzer z.B. in den Markierungsmodus wechselt, so wird ein Objekt der Komponente `MARK_COMMAND` erzeugt. Die Behandlung von Mausereignissen wird nun komplett in dieses

[5]Die Namensgebung erfolgte analog zum Konzept des *Listeners* in der Programmiersprache *Java*.

Objekt verlagert. Durch weitere Mausklicks des Benutzers können nun die Grafiken auf effiziente Weise markiert und demarkiert werden.

3.4 Herleitung der Design Patterns der Applikation

Das Design Pattern der Applikation, also das Design Pattern PDOCUMENT_WINDOW soll nun hergeleitet werden. Dabei soll intensiv von der Patternbibliothek Gebrauch gemacht werden, die in Kapitel 2 vorgestellt wurde. Diese Patternbibliothek bildet als elementare Bausteine den Ausgangspunkt aller patternorientierten Verfeinerungen dieses Abschnittes. Die Verfeinerung, als das Mittel der Wiederverwendung, erweitert und konfiguriert Design Patterns zu neuen, komplexeren Design Patterns. Eine weitere wichtige Aufgabe der Verfeinerung ist die Kombination von Design Patterns. Dadurch können die Grundbausteine in Form der 23 Design Patterns aus [3] nach dem Baukastenprinzip in mehreren Schritten so kombiniert werden, dass das gesamte Applikationspattern durch patternoriente Mechanismen aus diesen grundlegenden Design Patterns hervorgeht.

Eine Verfeinerungshierarchie wird immer so aufgebaut sein, dass von abstrakten zu applikationsspezifischen Design Patterns verfeinert wird. Der Übergang von abstrakt zu applikationsspezifisch erfolgt dabei normalerweise in mehreren Schritten. In den ersten Schritten werden zunächst *GoF*-Design Patterns den Erfordernissen entsprechend kombiniert, ohne dass die resultierenden Design Patterns diese Verfeinerungen ihren abstrakten Charakter verlieren. Da diese ersten Schritte in nahezu jeder beliebigen Applikation durchgeführt werden müssen, ist es sinnvoll, bestimmte, häufig benutzte Kombinationen in die Standardbibliothek grundlegender Design Patterns aufzunehmen. In [3] wird ebenfalls auf solche Standardkombinationen hingewiesen. Eine Übersicht grundlegender Design Patterns, die in *DrawIt* Anwendung finden, wird in Tabelle 3.3 gegeben. Standardkombinationen dieser grundlegenden Design Patterns werden in Tabelle 3.4 aufgeführt, insofern die Implementation von *DrawIt* diese benutzt. Die Gesamtheit dieser Design Patterns stellen die Basis der weiteren Betrachtungen dar.

Design Pattern	**Genese**	**Bedeutung**
PPARAMETER	grundlegend	siehe Parameter (15)
PCONTAINER	grundlegend	siehe Container (11)
PFACTORY_METHOD	grundlegend	siehe Fabrikmethode (17) und [3]
PITERATOR	PCONTAINER, PFACTORY_METHOD	siehe Iterator (24) und [3]
PCOMPOSITE	PCONTAINER, PPARAMETER	siehe Kompositum (28) und [3]
PLIST	PCONTAINER	Implementation einer einfach verketteten Liste (siehe Liste (12))
PDECORATOR	grundlegend	siehe Dekorierer (36) und [3]
PVISITOR	grundlegend	siehe Besucher (44) und [3]
PCOMMAND	PPARAMETER	siehe Befehl (55) und [3]
POBSERVER	PCONTAINER	siehe Beobachter (60) und [3]

Tabelle 3.3: In *DrawIt* benutzte grundlegende Design Patterns

Die folgenden Unterabschnitte beschreiben die Herleitung bestimmter Design Patterns für die Anwendung in *DrawIt*. Jeder dieser Unterabschnitte ist in Motivation und Genese

Design Pattern	**Genese**	**Bedeutung**
PCOMPOSITE_ITERATOR	PCOMPOSITE, PITERATOR	Kompositum, in dem die Komponenten im Kompositum iteriert werden können (siehe Abschnitt 2.5)
PLIST_ITERATOR	PITERATOR, PLIST	Liste, in der die Elemente der Liste iteriert werden können (siehe Abschnitt 2.5)
PCOMPOSITE_LIST_ITERATOR	PLIST_ITERATOR, PCOMPOSITE_ITERATOR	Kompositum, in dem das Kompositum seine Komponenten in einer iterierbaren Liste verwaltet (siehe Abschnitt 2.5)
PCOMPOSITE_DECORATOR	PCOMPOSITE_LIST_ITERATOR, PDECORATOR	Kompositum, in dem ein zusätzliches Blatt ein Dekorierer ist, der Komponenten dekoriert
POBSERVER_LIST	PLIST_ITERATOR, POBSERVER	Observer, in dem die Observer eines Subjektes in einer Liste verwaltet werden

Tabelle 3.4: In *DrawIt* benutzte Standardkombinationen der grundlegenden Design Patterns aus Tabelle 3.3

eingeteilt. Die Genese beschreibt jeden für die Herleitung wichtigen Verfeinerungsschritt auf Patternebene. Am Ende dieses Prozesses steht das Applikationspattern PDOCUMENT_WINDOW.

3.4.1 Herleitung der allgemeinen besuchergestützten Befehlskette (PCHAIN_OF_COMMAND_2_VISITOR)

Motivation

Eine allgemeine besuchergestützte Befehlskette verwaltet Kommandos in einer Liste. Zusätzlich müssen Kommandos rückgängig gemacht und wiederhergestellt werden können (für Ansätze hierfür siehe auch Abschnitt 3.3.6). Die eigentliche Aktion, für die ein Kommando steht, soll im Endeffekt von einem Besucher ausgeführt werden. Die Befehlskette ist zu diesem Zeitpunkt jedoch noch sehr allgemein. So soll z.B. noch keine Komponentenstruktur festgelegt werden.

Genese

1. *Allgemeine Befehlskette* `PCHAIN_OF_COMMAND` *aus* `PCOMMAND` *und* `PLIST_ITERATOR` — In diesem Schritt wird das Design Pattern `PCOMMAND` so mit dem Design Pattern `PLIST_ITERATOR` kombiniert, dass die Liste nun Kommandos verwaltet. Über die Kommandos kann mittels des Listeniterators iteriert werden. Die Liste selbst wird zur Befehlskette erweitert, die nun einen Iterator zur Navigation in der Befehlskette enthält. Dieser dient außerdem zur Abspeicherung des aktuellen Kommandos. Des weiteren werden spezielle Methoden zur Kommandoausführung zur Verfügung gestellt. Hierzu gehört unter anderem die Methode `play_to_previous`, die die Befehlskette bis zum vorletzten Eintrag in der Befehlskette abspielt. Abbildung 3.14 zeigt diesen Schritt der Herleitung.

2. *Besuchergestützte Befehlskette* `PCHAIN_OF_COMMAND_VISITOR` *aus* `PCHAIN_OF_COMMAND`*,* `PVISITOR` *und* `PFACTORY_METHOD` — Die allgemeine Befehlskette wird mit dem allgemeninen Besucher mittels der Fabrikmethode so kombiniert, dass die Methode `execute` des Kommandos autmatisch einen dem Kommando entsprechenden Besucher erzeugt und diesen beim Empfänger (`RECEIVER`) akzeptieren lässt. Zu diesem Zweck verschmilzt die Komponente `PARAMETER` aus `PCHAIN_OF_COMMAND` mit der Komponente `VISITOR` aus `PVISITOR`. Der formale Parametertyp des Design Patterns `PCHAIN_OF_COMMAND` wird nun also zum abstrakten Besucher, der Empfänger wird zum abstrakten Element des Besuchers verfeinert. Außerdem wird die Methode `operation` des Empfängers in `PCHAIN_OF_COMMAND` mit der Methode `accept` in `PVISITOR` so verschmolzen, dass die korrekte Besuchsfunktion (in diesem Fall `visit_concrete_element` aufgerufen wird. Damit der korrekte konkrete Besucher im `execute` der konkreten Kommandokomponente erzeugt wird, erfolgt nun die zusätzliche Kombinierung mit dem Design Pattern `PFACTORY_METHOD`. Die Methode `get_parameter` wird dabei die Fabrikmethode, die in den konkreten Kommandos automatisch überschrieben wird. Abbildung 3.15 zeigt diesen Schritt der Herleitung.

3. *Allgemeine besuchergestützte Befehlskette* `PCHAIN_OF_COMMAND_2_VISITOR` *aus* `PCHAIN_OF_COMMAND_VISITOR` *und* `PVISITOR` — Die besuchergestützte Befehlskette `PCHAIN_OF_COMMAND_VISITOR` wird nun ein weiteres Mal mit dem Besucher `PVISITOR` kombiniert, um die Komponente `CONCRETE_VISITOR` zu duplizieren. Die resultierenden Komponenten sind `CONCRETE_VISITOR` und `CONCRETE_COMMAND_VISITOR`. Mit `CONCRETE_COMMAND_VISITOR` existiert nun ein konkreter Besucher, der später zusammen mit dem `CONCRETE_COMMAND` dupliziert werden kann, ohne dass dabei ein *einfacher* konkreter Besucher, wie er zum Beispiel beim Neuzeichnen von Grafiken in *DrawIt* benötigt wird, geopfert werden muss. Es wird also eine Trennung zwischen einfachem und kommandogeneriertem konkreten Besucher vollzogen.

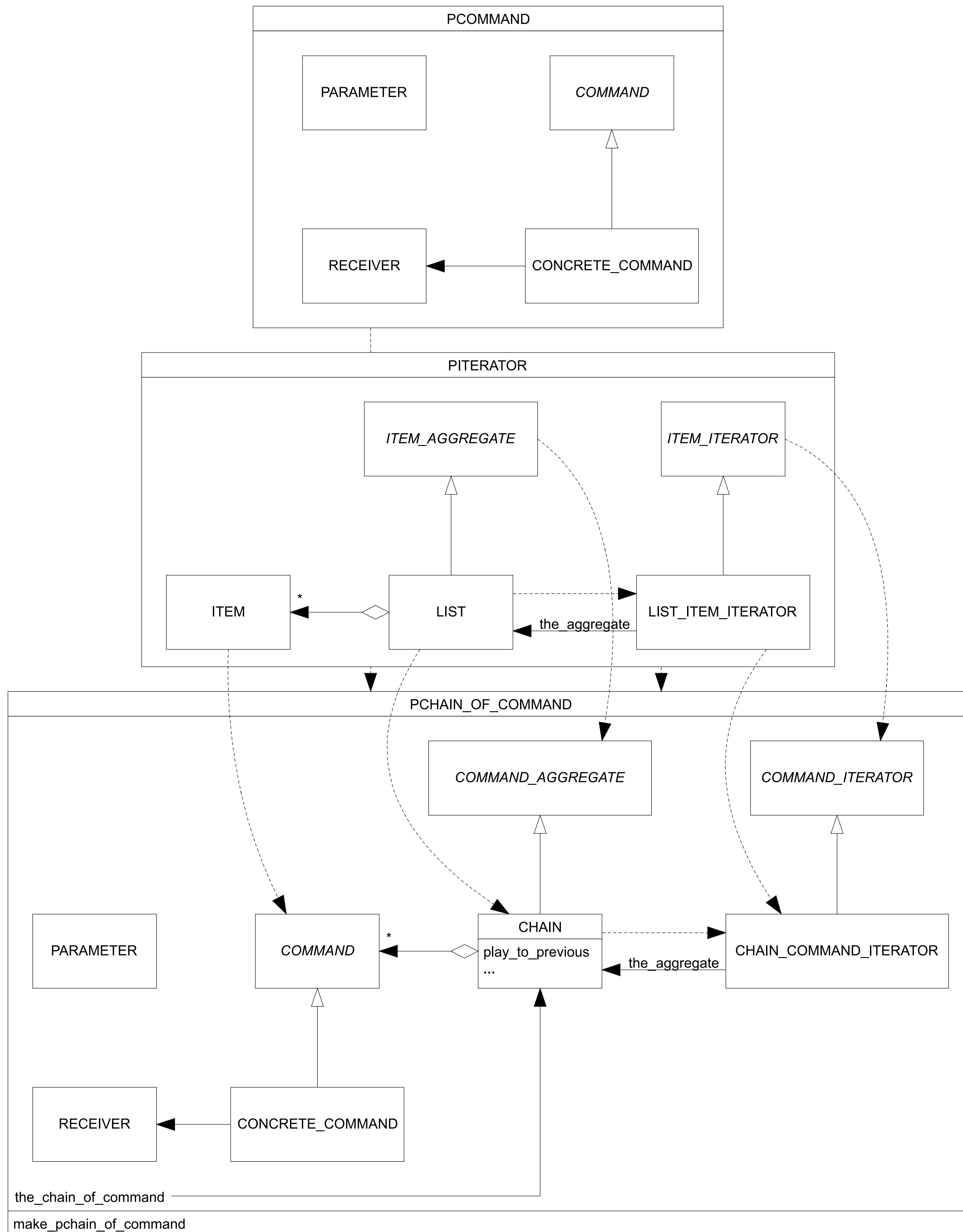


Abbildung 3.14: Die Verfeinerung von PCOMMAND und PLIST_ITERATOR zu PCHAIN_OF_COMMAND

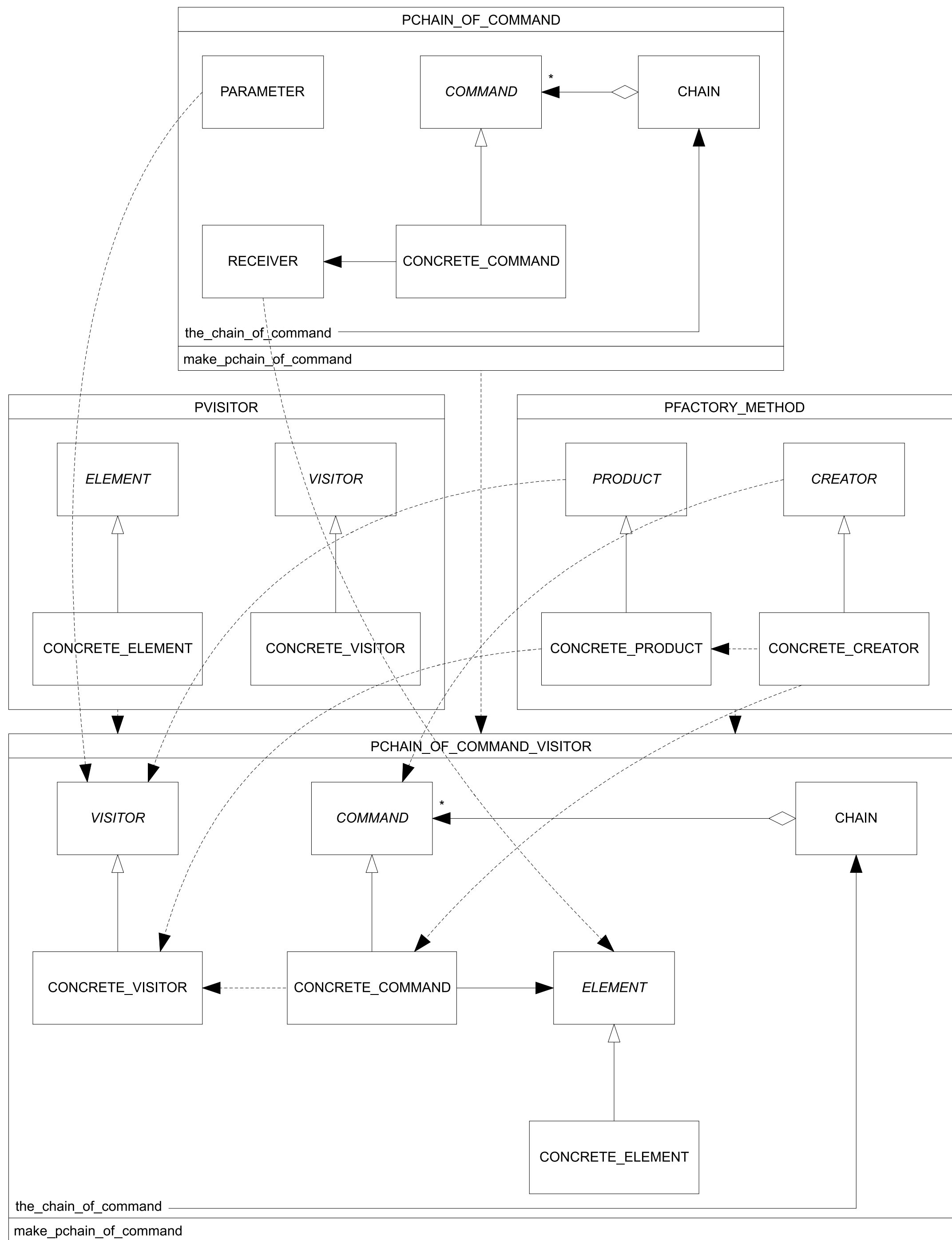


Abbildung 3.15: Die Verfeinerung von PCHAIN_OF_COMMAND, PVISITOR und PFACTORY_METHOD zu PCHAIN_OF_COMMAND_VISITOR (ohne Iteratoren)

3.4.2 Herleitung des abstrakten grafischen Kompositums (PABSTRACT_GRAPHIC_COMPOSITE)

Motivation

In diesem Schritt soll nun die *abstrakte* Struktur des späteren Applikationspatterns festgelegt werden. Das Attribut *abstrakt* wird verwendet, da diese Struktur als Basis für jedes Zeichenprogramm dienen könnte. Dabei ist es unerheblich, welche grafische Grundobjekte verwendet, welche Kommandos implementiert und welche konkrete Benutzungsschnittstelle im Endeffekt unterstützt werden soll. Aus diesem Grund werden z.B. Prototypen für konkrete grafische Komponenten (LEAF), konkrete Besucher (CONCRETE_VISITOR und CONCRETE_COMMAND_VISITOR) und konkrete Kommandos (CONCRETE_COMMAND) als Schablonen für applikationsspezifische Erweiterungen bereitgestellt.

Genese

1. *Kombination aus erweitertem Kompositum (Kompositum mit Dekorierer) und der allgemeinen besuchergestützten Befehlskette* PCOMPOSITE_DECORATOR_VISITOR *aus* PCHAIN_OF_COMMAND_2_VISITOR *(dreifach) und* PCOMPOSITE_DECORATOR — Gegenstand dieser Verfeinerung ist im Besonderen die Kombination des Besuchers aus der Befehlskette mit den Komponenten des Kompositums. Zu diesem Zweck wird die Komponente ELEMENT aus PCHAIN_OF_COMMAND_2_VISITOR mit der Komponente COMPONENT aus PCOMPOSITE_DECORATOR verschmolzen. Die Komponente CONCRETE_ELEMENT aus PCHAIN_OF_COMMAND_2_VISITOR kann mit den Komponenten COMPOSITE, LEAF und DECORATOR aus PCOMPOSITE_DECORATOR assoziiert werden. Um alle Abhängigkeiten erhalten zu können, wird von PCHAIN_OF_COMMAND_2_VISITOR dreifach verfeinert, wobei CONCRETE_ELEMENT je einmal auf COMPOSITE, LEAF und DECORATOR abgebildet wird. Auf diese Weise wird die Methode visit_concrete_element für COMPOSITE, LEAF und DECORATOR dupliziert und entsprechend umbenannt. Abbildung 3.16 zeigt diesen Schritt der Herleitung.

2. *Abstraktes grafisches Kompositum* PABSTRACT_GRAPHIC_COMPOSITE *aus* PCOMPOSITE_DECORATOR *und* PCOMPOSITE_DECORATOR_VISITOR — An dieser Stelle erfolgt der Schritt zum applikationsspezifischen Design Pattern PABSTRACT_GRAPHIC_COMPOSITE. Zunächst werden zahlreiche Umbenennungen durchgeführt, die dem Anwendungsgebiet entsprechen. So wird z.B. die Komponente COMPONENT in GRAPHIC und die COMPOSITE in PICTURE umbenannt (in beiden Verfeinerungen). Zusätzlich wird die neue Komponente RECT implementiert, die die Verwaltung von rechteckigen Bereichen im Arbeitsbereich übernimmt. Die in Abschnitt 3.3.3 diskutierte Methode move_and_resize soll patternorientiert auf Basis der Methode operation in PCOMPOSITE_DECORATOR implementiert werden. Der Methode move_and_resize wird als Parameter ein Objekt vom Typ RECT übergeben. Die Komponente PARAMETER in PCOMPOSITE_DECORATOR wird aus diesem Grund auf die Komponente RECT, die Methode operation in PCOMPOSITE_DECORATOR wird auf die Methode move_and_resize abgebildet. Des weiteren wird die Komponente TOPLEVEL_PICTURE implementiert. Zur Laufzeit wird diese Komponente für jede Patterninstanz genau einmal instanziiert und danach vom Patternattribut the_toplevel_picture referenziert. Das in the_toplevel_picture enthaltene Objekt repräsentiert das Kompositum, welches sich im Dokument auf höchster Ebene befindet. Da dieses Kompositum die Funktionalität des *normalen* Kompositums erweitert, erbt TOPLEVEL_PICTURE objektorientiert von PICTURE. Abbildung 3.17 zeigt das applikationsspezifische Design Pattern PABSTRACT_GRAPHIC_COMPOSITE.

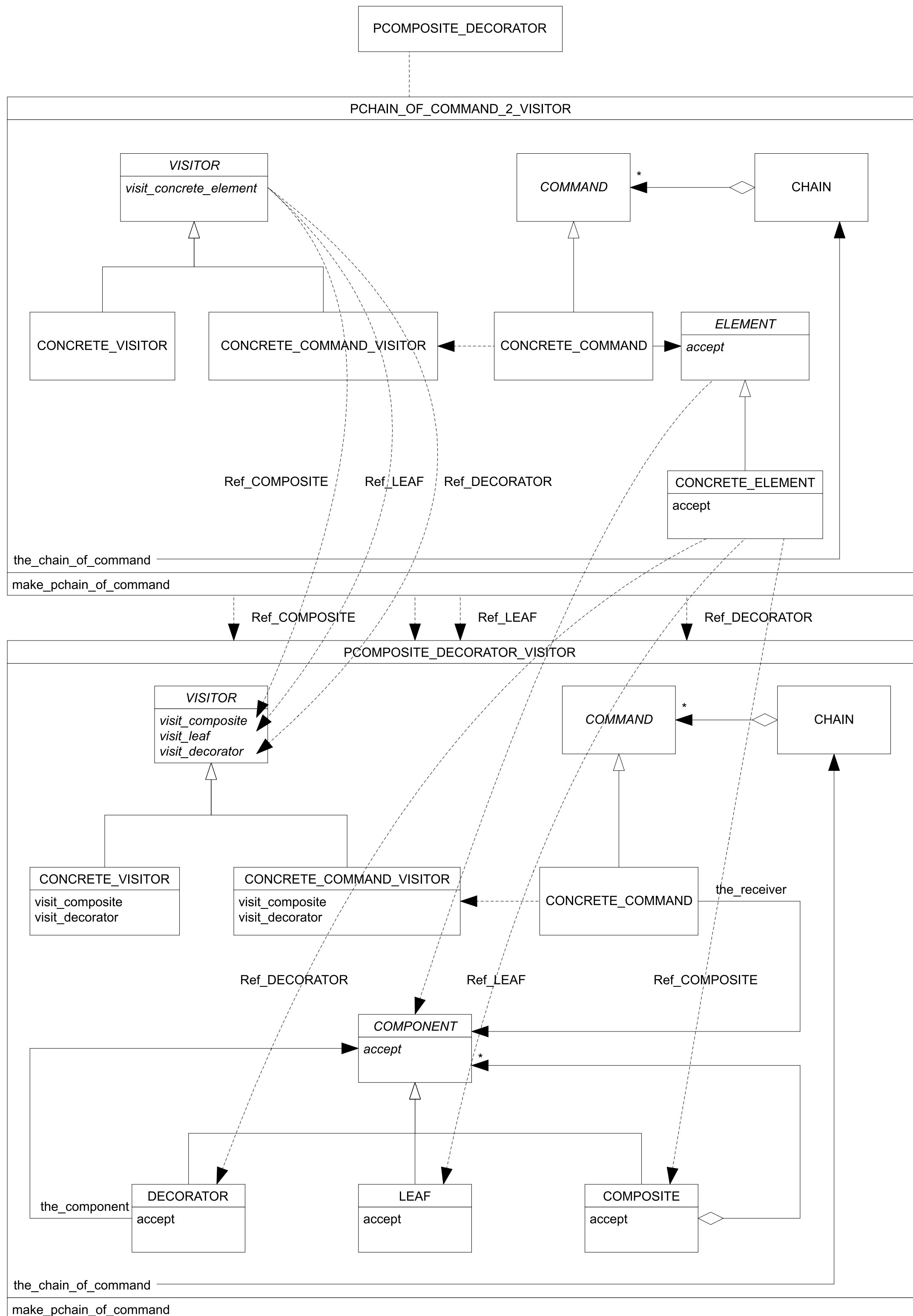


Abbildung 3.16: Die Verfeinerung von `PCHAIN_OF_COMMAND_2_VISITOR` (dreifach) und `PCOMPOSITE_DECORATOR` zu `PCOMPOSITE_DECORATOR_VISITOR` (ohne Iteratoren)

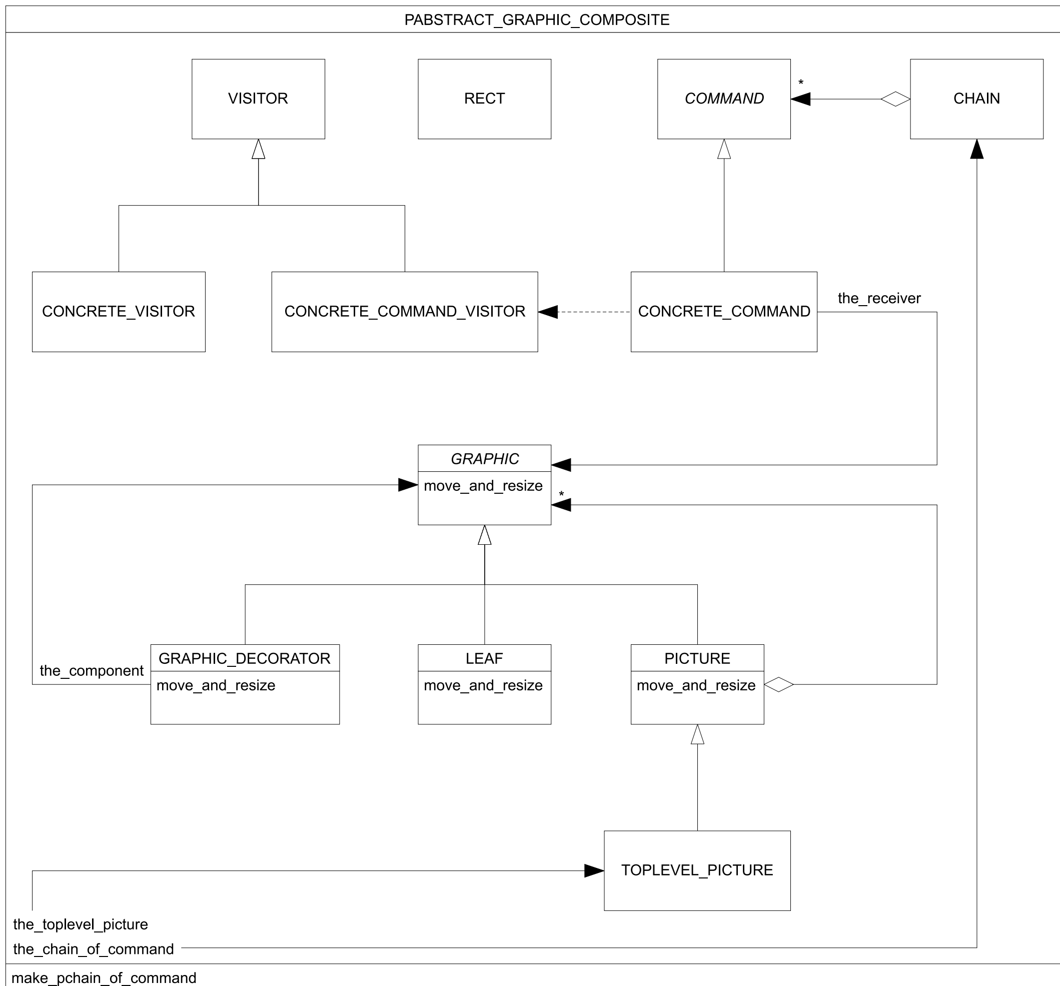


Abbildung 3.17: Das applikationsspezifische Design Pattern PABSTRACT_GRAPHIC_COMPOSITE (ohne Iteratoren)

3.4.3 Herleitung des Applikationspatterns (PDOCUMENT_WINDOW)

Motivation

Im vorigen Abschnitt wurde das abstrakte grafische Kompositum hergeleitet. In den in diesem Abschnitt beschriebenen Verfeinerungsschritten wird zum abstrakten grafischen Kompositum konkrete, applikationsspezifische Funktionalität hinzugefügt. Dabei ist jeder dieser Schritte so konzipiert, dass auch im Nachhinein Erweiterungen einfach möglich sind. Das Design Pattern PDOCUMENT_WINDOW steht am Ende der beschriebenen Verfeinerungshierarchie. Es wird zur Laufzeit des Programms instanziiert und ist somit ein Applikationspattern.

Genese

1. *Erzeugen konkreter Dekorierer (*PDECORATED_GRAPHIC_COMPOSITE*)* — Die Komponente GRAPHIC_DECORATOR wird dupliziert. Das Resultat dieser Duplizierung sind die Komponenten HIGHLIGHTER und CONNECTER und die entsprechenden Duplikate der Besuchermethode `visit_decorator`. HIGHLIGHTER ist verantwortlich für die Darstellung und die Verwaltung von Markierungen in *DrawIt* (siehe auch Abschnitt 3.3.4). Die Komponente CONNECTER wird für die Implementierung von Verbindern verwendet (siehe auch Abschnitt 3.3.5). Zu diesem Zeitpunkt besitzt CONNECTER jedoch noch keine Beobachterfunktionalität.

2. *Kombination mit dem Beobachter (*POBSERVED_GRAPHIC_COMPOSITE*)* — Das Design Pattern PDECORATED_GRAPHIC_COMPOSITE wird zweifach mit POBSERVER_LIST so kombiniert, dass CONNECTER zum konkreten Beobachter und GRAPHIC zum konkreten Subjekt wird (siehe auch Abschnitt 3.3.5). Dabei wird gleichzeitig das Attribut `the_subject` zu `graphic_tl` und `graphic_br` dupliziert, welche die Objekte der beiden Grundgrafiken als Subjekt refenzieren soll.

3. *Erzeugen konkreter Grafiken (*PGRAPHIC_COMPOSITE*)* — Die Komponente LEAF wird als Schablone zur Erzeugung der Komponenten LINE, RECTANGLE und CIRCLE benutzt. Gleichzeitig wird die Methode `visit_leaf` für LINE, RECTANGLE und CIRCLE dupliziert, um Besuchermethoden für die grafischen Blattkomponenten zu erhalten.

4. *Hinzufügen von Funktionalität (*PGRAPHIC_FUNCTIONALITY_COMPOSITE*)* — In diesem Schritt wird die Methode CONCRETE_VISITOR in DRAW_VISITOR umbenannt. Dieser Besucher ist in *DrawTt* für das Neuzeichnen der Grafiken zuständig (siehe auch Abschnitt 3.3.3). CONCRETE_VISITOR ist eine Schablone für jeden einfachen konkreten Besucher im grafischen Kompositum. Diese Komponente kann im nachträglichen Erweiterungsfall immer dann dupliziert werden, wenn Funktionalität aus dem grafischen Kompositum ausgelagert werden soll.

5. *Anbindung der* WEL *und Erzeugen konkreter Kommandos (*PDOCUMENT_WINDOW*)* — Neben der Verfeinerung von PGRAPHIC_FUNCTIONALITY_COMPOSITE wird hier von mehreren *WEL*-Klassen mittels des `import`-Konstruktes geerbt. In PDOCUMENT_WINDOW wird die Anbindung an die Benutzungsschnittstelle (hier *Windows 95/98/NT*) implementiert (siehe auch Abschnitt 3.3.7). Des weiteren werden die Kommandos mit den entsprechenden konkreten Besucher dupliziert. Auf diese Weise werden Paare von Kommando und Besucher erstellt die in Abschnitt 3.3.6 beschriebener Art und Weise miteinander interagieren.

Kapitel 4

Auswertung

In Kapitel 2 wurde eine Standardbibliothek grundlegender Design Patterns vorgestellt. Im Anschluss daran wurde diese Standardbibliothek in Kapitel 3 benutzt, um die Zeichenapplikation *DrawIt* zu implementieren. Dieses Kapitel wird nun hierbei aufgetretene Probleme benennen und mögliche Lösungen präsentieren, bevor diese Arbeit mit einem Resümee abschließt.

4.1 Spezielle Probleme und mögliche Lösungen

Patternwertige Komponenten

In einigen Situationen ist es hilfreich, in einem Pattern weitere Patterns zu schachteln. Dies entspricht einer Erweiterung des Begriffs der Komponente. Komponenten verhalten sich dann wie Patterns. Andererseits verhalten sich dann auch die geschachtelten Patterns wie Komponenten, da sie sich dann in einem übergeordnetem Pattern befinden. Auf diese Weise können die Begriffe Pattern und Komponente miteinander verschmolzen und damit auch othogonalisiert werden. Eine Vereinigung der Begriffe hat natürlich auch Konsequenzen auf die Sprache *PaL*. Es müsste ein Konstrukt eingeführt werden, welches erlaubt, Strukturen zu definieren, die andere Strukturen enthalten, die wiederum andere Strukturen enthalten, usw. Zusätzlich muss dann jedoch auch das Verfeinerungskonzept überarbeitet und verallgemeinert werden.

Als Beispiel einer solchen Situation, in der eine beliebige Schachtelungstiefe dem Entwickler von Vorteil sein würde, ist die Benutzungsschnittstellenprogrammierung zu nennen. In einem Fenstersystem enthalten Fenster andere Fenster. In *DrawIt* ist z.B. dem Programmfenster das Fenster einer Menüleiste und das Fenster für die eigentliche Darstellung des Dokumentes zugeordnet. Diese Verschachtelung kann sich beliebig fortsetzen, z.B. enthält das Dokumentfenster die Fenster für die Bildlaufleisten, usw. Bei der Benutzung einer Klassenbibliothek zur Benutzungsschnittstellenprogrammierung referenziert das Objekt des Elternfensters die der Kindfenster. Dieser Referenzierungsmechanismus ist in der objektorientierten Programmierung als adäquat anzusehen, im patternorientierten Modell ist es jedoch vorzuziehen, die Eigenschaft des Enthaltenseins auch wirklich durch die Schachtelung auf Patternebene auszudrücken.

Makromechanismen

Speziell bei der Implementation komplexerer Softwaresysteme sollte der Entwickler Design Patterns einsetzen. In *PaL* und dem zu Grunde liegendem Modell wird er zu diesem Schritt ermutigt. Dennoch ist es für ihn notwendig, um die Anwendung spezieller Patterns zu

wissen. Die beste Unterstützung von Seiten einer Standardbibliothek grundlegender Design Patterns nützt nur wenig, wenn der Benutzer die dort enthaltenen Patterns falsch einsetzt.

Um dieses Problem zu beheben, ist es vorstellbar, *PaL* dahingehend zu erweitern, dass sogenannte Standardanwendungen eines Pattern in die Definition dieses Patterns mit aufgenommen werden. Jede denkbare Standardanwendung wird durch einen Bezeichner identifiziert und durch eine Art Makro beschrieben. Bei Verfeinerungen von diesem Pattern kann dann dieser Bezeichner angegeben werden, wodurch das Pattern in einer bestimmten, im entsprechenden Makro festgelegten Weise verfeinert und dadurch angewandt wird.

Als Beispiel soll an dieser Stelle das Pattern *Kompositum* dienen. In seiner abstrakten Form enthält es die Blattkomponente `LEAF`. Bei der Benutzung des Kompositums wird im Normalfall in einem bestimmten Schritt der Verfeinerung diese Blattkomponente je nach den entsprechenden Erfordernissen dupliziert. Dies ist ein Standardanwendungsfall. Es wäre somit sinnvoll, die Verfeinerungsvorschrift für *Blatt duplizieren* mit in das Pattern *Kompositum* aufzunehmen. Bei der Anwendung des *Kompositums* können nun die Blätter beliebig oft durch die Verfeinerungsvorschrift *Blatt duplizieren* dupliziert werden.

Als weiteres Beispiel wird durch das Pattern *Liste* gegeben. *Liste* enthält die Komponenten `ITEM`, `ITEM_TEMPLATE` und `LIST`. Bei Verfeinerung dieses Patterns werden die Komponenten meist umbenannt, da Verfeinerungen oft konkreteren Charakter als die Ausgangspattern haben. In diesem Fall könnte die Liste benutzt werden, um Bücher in einem Bücherregal zu verwalten. Die Komponente `List` wird also z.B. in `SHELF` umbenannt, die Komponenten `ITEM` und `ITEM_TEMPLATE` werden in `BOOK` und `BOOK_TEMPLATE` umbenannt. Eine zweite Anwendung des Patterns *Liste* könnte dazu benutzt werden, die Angestellten in einem Buchgeschäft verwalten. Hierduch entstehen die Komponenten `BOOKSHOP`, `SALESPERSON` und `SALESPERSON_TEMPLATE`.
Es ist denkbar, dass beide Anwendungen der Liste im Endeffekt Teil eines konkreten Pattern zur Verwaltung eines Buchgeschäftes sind. Wird nun in beiden Anwendungen der Liste die Umbenennung der Komponente `ITEM_TEMPLATE` vergessen, so entsteht ein Konflikt der in einem schwer auffindbaren Fehler resultiert. Um Probleme dieser Art schon im Vorfeld zu vermeiden, könnte ein Standardanwendungsfall die Komponenten automatisch umbenennen. Falls der Entwickler `ITEM` in `BOOK` umbenennt, sorgt dann die Verfeinerungsvorschrift dafür, dass auch `ITEM_TEMPLATE` zu `BOOK_TEMPLATE` umbenannt wird.

Erweiterte Behandlung von Abhängigkeiten in der Vererbungshierarchie von Komponenten

Die Verfeinerung hat sich bei der Entwicklung von *DrawIt* als effektiv herausgestellt. In bestimmten Fällen kann die Benutzung des in *PaL* verwendeten Verfeinerungsmechanismus jedoch zu Problemen führen. Bei der Verfeinerung von mehreren Ausgangspatterns kann es vorkommen, dass die verfeinerte Komponentenhierarchie redundant wird, d.h. bestimmte Vererbungsbeziehungen sind syntaktisch herleitbar, aber dennoch als gesonderte Vererbung im verfeinerten Pattern enthalten. Dies ist jedoch nicht notwendig und in den vielen Fällen nicht erwünscht. Besonders die Wartbarkeit des verfeinerten Patterns wird hierdurch beeinträchtigt. Zur Lösung dieses Problems, kann der Verfeinerungsmechanismus so erweitert werden, dass er aus allen geforderten Vererbungsbeziehungen unter Komponenten die minimale Menge[1] von Vererbungenbeziehungen berechnet. Diese Menge muss die Eigenschaft haben, dass aus den dort enthaltenen Vererbungsbeziehungen alle geforderten Vererbungsbeziehungen impliziert werden können.

[1] *Minimal* bedeutet in diesem Fall: minimal bezüglich der Mächtigkeit der Menge.

Beispiel: In Pattern P_1 erbt die Komponente `B` von Komponente `A` und die Komponente `C` von Komponente `B`. In Pattern P_2 erbt die Komponente `C` von Komponente `A`. Bei einer Verfeinerung ohne Umbenennungen entsteht nun folgende Vererbungshierarchie:

$$\{B \ll A, C \ll B, C \ll A\}$$

Die Vererbungsbeziehung $C \ll A$ ist jedoch überflüssig, da sie aus den anderen beiden Vererbungen deduziert werden kann. Die verfeinerte Vererbungshierarchie könnte also optimiert werden:

$$\{B \ll A, C \ll B\}$$

4.2 Resümee

Das patternorientierte Paradigma hat sich bei der Entwicklung von *DrawIt* bewährt. An dieser Stelle ist es jedoch nicht möglich, hieraus zu schließen, dass diese Aussage für die Entwicklung jeglicher komplexer Software gilt. Dennoch kann man die gewonnenen Resultate als Indizien bewerten.

Für eine effektive Nutzung des patternorientierten Modells ist es notwendig, den Entwurfsprozess an die erweiterten Möglichkeiten des Modells anzupassen. Des weiteren ist es unumgänglich, die Funtionsweise und den Einsatz grundlegender Design Patterns genau zu kennen. Dies erfordert eine längere Einarbeitungsphase für den Entwickler, als es z.B. für die strukturierte Programmierung mit Benutzung von APIs nötig wäre.

Zusammenfassend lässt sich dennoch sagen, dass die Anwendung des patternorientierten Modells die Wiederverwendung einzelner Softwarekomponenten sowie die Wartbarkeit und Flexibilität der gesamten Implementation eines Programmes deutlich verbessert.

Anhang A

A.1 Erweiterung der Sprache *PaL* und des Compilers

Die Sprache *PaL* wurde in [1] für den SmallEiffel-Compiler entwickelt. Für die Realisierung eines größeren Projektes reichen die Möglichkeiten dieses Compilers nicht aus. Mit diesem Compiler ist keine Programmierung einer grafischen Benutzungsschnittstelle möglich. Daher wird für dieses Projekt der ISE-Eiffel-Compiler verwendet. Dieser bietet ein umfassendes Entwicklungswerkzeug, das zum Debuggen der generierten Eiffelklassen verwendet werden kann und Bibliotheken zur Programmierung von Benutzungsschnittstellen für Windows. Der Wechsel des Compilers hat einige kleine Erweiterungen des *PaL*-Compilers zur Folge.

deferred class

In Eiffel besteht die Möglichkeit, Methoden und Klassen mit Hilfe des Konstruktes `deferred` abstrakt zu implementieren. Wird eine Methode als `deferred` deklariert, so ist die gesamte Klasse abstrakt. Das gilt auch für die davon erbenden Klassen, solange die abstrakte Methode nicht mit einer konkreten Implementation überschrieben wird.
Im Gegensatz zum SmallEiffel-Compiler besteht der ISE-Eiffel-Compiler darauf, dass Klassen mit abstrakten Methoden auch mit dem Schlüsselwort `deferred` gekennzeichnet werden. Durch die vielen Kombinationsmöglichkeiten von Komponenten in der Sprache *PaL* passt dieses Konstrukt nicht in das Konzept dieser Sprache. Daher wurde der *PaL*-Compiler so erweitert, dass er alle Eiffelklassen, die abstrakte Features enthalten oder erben, als `deferred` kennzeichnet.
In diesem Zusammenhang konnte auch ein Problem gelöst werden, dass sich schon mit dem SmallEiffel-Compiler andeutete. Dieser meldete Warnungen, wenn eine generierte abstrakte Eiffelklasse eine `creation`-Klausel enthielt. Dies kann durch Kombination zweier Komponenten vorkommen und ist für den *PaL*-Programmierer teilweise unvermeidbar. Der ISE-Eiffel-Compiler meldet in solchen Fällen nicht nur Warnungen, sondern bricht mit einer Fehlermeldung ab.
Da der erweiterte *PaL*-Compiler aus der Klassenhierarchie ermittelt, ob eine Eiffelklasse abstrakt ist, kann in einem solchen Fall die Generierung der `creation`-Klausel in der Eiffelklasse unterdrückt werden. So meldet der SmallEiffel-Compiler keine Warnungen mehr und der ISE-Eiffel-Compiler bricht die Übersetzung nicht mehr aus diesem Grund ab.

import und der !-Konstruktor

ISE-Eiffel bietet eine umfangreiche Bibliothek zur Erzeugung grafischer Benutzungsschnittstellen für Windows. Mit der in [1] beschiebenen Sprache *PaL* ist es nicht möglich, solche Eiffelbibliotheken zu nutzen. Daher wurde die Sprache *PaL* zur Verwendung von Eiffelbibliotheken erweitert.
Die erste Erweiterung ist das `import`-Konstrukt für Patterns und Komponenten. Es ermöglicht das Erben von Eiffelklassen. Sollen Methoden der Eiffelklassen in der Sprache *PaL* überschrieben werden, so muss, wie bei der Vererbung in Eiffel üblich, das

`redefine`-Konstrukt verwendet werden. Im Gegensatz zur Vererbung zwischen reinen *PaL*-Komponenten kann der *PaL*-Compiler das `redefine`-Konstrukt beim Erben von einer Eiffelklasse nicht selbst generieren.

Die Syntaxregeln aus [1] müssen daher wie folgt ergänzt werden:

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

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

<IMPORT> ::=
  <CLASSNAME>
    [ redefine <CLASSFEATURENAME> { , <CLASSFEATURENAME> }* ]
  [ ;
  <CLASSNAME>
    [ redefine <CLASSFEATURENAME> { , <CLASSFEATURENAME> }* ] ]
```

Auf die oben beschriebene Weise lassen sich Eiffelklassen verwenden, indem man von ihnen erbt. In manchen Fällen benötigt man aber direkte Instanzen der Eiffelklassen und nicht der davon erbenden Patterns und Komponenten. Oft ist es auch einfach zu umständlich, für jede verwendetet Eiffelklasse noch eine *PaL*-Komponente einzuführen.
Für solche Fälle wurde ein neuer Konstruktor eingeführt, der es ermöglicht, Eiffelklassen direkt zu instanziieren. Der neue Konstruktor besteht aus nur einem Ausrufezeichen und soll sich so von dem !!-Konstruktor von Eiffel und *PaL* unterscheiden.

Die Syntax von *PaL* wird also wie folgt erweitert:

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

<IMPCREATION> ::=
  ! <WRITEABLE>.<LOCALCALL>
```

once

In der ersten Version von *PaL* war das Konstrukt `once` noch nicht vorgesehen. Es bewirkt, dass eine Methode nur einmal ausgeführt wird. Die importierten Klassen der Eiffelbibliothek machen häufig Gebrauch von diesem Konstrukt. Für die Implementation des Patterns Singleton (69) war es ebenfalls erforderlich. Daher wurde das `once`-Konstrukt auch in *PaL* eingeführt.

A.2 Die Komponentenstruktur des Design Patterns PDOCUMENT_WINDOW

Die Abbildung A.1 zeigt die vollständige Komponentenstruktur des Applikationspatterns `PDOCUMENT_WINDOW`.

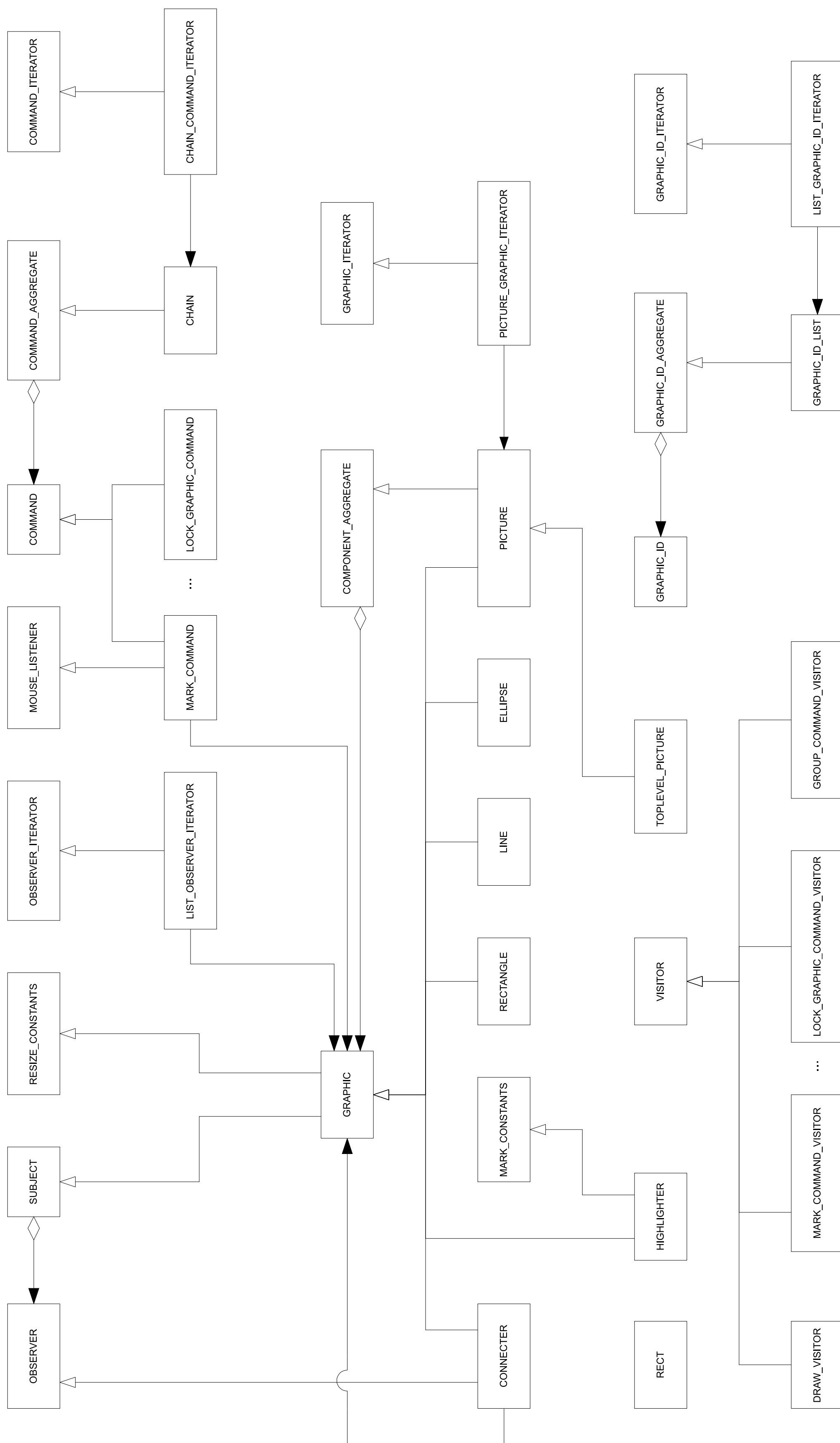


Abbildung A.1: Die Komponentenstruktur des Design Pattern `PDOCUMENT_WINDOW`

A.3 Der Quelltext von *DrawIt*

source.tex

Abbildungsverzeichnis

2.1 Die Struktur des Hilfspatterns Container 11
2.2 Verfeinerung zun Hilfstpattern Liste 12
2.3 Objektdiagramm einer Liste 12
2.4 Die Struktur des Hilfspatterns Parameter 15
2.5 Struktur des Patterns Fabrikmethode 17
2.6 Verfeinerung des Patterns Fabrikmethode zur Abstrakten Fabrik 20
2.7 Anwendung des Patterns Fabrikmethode 22
2.8 Verfeinerung zum Design Pattern Iterator 24
2.9 Verfeinerung zum Design Pattern Kompositum 28
2.10 Verfeinerung zum Design Pattern Interpreter 31
2.11 Verfeinerung zum Design Pattern Fliegengewicht 33
2.12 Verfeinerung zum Design Pattern Dekorierer 36
2.13 Verfeinerung zum Design Pattern Proxy 39
2.14 Erweiterung des Design Patterns Proxy 41
2.15 Verfeinerung zum Design Patterns Zuständigkeitskette 42
2.16 Die Struktur des Design Patterns Besucher 44
2.17 Erweiterung des Design Patterns Besucher 46
2.18 Verfeinerung zum Design Patterns Strategie 47
2.19 Verfeinerung zum Design Patterns Zustand 50
2.20 Verfeinerung zum Design Pattern Brücke 52
2.21 Verfeinerung zum Design Patterns Befehl 55
2.22 Die Struktur des Design Patterns Memento 58
2.23 Verfeinerung zum Design Pattern Beobachter 60
2.24 Verfeinerung zum Design Pattern Vermittler 63
2.25 Verfeinerung zum Design Pattern Erbauer 65
2.26 Die Struktur des Design Patterns Prototyp 67
2.27 Die Struktur des Design Patterns Singleton 69
2.28 Die Struktur des Design Patterns Fassade 71
2.29 Zwei Versionen des Design Patterns Adapter 73
2.30 Beispiel für das Design Pattern Schablonenmethode 75
2.31 Die Kombination der Patterns Liste und Iterator 77
2.32 Die Kombination der Patterns Kompositum und Iterator 78
2.33 Die Kombination der beiden vorhergehenden Patterns 79
2.34 Die Kombination eines Dekorierers mit dem vorhergehenden Pattern 80
2.35 Die Kombination der Patterns Beobachter und iterierbare Liste 81

3.1 Die Benutzungsschnittstelle von DrawIt 85
3.2 Das grafische Kompositum 89
3.3 Die Koordinatensysteme in DrawIt 90
3.4 Der Unterschied zwischen Ausdehnung und dem umschließenden Rechteck 91
3.5 Funktionalität für das grafische Kompositum 92
3.6 Darstellung von Markierungen 95
3.7 Das grafische Kompositum mit Dekorierer 96

3.8 Markierer im grafischen Kompositum 97
3.9 Verbinder in DrawIt 98
3.10 Verbinder im grafischen Kompositum 99
3.11 Eine Dokumentstruktur mit Verbinder 100
3.12 Kombination aus Befehlskette, Kommandos und grafischem Kompositum . . 103
3.13 Die Verwendung der WEL am Beispiel der Mausereignisbehandlung 105
3.14 Die Verfeinerung von PCOMMAND und PLIST_ITERATOR zu PCHAIN_OF_COMMAND 110
3.15 Die Verfeinerung von PCHAIN_OF_COMMAND, PVISITOR und PFACTORY_METHOD zu PCHAIN_OF_COMMAND_VISITOR 111
3.16 Die Verfeinerung von PCHAIN_OF_COMMAND_2_VISITOR (dreifach) und PCOMPOSITE_DECORATOR zu PCOMPOSITE_DECORATOR_VISITOR 113
3.17 Das applikationsspezifische Design Pattern PABSTRACT_GRAPHIC_COMPOSITE 114

A.1 Die Komponentenstruktur des Design Pattern PDOCUMENT_WINDOW . . 124

Tabellenverzeichnis

3.1 Attribute zur Positionierung in GRAPHIC 90
3.2 Kommandos in DrawIt 102
3.3 Grundlegende Design Patterns 107
3.4 Standardkombinationen 108

Literaturverzeichnis

[1] Stefan Bünnig: *Entwicklung einer Sprache zur Unterstützung von Design Patterns und Implementierung eines dazugehörigen Compilers*, Diplomarbeit, Universität Rostock, Fachbereich Informatik, 1999

[2] Stefan Bünnig, Peter Forbrig, Ralf Lämmel, Normen Seemann: *A Programming Language For Design Patterns*, ATPS'99, contained in ISBN 3-540-66450-5, Springer Verlag, 1999

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*, ISBN 0-201-63361-2, Addison Wesley, New York, 1995

[4] Normen Seemann: *A Design Pattern Oriented Programming Environment*, Master's thesis, University of Rostock, Department of Computer Science, 1999

Erklärung

Ich, Stefan Bünnig, erkläre, dass ich das Kapitel 2 der vorliegenden Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe[1].

Rostock, den 31.10.1999 Stefan Bünnig

Ich, Normen Seemann, erkläre, dass ich die Kapitel 3 und 4 der vorliegenden Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 31.10.1999 Normen Seemann

[1] Die Autoren sind für den Entwurf des Aufbaus dieser Arbeit sowie für die Implementation der Applikation *DrawIt* zu gleichen Teilen verantwortlich.