Diplomarbeit

Vergleich von Vererbungsmechanismen anhand von Design Patterns in C++, Eiffel und Smalltalk

ausgeführt am Institut für
Computersprachen
der Technischen Universität Wien

unter der Anleitung von
o.Univ.Prof. Dr. Manfred Brockhaus
und
Dr. Franz Puntigam
als verantwortlich mitwirkendem Universitätsassistenten

durch

Inge Bermann
Rosensteing. 42/7 1170 WIEN
Matr.Nr. 9325040

Wien, 14. Dezember 1999 ______________________

Kurzfassung

Diese Arbeit beschäftigt sich damit, die Frage nach der Eignung der drei objektorientierten Progammiersprachen C++, Eiffel und Smalltalk für bestimmte Aufgaben zu beantworten. Meist ist nämlich eine Sprache besser anwendbar als eine andere, weil sie hilfreiche Sprachelemente bereits besitzt oder aber vorausgesetzte Konstrukte nicht existieren und erst neu geschaffen werden müssen.
Der Vergleich wird mit Hilfe von Design Patterns durchgeführt, die bewährte Klassenentwürfe für bestimmte Anwendungsfälle darstellen. Die ausgesuchten Design Patterns beschäftigen sich speziell mit Vererbung. Im Rahmen dieser Arbeit werden sie implementiert und die Resultate jeder Sprache miteinander verglichen.
Das Ergebnis dieser Vergleiche zeigt, daß sowohl C++ als auch Eiffel und Smalltalk Vorteile wie auch Schwächen besitzen. Es existieren aber auch Unterschiede in den Möglichkeiten der Sprachen für eine effiziente Implementierung der Design Patterns.

Abstract

This thesis' purpose is to find out which of the three object-oriented languages C++, Eiffel and Smalltalk is best for implementing certain issues. Usually one programming language fits better than the other because it owns some special language features needed for implementing this problem. On the other hand it is possible that required features for a specific problem are not available, and so they have to be created before.
This comparison is done with Design Patterns, which stand for successfull and well-tried class-designs. The chosen Design Patterns deal with inheritance. They will get implemented and the results of each language will be compared.
On the whole it will be shown that both C++, Eiffel and Smalltalk have advantages as well as disadvantages. But, there are also differencies in the capabilities of the languages when trying to build efficient implementations.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Aufgabenstellung

Eine bewährte Methode in der Softwareentwicklung besteht darin, aus einem vorangegangenen Softwareprojekt die wiederverwendbaren Teile herauszufinden und brauchbare Elemente im aktuellen Projekt einzubinden. Zu diesem Zweck soll mit Hilfe einer speziell auf Wiederverwendung zugeschnittenen Implementierung eine Software produziert werden, deren Teile auch in zukünftigen Projekten eingesetzt werden können. Dies gilt im besonderen bei objektorientierten Programmiersprachen, wo ganze Klassen vergangener Projekte – sofern sie dafür geeignet sind beziehungsweise entworfen wurden – in späteren Applikationen erneut eingesetzt werden können.

Für verschiedene Aufgabenstellungen existieren natürlich mehrere Möglichkeiten, wie einzelne Klassen miteinander agieren beziehungsweise welche Beziehungen sie zueinander haben. Zur Beschreibung dieser Beziehungen gibt es Design Patterns, die jeweils eine bestimmte Aufgabenstellung mit einem gewissen Muster aus Klassen und Klassenbeziehungen verbinden. Design Patterns haben die Aufgabe, immer wiederkehrende Muster formal festzuhalten, sodaß sie – bei richtiger Einordnung der jeweiligen Aufgabe zu einem Design Pattern in der Entwurfsphase – herangezogen werden können, um die eigenen Klassen nach einem ähnlichen Schema zu entwerfen.

In dieser Arbeit wird der Schwerpunkt besonders auf die Thematik der Vererbungsmechanismen in objektorientierten Sprachen gelegt. Es werden spezielle Möglichkeiten behandelt, Probleme zu vermeiden, die bei üblicher Vererbung einer Subklasse von einer Superklasse auftreten können: Das Definieren einer Subklasse benötigt ausreichendes Verständnis der jeweiligen Superklasse, um deren Inhalte verwenden zu können. Beispielsweise könnte es vorkommen, daß durch das Überschreiben einer Methode in einer abgeleiteten Klasse (*overriding*) auch eine weitere Methode wegen der sonst eventuell fehlenden erweiterten Funktionalität überschrieben werden muß. Um diesen beispielhaften Fall gleich in der Entwurfsphase zu berücksichtigen und allfällige daraus resultierende Fehlermöglichkeiten auszuschließen, existieren Design Patterns, die Alternativlösungen zur einfachen Vererbung anbieten.

Weiters beschäftigt sich diese Arbeit mit der Frage, welche der drei behandelten Programmiersprachen – C++, Eiffel, Smalltalk – am ehesten für die Implementierung der ausgesuchten Muster geeignet ist. Eine besondere Eignung wird dann positiv interpretiert, wenn beispielsweise ein Muster bereits Bestandteil einer Sprache ist. Auf der anderen Seite ist es auch möglich, daß in einer Sprache das Verwirklichen eines Patterns äußerst schwierig und umständlich ist, da die dafür notwendigen Voraussetzungen (Sprachbestandteile) nicht vorhanden sind und eventuell erst programmiert werden müssen.

In diesem Dokument werden die Design Patterns als Werkzeug betrachtet, um die Möglichkeiten, Vor- und Nachteile der untersuchten Programmiersprachen herauszufinden und klar darzustellen.

C++, Eiffel und Smalltalk werden für die Implementierung und den Vergleich in dieser Arbeit herangezogen, da es sich dabei um typische bekannte objektorientierte Programmiersprachen handelt, die aber trotzdem einige unterschiedliche Sprachkonzepte besitzen. Bei der Implementierung werden nur GNU-Compiler unter Linux benutzt.

1.2 Aufbau der Arbeit

Nach der Einleitung wird im zweiten Kapitel der Begriff des Design Patterns anhand verschiedener in der Literatur existierender Begriffserläuterungen erklärt. Danach werden die einzelnen Design Patterns, die sich mit Alternativen zu herkömmlichen Entwurfsstrategien in Bezug auf die einfachen Vererbungshierarchien beschäftigen, gesondert vorgestellt und sinnvolle Verwendungsmöglichkeiten untersucht. Folgende Muster werden herangezogen:

- Bridge
- Chain of Responsibility
- Composite
- Decorator
- Observer
- Strategy

Das dritte Kapitel beschäftigt sich mit Programmiersprachen. Zunächst wird allgemein auf Eigenschaften, die Programmiersprachen erfüllen sollten, eingegangen und danach geht es speziell um die in der Arbeit verwendeten Sprachen C++, Eiffel und Smalltalk. Es werden unter anderem die Geschichte und die Spracheigenschaften untersucht.

Ab dem vierten Kapitel beginnt der Implementierungsvergleich der Design Patterns in den Programmiersprachen. Hier werden die für den Vergleich wichtigen Code-Fragmente dargestellt und einzeln kurz erklärt. Da einige Sprachkonstrukte in fast allen Patterns verwendbar sind und auch eingesetzt werden, wird die Beschreibung so aufgeteilt, daß es bei jedem Muster ein Spezialthema gibt, das hier besonders untersucht wird. Am Schluß jedes Kapitels gibt es eine Zusammenfassung, aus der hervorgehen soll, welche der untersuchten Programmiersprachen am ehesten für dieses Muster geeignet ist.

Zur Abrundung der Arbeit gibt es im elften Kapitel einen abschließenden Vergleich, in dem die zuvor gewonnenen Erkenntnisse über die Programmiersprachen zusammengefaßt werden.

Teil I

Übersicht

Kapitel 2

Design Patterns

Zu den Gründen, warum Design Patterns in neuen Projekten berücksichtig werden sollen, gehört unter anderem, daß nicht alle auftretenden Probleme der Realisierung bei jedem Projekt von Grund auf neu gelöst werden sollten. Stattdessen ist es sinnvoller, bereits bestehende Lösungsvarianten aus vergangenen Projekten heranzuziehen und auch in neuen Projekten einzusetzen (Reuse).

Solche immer wieder auftretenden Muster (*Patterns*) können allerdings nur dann verwendet werden, wenn man über ihre Existenz Bescheid weiß. In diesem Punkt haben Design-Spezialisten und erfahrene Programmierer deutlich Vorteile gegenüber weniger erfahrenen Designern. Da in anderer Form kaum problemspezifische Erfahrungen im Software Design weitergegeben werden, wurden sie in [Gamma et al., 1995] in Form der gängigsten allgemeinen Design Patterns zusammengefaßt.

Der Sinn dieser Zusammenstellung besteht darin, bei neuen Projekten in objektorientierten Programmiersprachen diese Muster wiederverwenden zu können, ohne sie aus eigener Erfahrung (aus bereits vorangegangenen Projekten) herausfiltern zu müssen. Kurz gefaßt kann durch Design Patterns schneller ein gutes Design erstellt werden.

2.1 Herkunft

Erstmals wurden Design Patterns von einem Architekten namens Christopher Alexander erwähnt. Seine Definition von Patterns wird folgendermaßen in [Alexander et al., 1977] zitiert: *„Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice.“* Obwohl Alexander hier eher über architektonische Muster in Bauten und Städten spricht, kann seine Definition ebensogut auf objektorientiertes Programmdesign angewandt werden.

Seit dieser ersten Definition von Alexander ist es bis 1987 rund um die Idee der Design

Patterns wieder still geworden, bis sie plötzlich auf der OOPSLA Konferenz 1987 auftauchten. Dort berichteten Ward Cunningham und Kent Beck über ihre eigene Zusammenstellung von fünf Patterns für das Entwerfen von Fenstern in Smalltalk. Sie schrieben auch über ihre positiven Erfahrungen, daß Spezialisten für das Erstellen von Applikationen bereits nach einem Tag sehr sinnvolle Benutzerschnittstellen erstellen konnten, ohne davor jemals mit diesen Mechanismen in Smalltalk etwas getan zu haben [Cunningham, Beck, 1987]. Seit damals wurden viele Artikel, Bücher und Präsentationen über dieses Thema verfaßt.

Schließlich erschien 1995 das Standardwerk [Gamma et al., 1995], dem großes Interesse und noch viele weitere Artikel und Bücher folgten und das auch als Grundlage dieser Arbeit dient.

2.2 Begriffsdefinitionen

Ein Design Pattern besteht laut [Gamma et al., 1995] aus folgenden Teilen:

- Name des Design Patterns
- Problem, das durch das Design Pattern gelöst werden soll
- Lösung des Problems
- Konsequenzen bei Anwendung des Patterns

Dirk Riehle und Heinz Zullighoven stellen in [Riehle, Zullighoven, 1996] folgendes fest: *A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts.*
In anderen Worten bedeutet das, daß es sich bei einem Muster um eine in vielen Projektentwürfen immer wiederkehrende Lösung eines bestimmten Problems handelt, das allerdings in abstrakter Form notiert ist. In diesem Artikel wird außerdem ergänzt, daß dieses Problem nicht unbedingt im Softwaredesign vorkommen muß; es handelt sich eher um eine allgemeine Definition.

In [Coplien, 1994] wird berichtet, daß die meisten Muster in der sogenannten *Alexandrian form* ausgedrückt werden, die folgendermaßen aussieht:

Name des Musters: Mit diesem Namen kann das Pattern referenziert werden.

Zu lösendes Problem: Patterns können angewandt werden, wenn das damit lösbare Problem bekannt ist.

Kontext: Ein Pattern löst ein Problem nur in einer gewissen Umgebung, in einem anderen Kontext muß es nicht unbedingt einen Sinn ergeben.

Forces: Nicht alle Probleme sind klar abgegrenzt. Ein Problem tritt in einer bestimmten Umgebung mit mehreren ebenfalls konkurrierenden Angelegenheiten (*forces*) auf.

Lösung: Mit einem Pattern wird auch die Struktur und das Verhalten einer Lösung beschrieben. Dies ist oft gleichwertig mit einer Beschreibung, wie eine Lösung erreicht werden kann.

Beispiele: In jedem guten Pattern werden auch Beispiele aufgezählt.

Resultierender Kontext: Nur wenige Patterns lassen ihre Umgebung unberührt. Jedes gute Pattern sollte daher auch aufzeigen, wie sich seine Umgebung durch seine Anwendung verändert.

Entwurfsgründe: In diesem Punkt wird erläutert, woher ein Pattern kommt, warum es funktioniert und warum Experten es benützen. Gute Designer können nämlich effektiver arbeiten, wenn sie die Hintergründe des Patterns kennen.

2.3 Ausgesuchte Design Patterns

Die folgenden in [Gamma et al., 1995] definierten Design Patterns beschäftigen sich speziell mit Vererbungsmechanismen. Ihr Aufbau und einige Verwendungsmöglichkeiten werden in den kommenden Abschnitten vorgestellt.

2.3.1 Bridge

Dieses Pattern ist auch unter dem Namen „Handle“ beziehungsweise „Body“ bekannt.
Die Motivation liegt darin, daß der übliche Weg, einer Abstraktion mehrere Implementationen hinzuzufügen, nicht besonders flexibel ist. Normalerweise besteht dieser Weg nämlich darin, eine abstrakte Klasse mit mehreren abgeleiteten Subklassen, die auf verschiedene Arten die Schnittstellen der abstrakten Klasse implementieren, zu schreiben. Der Nachteil an dieser Variante liegt darin, daß durch Vererbung die Subklasse an die Abstraktion permanent gebunden wird. Dadurch ist es nicht möglich, die Abstraktion beziehungsweise die Implementierung unabhängig voneinander zu verändern.

In [Gamma et al., 1995] wird als Anwendungsbeispiel dieses Patterns die Implementation einer portierbaren Abstraktion eines Programmfensters (*Window*) genannt. Sowohl in X-Window Systemen als auch im Presentation Manager (IBM) sollen Fenster eingesetzt werden können.

Bei einfacher Vererbung würde man diese beiden Subklassen von *Window* einfach von der abstrakten Klasse ableiten. Ein Nachteil liegt allerdings darin, daß dann Spezialklassen von *Window* (z.B. IconWindows oder ähnliches) sowohl von der X-Window-Implementation als auch von der Presentation Manager-Implementation abgeleitet werden müssen, da sie für jedes dieser Systeme einzeln zu implementieren sind. Weiters würde diese Vorgangsweise ein Programm vollkommen plattformabhängig machen, weil der Programmierer sich entscheiden müßte, ob er beispielsweise ein *IconWindow* aus dem Klassenbaum der X-Window Spezialisierung oder aus dem der Presentation Manager Spezialisierung verwenden möchte.

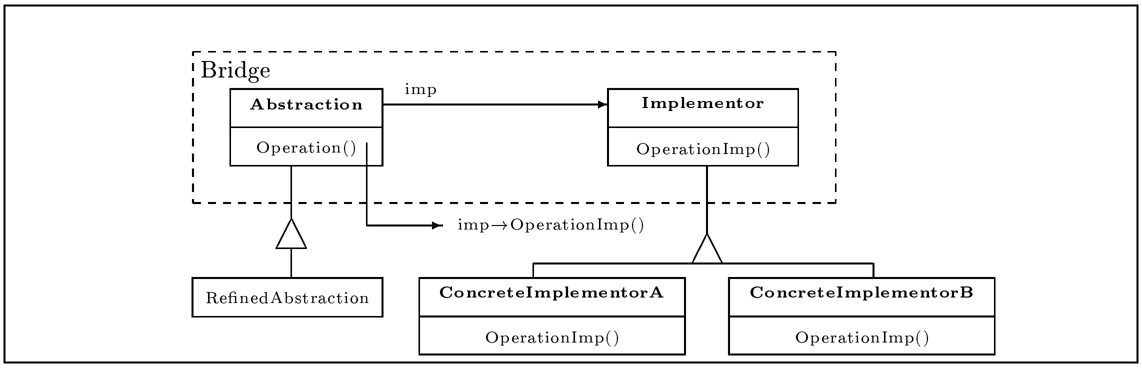

Abbildung 2.1: Bridge; aus [Gamma et al., 1995]

Als Lösung dieser Probleme ergibt sich das Design Pattern Bridge: Wie in Abbildung 2.1 dargestellt, existiert hier zusätzlich zur abstrakten Klasse *Window* die ebenfalls abstrakte Klasse *WindowImp*. Von der erstgenannten sind die Spezialisierungen bezüglich Art abgeleitet (z.B. *IconWindow*, *TransientWindow*) und von der zweitgenannten die bezüglich Plattform (z.B. *XWindowImp*, *PMWindowImp*). Die Verwendung sieht dann so aus, daß die plattformabhängigen Klassen primitive Methoden zur Verfügung stellen, die zum Zeichnen einfacher Elemente (Rahmen, Schrift, et cetera) dienen. Die speziellen Fensterklassen verwenden dann jeweils die Methoden der Plattform, die sie beispielsweise aus Umgebungsvariablen als die aktuellen identifizieren.

2.3.2 Chain of Responsibility

Die Aufgabe dieses Patterns liegt darin, den Sender einer Nachricht mit dessen Empfänger nicht statisch zu verbinden, indem ein Empfänger nur von einem bestimmten Sender etwas erhalten kann, sondern sie stattdessen voneinander zu trennen. Alle Objekte, die eine Nachricht empfangen und weiterverarbeiten können, sollen in eine Kette gereiht werden und jede Nachricht wird entlang dieser Reihung solange weitergegeben, bis der entsprechende Empfänger einer ganz bestimmten Nachricht gefunden ist und sie übernimmt.
In [Gamma et al., 1995] wird als Beispiel das Hilfesystem einer Applikation genannt. Hier hängt der Inhalt des darzustellenden Hilfetexts davon ab, aus welchem Teil der Applikation die Hilfe aufgerufen wurde. Beispielsweise stimmt der Hilfetext, der von einem entsprechenden Button in einem *Datei öffnen...* Dialog aufgerufen wurde, nicht mit dem in einem *Datei speichern...* Dialog überein.

Die Realisierung dieser Struktur wird so durchgeführt, daß das erste Objekt in dieser „Kette", das eine Nachricht erhält, diese entweder selbst behandelt oder entlang der Kette weiterreicht an das nächste Objekt. Der Absender der Nachricht weiß dadurch aber nicht, von welchem Objekt seine Anfrage letztendlich behandelt wird.
Der Aufbau der Klassen für dieses Pattern sieht wie in Abbildung 2.2 aus: Wenn eine Anfrage an *Handler* geschickt wird, sendet dieser die Nachricht an einen der abgeleite-

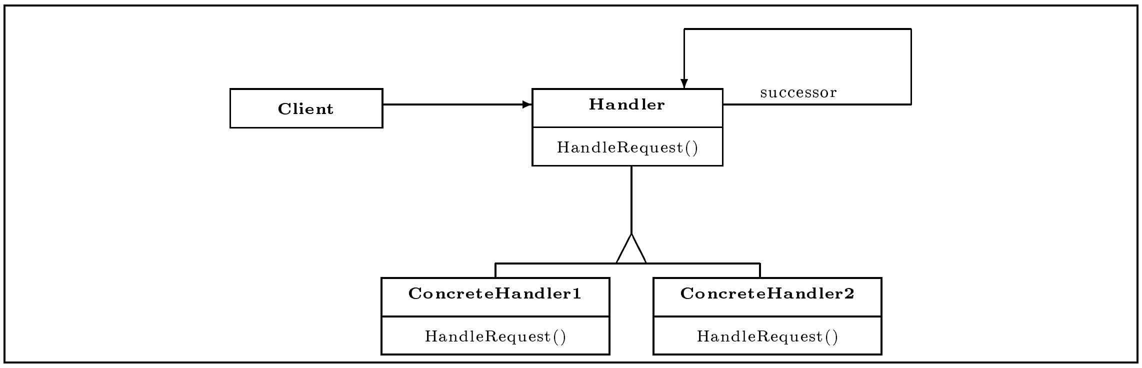

Abbildung 2.2: Chain of Responsibility; aus [Gamma et al., 1995]

ten *ConcreteHandlerX* weiter. Falls dieser nicht für die Behandlung zuständig ist, schickt er die Nachricht an seinen Vorgänger weiter, ansonsten kümmert er sich selbst um die Durchführung.

Wie in [Alpert et al., 1998] erläutert, kann man sich als gutes Beispiel einer *Chain of Responsibility* ein Militärkommando vorstellen: Angenommen, eine Privatperson möchte eine Auskunft haben und wendet sich daher an einen in der Militärhierarchie weit unten stehenden Beamten. Dieser wird seinen Vorgesetzten fragen, da er selbst keine ausreichende Antwort liefern kann. Der Vorgesetzte wendet sich ebenfalls an seinen Vorgesetzten und diese Reihung setzt sich solange fort, bis ein Offizier genügend Fachwissen besitzt und eine Antwort liefert. Diese wird dann in umgekehrter Reihenfolge bis zum ersten Beamten zurückgegeben, damit dieser sie schließlich an die Privatperson geben kann.
Beachtenswert bei diesem Beispiel sind einige Punkte, die die wesentlichen Charakteristika einer *Chain of Responsibility* ausmachen:

Keine statische Reihung: Alle Personen in dieser Kette werden öfters ausgetauscht oder umgereiht (beispielsweise durch Pensionierungen, Neueinstellungen oder Beförderungen).

Transparenz Die Privatperson, die den Auftrag gibt, erfährt nicht, welchen Weg die Anfrage in der Reihung genommen hat beziehungsweise ob sie sofort vom ersten Beamten oder vom obersten Militärkommandanten beantwortet wurde.

In [Alpert et al., 1998] wird die *Chain of Responsibility* als Spezialisierung eines generelleren Patterns, der objektorientierten Rekursion genannt. Diese behandelt Anfragen (Methodenaufrufe in einem Objekt) gleich wie das Design Pattern:

- Entweder wird die Anfrage vollkommen beantwortet und gleich dem aufrufenden Objekt retourniert
- oder die Anfrage wird der Superklasse weitergegeben, die dann ebenfalls den Methodenaufruf behandelt.

Das bedeutet, daß nur eine Klasse in der Klassenhierarchie wirklich die Anfrage inhaltlich behandelt; die anderen geben nur den Aufruf beziehungsweise danach die Antwort weiter.

2.3.3 Composite

Klassen einer Applikation, die miteinander eine Beziehung der Form „Whole-Part“[1] haben, können auf mehrere Arten entworfen werden.

- Es gibt einerseits die Möglichkeit, daß die Klassen, die andere beinhalten, unabhängig von diesen entworfen werden. Zwischen ihnen befindet sich lediglich die bereits angesprochene „Whole-Part“-Beziehung.
- Die andere Variante besteht darin, daß eine zusätzliche abstrakte Klasse in den Entwurf eingebaut ist, von der alle anderen Klassen direkt abgeleitet sind. Die abstrakte Klasse beinhaltet sowohl alle Methodendefinitionen der Containerklassen als auch die der einzelnen Teilklassen, sodaß hier eine Zusammenfassung erreicht wird.

Der Nachteil der ersten vorgestellten Version ist, daß die einzelnen Klassen unterschiedlich behandelt werden müssen. In [Gamma et al., 1995] wird als Beispiel eine graphische Applikation genannt, die mehrere Klassen für darstellbare Einzelkomponenten (z.B. Linie, Rechteck, Kreis, et cetera) und für zusammenfassende Klassen (z.B. Bild, das aus mehreren Einzelteilen besteht) vorsieht. Bei der ersten Entwurfsvariante wird ein Rechteck bei den Methodenaufrufen vollkommen anders behandelt als ein Bild, obwohl es beispielsweise in der Applikation für den Benutzer mit gleichem Aufwand möglich ist, Objekte sowohl von der einen als auch von der anderen Klasse am Bildschirm zu plazieren.
Die zweite Variante schafft hier Abhilfe: Eine übergeordnete Klasse (z.B. mit dem Namen

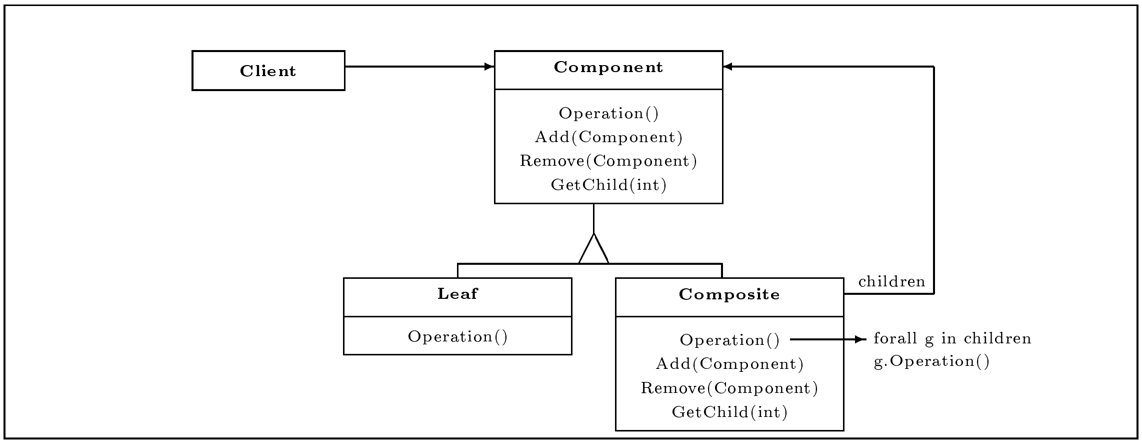

Abbildung 2.3: Composite; aus [Gamma et al., 1995]

[1] Eine Whole-Part Beziehung besteht zwischen zwei Klassen, wenn eine ein Teil der anderen ist (z.B. als Typ einer Klassenvariable).

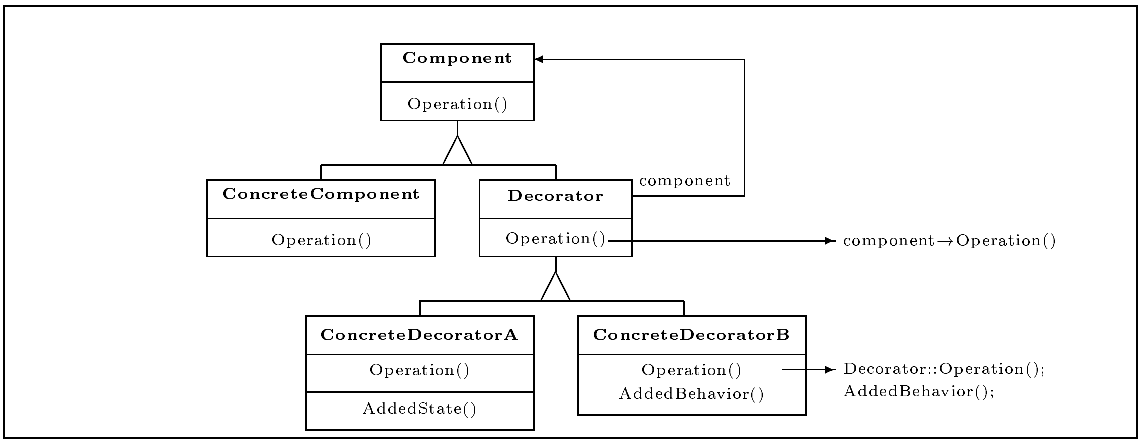

Abbildung 2.4: Decorator; aus [Gamma et al., 1995]

Graphic) stellt die Wurzel dieses Klassenbaums dar, der aus Einzel- wie auch aus zusammengesetzten Komponenten besteht. Jetzt ist es möglich, jedes Objekt dieses Baums gleichermaßen zu behandeln.
Eine allgemeine Darstellung dieser Struktur findet sich in Abbildung 2.3.

2.3.4 Decorator

Das Pattern *Decorator* ist auch unter dem Namen *Wrapper* bekannt und hat die Aufgabe, einfache Objekte mit zusätzlichen Eigenschaften dynamisch zu erweitern.
Beispielsweise ist es in einer Applikation mit graphischer Oberfläche möglich, daß einzelne Oberflächenkomponenten mit weiteren Eigenschaften ausgestattet werden können. In [Gamma et al., 1995] findet sich dazu das Beispiel einer Textkomponente, die mit einer Umrahmung und weiters mit einem Scrollfenster um die Textkomponente angereichert werden kann. In diesem Beispiel bietet es sich an, alle Klassen, die eine Erweiterung darstellen von einer abstrakten Klasse *Decorator* abzuleiten und diese wie auch die ursprüngliche einfache Oberflächenkomponente ebenfalls von einer gemeinsamen Klasse (*Component*) abzuleiten. Der Zusammenhang zwischen den Klassen in einer abstrakteren Form ist aus Abbildung 2.4 zu entnehmen.
Durch diese Vorgangsweise erreicht man, daß die Objekterweiterungen dynamisch zu einem Objekt dazufügbar sind. Die andere in Frage kommende herkömmliche Möglichkeit einer einfachen statischen Vererbung, daß das erweiterte Objekt vom allgemeinen abgeleitet wird, ist unflexibel und bei Verwendung vieler Dekoratoren unübersichtlich, da ja alle möglichen Erweiterungen als eigene Klasse vorhanden sein sollten. Wichtig an diesem Pattern ist, daß ein nicht dekoriertes gleich einem dekorierten Objekt angesprochen wird und daher der Verwender der Objekte keinen Unterschied sehen kann. Weiters kann ein Dekorator überall dort auftreten wo es auch einer einfachen Komponente möglich ist.

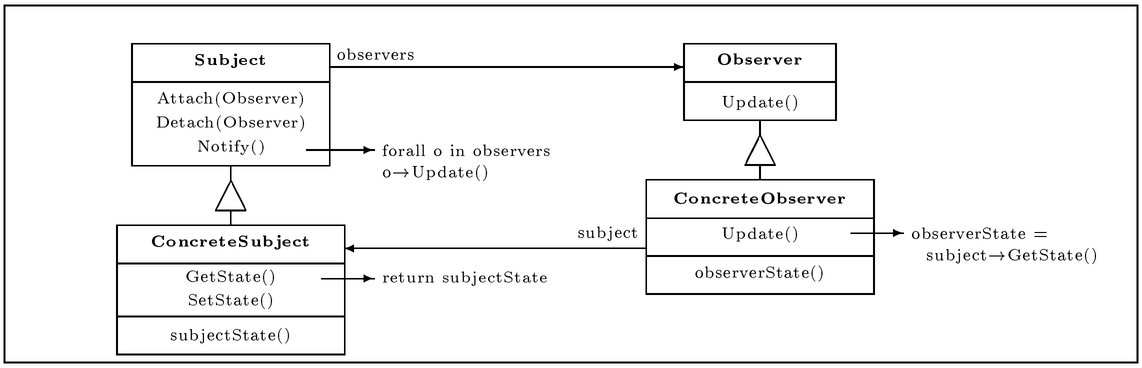

Abbildung 2.5: Observer; aus [Gamma et al., 1995]

2.3.5 Observer

Dieses Design Pattern ist auch unter dem Namen *Dependents* beziehungsweise *Publish-Subscribe* bekannt. Es bietet die Möglichkeit, mehrere Klassen, die inhaltlich voneinander abhängig sind, völlig unabhängig zu implementieren und auch wiederzuverwenden.
Genauer gesagt bedeutet das, daß beispielsweise in einer Applikation, die Daten darstellen soll, diese in einer Klasse enthalten sind. Die verschiedenen graphischen Darstellungen werden in jeweils separaten Observerklassen untergebracht, die unabhängig voneinander sind. Falls sich allerdings die Daten verändern, benachrichtigt das Datenobjekt alle Observerklassen, die sich dann den aktuellen Stand der Daten abholen. Dieser Zusammenhang ist auch in Abbildung 2.5 dargestellt. Im obigen Beispiel ist das Datenobjekt das *ConcreteSubject* und ein graphisches Darstellungsobjekt ein *ConcreteObserver*.
Bei diesem Pattern ist zu beachten, daß das *ConcreteSubject* alle seine *Observer* sofort benachrichtigt, wenn sich sein Zustand verändert hat, damit der Zustand der *Observer* nie inkonsistent mit seinem eigenen wird. Weiters ist es die Aufgabe jedes *ConcreteObservers*, sich den aktuellen Zustand des *ConcreteSubjects* sofort nach Benachrichtigung zu holen, um aktuell zu bleiben.

2.3.6 Strategy

Dieses Design Pattern ist auch unter dem Namen *Policy* bekannt. Es geht dabei darum, für einen Algorithmus mehrere Implementationen zu haben, die jedoch alle die gleiche Schnittstelle nach außen unterstützen. Alle Klassen, die eine konkrete Implementation des Algorithmus beinhalten, sind von einem übergeordneten Schnittstellenobjekt abgeleitet.
In [Gamma et al., 1995] wird als Beispiel eine Textverarbeitungsapplikation genannt, die einen Algorithmus für den Zeilenumbruch in einem Textfeld besitzen soll. Dieser Algorithmus ist allerdings ziemlich komplex. Deshalb erscheint es sinnvoller, ihn aus dem eigentlichen Programm in ein eigenes Objekt auszulagern. Da es weiters unterschiedliche Implementationsmöglichkeiten für den Zeilenumbruch gibt – beispielsweise sieht die Syntax in TEX anders aus als bei einem einfachen Text – wird eine *Strategy* eingesetzt.

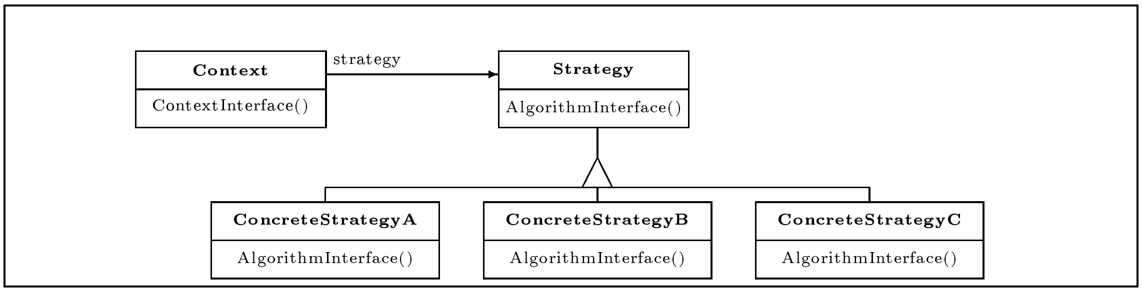

Abbildung 2.6: Strategy; aus [Gamma et al., 1995]

Mit verallgemeinerten Klassennamen sieht die Struktur wie in Abbildung 2.6 aus: *Strategy* ist die abstrakte Klasse des Algorithmus und in *ConcreteStrategyX* finden sich die jeweiligen Implementierungen. Das aufrufende Programm, das diesen Algorithmus verwenden möchte, wird mit der Klasse *Context* dargestellt.

Kapitel 3

Programmiersprachen

Anhand ihrer Eigenschaften können Programmiersprachen in einzelne Gruppen unterteilt werden. Als Programmierer sucht man sich die passende Sprache für ein Projekt mittels der Anforderungen, die sie unterstützen soll, aus. Folgende Kategorisierungen von Programmiersprachen wurden durchgeführt [Ghezzi, Jazayeri, 1996]:

Prozedurale Programmierung: Dabei handelt es sich um den konventionellen Programmierstil, wo Programme in einzelne Schritte unterteilt werden. Diese werden in Funktionen zur Modularisierung des Programms definiert.

Funktionale Programmierung: Diese Programmiertechnik stammt aus der Form der mathematischen Funktionen, wo Werte mit Hilfe von Ausdrücken und Funktionen errechnet werden.

Programmierung mit abstrakten Datentypen: Hier werden abstrakte Datentypen als Modularitätseinheiten im Programm angesehen.

Modulbasierte Programmierung: Bei dieser Technik besteht ein Programm aus Modulen, in denen Variable, Prozeduren, Funktionen, Typen et cetera enthalten sind. Beispiele solcher Sprachen sind Modula-2 und Ada.

Objektorientierte Programmierung: Die Grundkomponenten dieser Programmierart sind Klassen von Objekten. Auf diese Art von Programmiersprachen wird detailliert im nächsten Kapitel eingegangen.

Generische Programmierung: Hier existieren generische Module, die instanziert werden können, um benötigte Einheiten des Programmes (Datenstrukturen, Funktionen, Prozeduren) zu erzeugen. Dieses Programmparadigma existiert in keiner Programmiersprache als einziges; es wird gemeinsam mit ojektorientieren Ansätzen in Eiffel, mit funktionalen in ML sowie auch gemeinsam mit weiteren Paradigmen in Ada und C++ vermischt.

Deklarative Programmierung: Bei dieser Kategorie von Programmiersprachen gibt es nur eine deklarative Beschreibung des Problems, jedoch keine Aufteilung in eine algorithmische Implementation. Dadurch sind solche Programmiersprachen eher einer

Spezifikation als einer traditionellen Implementation ähnlich. Exemplare dieser Kategorie sind beispielsweise logikorientierte Sprachen wie Prolog oder regelbasierte wie OPS5 und CLIPS.

3.1 Objektorientierte Sprachen

Eine Programmiersprache wird nach [Stroustrup, 1995] als *objektorientiert* bezeichnet, wenn sie folgende Konzepte unterstützt:

Abstraktion: es werden Klassen beziehungsweise Objekte in der Sprache integriert.

Vererbung: neue Klassen können in Bezug auf bereits bestehende und als Erweiterung dieser gebaut werden.

Polymorphismus zur Laufzeit: erst zur Laufzeit werden Objekte an ihre Klassen gebunden.

Dadurch werden die Sprachen Ada95, Beta, C++, Eiffel, Simula, Smalltalk und weitere als *objektorientiert* definiert. Andere Sprachen wie beispielsweise C, Fortran oder Pascal sind dagegen aus dieser Gruppe ausgeschlossen, da sie unter anderem kein Klassenkonzept beinhalten.

Ein etwas anderer Ansatz zur Begriffserklärung einer objektorientierten Sprache findet sich bei [Meyer, 1988]. Hier werden folgende sieben Schritte angeführt, die eine objektorientierte Programmiersprache definieren:

Objektbasierte modulare Struktur: Systeme werden aufgrund ihrer Datenstrukturen modularisiert.

Datenabstraktion: Objekte sollten als Implementation von abstrakten Datentypen angesehen werden.

Automatische Speicherverwaltung: Objekte, die nicht mehr benötigt werden, sollten automatisch ohne Eingriff des Programmierers aus dem Speicher gelöscht werden (*garbage collection*).

Klassen: Jeder nicht-simple Typ ist ein Modul[1] und jedes höherwertige[2] Modul ist ein Typ.

Vererbung: Eine Klasse kann als eine Erweiterung oder Einschränkung einer anderen definiert werden.

Polymorphismus und dynamisches Binden: Programmentitäten sollen auf Objekte mehrerer Klassen zeigen können und es soll möglich sein, daß Operationen verschiedene Realisierungen in verschiedenen Klassen besitzen.

[1]Vordefinierte Typen wie beispielsweise Integer werden daher nicht als Module angesehen.

[2]Der Begriff „höherwertig“ läßt Programmstrukturen wie Prozeduren nicht in dieser Definition zu, diese sind nicht Typen.

Mehrfachvererbung und wiederholte Vererbung: Es sollte möglich sein, daß eine Klasse einerseits von mehreren Klassen abgeleitet ist und andererseits, daß auch sie mehrere Nachfahren haben kann.

Wenn diese Kriterien auf den zu behandelnden Programmiersprachen C++, Smalltalk und Eiffel angewendet werden, kann man feststellen, daß sie hauptsächlich für Eiffel definiert wurden. Dies kommt daher, daß Bertrand Meyer der Entwickler dieser Sprache ist, näheres dazu weiter unten.
C++ wird von Meyer laut seinen Kriterien nicht als objektorientiert angesehen, da es beispielsweise keine *garbage collection* durchführt[3], Smalltalk ebenfalls nicht, weil es nur einfache Vererbung zuläßt.

Eine Unterscheidung innerhalb der objektorientierten Programmiersprachen kann weiters vorgenommen werden, indem „hybride" von „reinen" objektorientierten Sprachen differenziert werden. C++ gehört beispielsweise den „hybriden" Programmiersprachen an, da es nicht nur objektorientierte sondern auch prozedurale Konzepte unterstützt. Dieser Ansatz hat aber sowohl Vor- als auch Nachteile. [Wirfs-Brock et al, 1990] führt an, daß es in einer hybriden Programmiersprache auf der einen Seite möglich ist, bereits existierende Programmierfähigkeiten einzusetzen und eventuell sogar bestehenden Code. Auf der anderen Seite wird das Programm allerdings schlechter lesbar sein, da Objekte mit gewöhnlichen Datentypen und gewöhnlichen Operationen gemischt werden können. Außerdem sind gewöhnliche Datentypen wie beispielsweise *int* oder *float* in C++ keine Klassen und es kann daher auch nicht eine Unterklasse von ihnen abgeleitet werden.

Kurz soll noch auf den Unterschied zwischen *objektbasiert*, *klassenbasiert* und *objektorientiert* eingegangen werden. In [Wegner, 1990] findet sich dazu folgende Definition:

- Objektbasierte Sprachen unterstützen die Verwendung von Objekten.
- In klassenbasierten Sprachen müssen alle Objekte zu einer Klasse gehören. Außerdem wird kein Klassenmanagement, sondern nur ein Objektmanagement durchgeführt.
- Erst in objektorientierten Sprachen müssen Klassen Vererbung anbieten, wodurch dann mehr Möglichkeiten beim Modellieren bestehen.

Jede objektorientierte Sprache ist durch die Definition also auch objektbasiert und klassenbasiert. Eine Unterscheidung, die besonders in [Hetzel-Herzog, 1994] hervorgehoben wird, ist die des Moduls als zentrales Sprachkonzept bei objektbasierten Sprachen. Es können zwar Datenabstraktionen formuliert werden, aber es fehlt beispielsweise bei ADA oder MODULA-2 ein Vererbungskonzept. Unter einem Modul wird eine Übersetzungseinheit verstanden, die zur Abkapselung von Datenstrukturen verwendet werden kann.

Bevor im nächsten Abschnitt genauer die Vererbung behandelt wird, soll noch zur Vollständigkeit auf den Unterschied zwischen *Subtyping* und *Subclassing* eingegangen werden, um

[3]Es ist allerdings möglich, die Operatioren *new* und *delete* in C++ neu zu implementieren und damit speicherverwaltende Maßnahmen zu setzen.

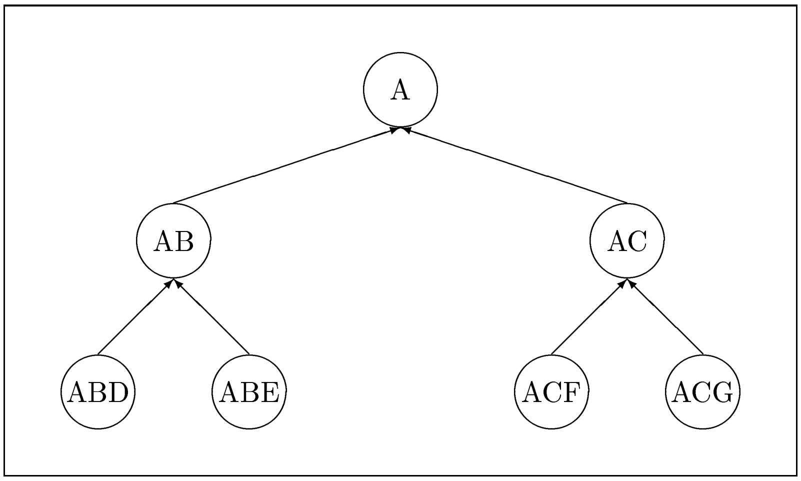

Abbildung 3.1: Einfachvererbung

auch eine Abgrenzung in diese Richtung zu bekommen. [LaLonde, Pugh, 1991] schreiben dazu in ihrem Artikel, daß Typen eine Spezifikation für ein Verhalten darstellen, während Klassen eine Implementation davon sind. Das bedeutet, daß in Subklassen sowohl die Implementierung als auch die Instanzvariablen der Superklasse geerbt werden. Bei *Subtyping* hingegen wird der Supertyp ersetzt, das heißt, daß hier nicht auf die Implementierung Rücksicht genommen wird. Nur die Schnittstelle nach außen muß gleich sein, damit der neue Typ anstatt des ursprünglichen eingesetzt werden kann.

3.2 Vererbungsmechanismen

Vererbung ist ein wichtiges Element der objektorientierten Programmierung, da sie eine einfache Wiederverwendung von Software ermöglicht. Durch Vererbung werden die Objekte nach Ähnlichkeiten sortiert, in Beziehung zueinander gebracht und in verschiedenartige Hierarchien eingesetzt [Singh, 1995].

- In vererbten Klassen ist es möglich, Eigenschaften der Elternklasse(n) zu spezialisieren beziehungsweise zu generalisieren.
- Durch Vererbung muß man beim Entwurf nur noch auf die Unterschiede zwischen Super- und Subklasse eingehen. Dadurch ist es nicht mehr nötig, Klassen von Grund auf neu zu entwerfen, wenn sie sich beispielsweise nur geringfügig von einer bereits bestehenden Klasse unterscheiden.
- Die Programmgröße wird reduziert, da die Funktionalität wiederverwendet wird.
- Vererbung bindet die Klassen auch konzeptuell zueinander: Bei einer Änderung in einer übergeordneten Klasse wird diese Veränderung an alle Subklassen weitergegeben, wodurch sich auch diese in ihrem Verhalten verändern.

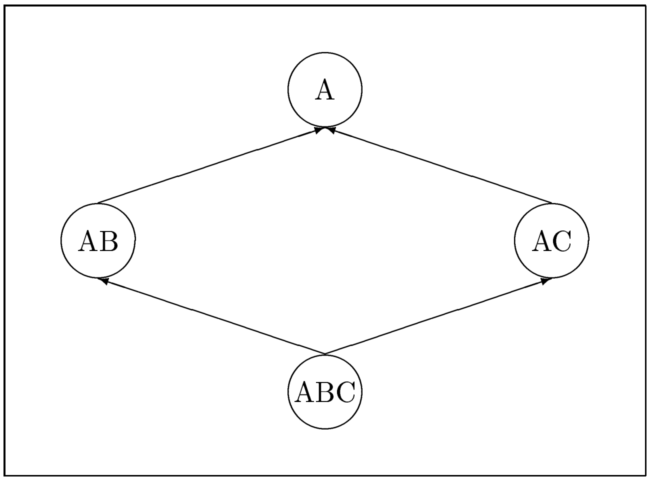

Abbildung 3.2: Mehrfachvererbung

Ein Unterschied innerhalb objektorientierter Programmiersprachen liegt auch in der Art der Vererbungsmöglichkeiten. Viele Sprachen lassen es zu, daß eine Subklasse von mehreren Klassen gleichzeitig abgeleitet wird (und laut [Meyer, 1988] ist die Mehrfachvererbung auch ein Kriterium einer objektorientierten Programmiersprache).
Auf der anderen Seite kann bei der einfachen Vererbung eine abgeleitete Klasse nur Eigenschaften *einer* Klasse erben. Dies führt dazu, daß alle Merkmale einer Klasse nur entweder von seiner Elternklasse ererbt oder in der eigenen Klasse definiert sein können. Einfachvererbung hat weiters die Eigenschaft, daß sie zu einer Klassenhierarchie beziehungsweise zu einem Klassenbaum führt, da zwar mehrere Klassen von einer Superklasse abgeleitet sein können, aber eben nur jeweils eine Superklasse haben können (siehe Abbildung 3.1[4]). Einfache Vererbung wird beispielsweise in Smalltalk, Objective-C und Java praktiziert.

Bei Mehrfachvererbung ist es Klassen auch möglich, sich von mehr als einer Superklasse abzuleiten. Dadurch resultiert diese Vererbung in einem gerichteten azyklischen Graphen. In der in Abbildung 3.2 schematisch dargestellten Klassenhierarchie besitzt die Klasse ABC zwei Superklassen, wodurch sie die Eigenschaften dieser beiden Klassen vererbt bekommt.

Mehrfachvererbung bringt aber auch Probleme mit sich:

Namenskonflikte: Wenn zwei Superklassen beispielsweise eine Methode mit dem gleichen Namen besitzen ist zu klären, von welcher der beiden die abgleitete Klasse diese Methode übernimmt beziehungsweise welche der beiden ausgeführt wird.

Wiederholte Vererbung: Bei Mehrfachvererbung kann eine Klasse von einer Superklas-

[4]In Abbildung 3.1 besitzt beispielsweise die Klasse AB die Klassen ABD und ABE als Subklassen. Diese sind nur von AB abgeleitet und keiner weiteren Klasse.

se öfters abgeleitet sein.

Methodenkombination: Falls Superklassen gleichlautende Methoden besitzen, die jedoch alle von der abgeleiteten Subklasse aufgerufen werden sollen, ist es notwendig, eine Möglichkeit für diesen Aufruf zu finden (das heißt, eine unterschiedliche Ansprechmöglichkeit der Elternklassen bereitzustellen).

Implementationsprobleme: Da Superklassen auf viele Arten in einer Subklasse kombiniert sein können, ist die Implementation der Mehrfachvererbung in einer Programmiersprache schwierig.

Mißbrauch der Mehrfachvererbung: Da es bei Vererbung jeder Subklasse möglich ist, auf die Variablen seiner Elternklasse zuzugreifen und es weiters gerade bei Mehrfachvererbung keinerlei Einschränkungen bei den Kombinationen gibt, kommt ein Mißbrauch davon öfters vor. Richtiger wäre es in diesen Fällen (wenn einige Klassen eigentlich keine Erweiterung der Superklassen sind), beispielsweise die vermeintliche Superklasse als Typ einer Instanzvariable einzusetzen.

Zu den für diese Arbeit ausgesuchten Design Patterns – Bridge, Chain of Responsibility, Composite, Decorator, Observer und Strategy – ist festzuhalten, daß sie sich mit Alternativen zur gewöhnlichen Vererbung (Composition, Delegation) beschäftigen.
Während eine herkömmliche Wiederverwendung von Klassen durch Erzeugen von Subklassen als *white-box reuse* bezeichnet wird, nennt man die Alternative durch Zusammenfügen oder Erzeugen neuer Objekte (Composition) *black-box reuse* [Gamma et al., 1995]. Eine solche Composition erfordert aber, daß es definierte Schnittstellen gibt, da keine internen Details der Objekte sichtbar sind. Der sichtbarste Unterschied zwischen Vererbung und Composition ist, daß die erste Variante statisch definiert ist, während die zweite erst zur Laufzeit dynamisch festgelegt wird.
Bei der Delegation geht es darum, daß zwei Objekte beim Behandeln einer Anfrage eingebunden sind. Dabei gibt die Klasse, die eine Nachricht von außen erhält, diese einfach an die andere weiter. Der Vorteil der Delegation liegt darin, daß diese zweite Klasse einfach austauschbar ist (und damit auch ihre spezielle Implementation). So ist es beispielsweise möglich, daß eine allgemeine Fensterklasse als *delegate* eine Rechtecksklasse besitzt, der sie Anfragen bezüglich ihrer Größe weitergibt. Bei Bedarf könnte aber ein Austausch auf eine Kreisklasse erfolgen, sodaß dieses Fenster rund wird. Der Nachteil der Delegation liegt allerdings darin, daß meist der produzierte Programmcode schwerer zu verstehen ist als bei statischen Definitionen.

Im Rest dieses Kapitels werden die drei Sprachen C++, Eiffel und Smalltalk näher beleuchtet und es wird speziell auf ihre Sprachkonzepte eingegangen.

3.3 C++

3.3.1 Geschichtliche Entwicklung

Die Programmiersprache C wurde in den AT&T Bell Laboratories mit der Aufgabe entwickelt, ein Betriebssystem fuer den PDP-11 Computer schreiben zu können (aus dieser Idee wurde dann später UNIX entwickelt). Bjarne Stroustrup, der ebenfalls bei AT&T Bell Laboratories arbeitete, wollte objektorientierte Konzepte in C einbauen. Da alle damals existierenden Implementierungen objektorientierter Programmiersprachen aber langsam und ineffizient waren, gab es auch das primäre Ziel, die existierende Effizienz von C in C++ beizubehalten.

C++ kann als Erweiterung von C angesehen werden: eine prozedurale Programmiersprache mit objektorientierten Konzepten. Daher ist es auch weiterhin möglich, in C++ ein einfaches prozedurales Programm mit C-Syntax zu schreiben. C++ wird deswegen auch als *hybride* objektorientierte Programmiersprache bezeichnet – im Gegensatz zu reinen objektorientierten Sprachen wie Eiffel oder Smalltalk.

Die objektorientierten Konzepte stammen von Simula und es handelt sich dabei um die Notation von benutzerdefinierten Klassen und Klassenhierarchien. Allerdings existieren trotzdem grundlegende Unterschiede zwischen C++ und Simula: Beispielsweise können in Simula Objekte von benutzerdefinierten Klassen nicht am Stack abgelegt und direkt adressiert werden. Stattdessen ist es notwendig, sie dynamisch anzulegen und mit Zeigern zu adressieren (diese Zeiger werden in Simula *Referenzen* genannt).

Die erste Version von C++ (mit dem Namen „C with classes“) gab es 1980. Sie unterstützte die traditionellen Systemprogrammiertechniken und abstrakte Klassen. Möglichkeiten für objektorientiertes Programmieren wurden erst 1983 eingeführt. Die erste kommerzielle Version von C++ wurde 1985 veröffentlicht, worauf sie in Industrie und Forschung breit eingesetzt wurde. In den Jahren 1987 bis 1989 wurden weiters noch Möglichkeiten für generische Programmierung in die Sprache eingebaut.

Standardisiert wurde C++ erstmals von ANSI (The American National Standards Organization) im Jahr 1990. Gemeinsam mit ISO (The International Standards Organization) führte das zu einem internationalen Standard im Jahr 1998.

In der momentan von ANSI standardisierten Version ist auch schon die STL (*Standard Template Library*) ein Teil von C++. Diese von Alexander Stepanov und Meng Lee geschriebene Klassenbibliothek beinhaltet vor allem generische Konzepte und Datenstrukturen. Die Idee der STL [Weidl, 1996] lag in den späten siebziger Jahren darin, daß Algorithmen nicht von einer spezifischen Implementierung ihrer Datenstrukturen abhängen. Das heißt, daß es zum Beispiel bei Sortieralgorithmen nicht wichtig ist, ob die Daten in einer verketteten Liste oder in einem Array angeordnet sind. Trotz dieser Unabhängigkeit wird in der STL großer Wert auf die Effizienz der Datenstrukturen gelegt.

3.3.2 Sprachkonzepte

Die folgenden Konzepte werden von C++ unterstützt [Stroustrup, 1995]:

Abstraktion: dadurch ist es möglich, Konzepte direkt in einem Programm darzustellen, wobei allerdings Details hinter definierten Schnittstellen verborgen bleiben.

Kapselung: es wird sichergestellt, daß eine Abstraktion nur gemäß ihrer Definition verwendet wird (dies erfordert die Schlüsselwörter *public*, *private* und *protected*).

Polymorphismus: die gleiche Schnittstelle nach außen wird von Objekten mit unterschiedlicher Funktionalität und Implementierung benutzt.

Vererbung: neue Abstraktionen werden aufgrund bereits bestehender definiert.

Generizität: Methoden von Klassen werden mit Typparametern versehen, sodaß sie als generelle Algorithmen für mehrere Typen verwendet werden können („Templates").

Koexistenz mit anderen Sprachen und Systemen: Teile eines Programms können auch in anderen Sprachen verfaßt sein, wodurch die Programmiersprache C++ in heutigen Systemen besser einsetzbar ist.

Kompaktheit des exekutierenden Programms und Geschwindigkeit: Da C++ ja von C abstammt, ist es ebenso für systemprogrammiertechnische Tätigkeiten geeignet.

Statische Typsicherheit: eine statische Typüberprüfung ist in Design und Implementierung vor allem größerer Projekte wichtig.

In C++ wird grundsätzlich laut [Stroustrup, 1997] auch die Idee der Invarianten berücksichtigt. Dabei handelt es sich um Programmteile, die dazu dienen, den Status eines Objektes auf Gültigkeit zu überprüfen. Beispielsweise kann untersucht werden, ob Zeiger auf Objekte, die gleich darauf angesprochen werden, ungleich NULL sind, um Laufzeitfehler zu vermeiden. Diese Art von Assertions kann allerdings nicht mit den in Eiffel vorhandenen diesbezüglichen Sprachkonzepten verglichen werden.

3.4 Eiffel

3.4.1 Geschichtliche Entwicklung

Im Jahr 1985 wollten die Franzosen Bertrand Meyer und Jean Marc Nerson eigentlich einen syntaxgesteuerten Editor programmieren. Bei einer Untersuchung, welche existierende Programmiersprache am besten für die Implementierung geeignet sei, fanden sie allerdings nichts passendes und beschlossen daher, eine neue objektorientierte Programmiersprache zu entwickeln.
Zu dieser Zeit existierte bereits C++ und Bertrand Meyer fand darin genau die Sprache, die Eiffel *nicht* werden sollte. Konzeptuelle Anleihen wurden eher von Ada genommen, obwohl sie keine objektorientierte Sprache ist. Beispielsweise hatten Syntaxkonventionen,

Kontrollstrukturen, et cetera von Ada großen Einfluß auf Eiffel.
Weiteren semantischen Einfluß auf Eiffel hatten noch Simula 67, Algol W, Alphard, CLU und zwei Spezifikationssprachen: Abrials Z und M (eine eigene Sprache von Meyer).

Bereits im April 1986 wurde der erste Eiffel-Compiler vorgestellt und schon 1987 wurde die Sprache in zahlreichen kommerziellen Anwendungsentwicklungen eingesetzt. Im Jahr 1991 wurde NICE (Nonprofit International Consortium for Eiffel) gegründet, das über die Rechte von Eiffel verfügt und über deren Normierung und Weiterentwicklung wacht sowie Standards für grundlegende Klassen definiert. In der weiteren Folge wurde 1995 mit ELKS (Eiffel Library Kernel Standard) ein Standard für Eiffel-Basisklassen von NICE definiert.

3.4.2 Sprachkonzepte

Bertrand Meyer beschreibt in [Meyer, 1988] unter anderem die Entwurfstechnik *Design by Contract*. Diese geht auf Dijkstra und Hoare in den späten 60er und 70er Jahren zurück und wurde in einer kommerziellen Programmiersprache erstmals in Eiffel implementiert. *Design by Contract* beinhaltet das Sprachkonzept der sogenannten *Assertions*. Auf der einen Seite handelt es sich dabei um die Möglichkeit, daß eine Methode mittels *Preconditions* beziehungsweise *Postconditions* darstellt, welche Vorbedingungen vor ihrem Aufruf erfüllt sein müssen beziehungsweise welche Bedingungen nach ihrer Ausführung zufriedengestellt sein werden [Meyer, 1995].
Beispielsweise könnte eine einfache Routine, die zwei Zahlen durcheinander dividiert, als *Precondition* verlangen, daß der Divisor ungleich Null ist. Dadurch können bei der Division keine undefinierten Werte entstehen. Eine andere Routine, die beispielsweise den Absolutbetrag einer Zahl retourniert, könnte als *Postcondition* beinhalten, daß der Ergebniswert sicher größer oder gleich Null sein wird.
Der Vorteil dieser Spracheigenschaft liegt darin, daß bei Verwendung solcher Routinen die *Pre-* und *Postconditions* bereits in der Schnittstelle erkennbar und dadurch im Gesamtprogamm diese Kenntnis verwendbar ist.

Die zweite Art von Assertions heißt *Invariants* und bezieht sich auf eine ganze Klasse (wie bereits auch oben für C++ beschrieben): Das bedeutet, daß sich diese Bedingungen im Gegensatz zu *Pre-* und *Postconditions* nicht nur auf einzelne Routinen auswirken, sondern auf alle Methoden und Attribute.

In Eiffel gibt es selbstverständlich eine Mehrfachvererbung. Namenskonflikte in einer von zwei Elternklassen abgeleiteten Klasse werden dadurch behoben, daß es möglich ist, Routinen oder Attribute in der Sublasse umzubenennen.

Eine wichtige Eigenschaft von Eiffel ist das dynamische Binden und die statische Typisierung. Das bedeutet, daß einer Variable, die als Typ eine allgemeine (eventuell abstrakte) Klasse hat, im Programm eine spezialisierte Instanz zugewiesen werden kann. Falls diese Zuweisung abhängig von einer Bedingung stattfindet, kann erst zur Laufzeit entschieden werden, welchem der beiden Typen diese Variable dann zugehört (dynamisches Binden).

Durch statische Typisierung wird allerdings sichergestellt, daß Methodenaufrufe dieser Instanz zur Laufzeit funktionieren, indem beim Kompilieren überprüft wird, ob in der Klassenhierarchie zumindest eine Klasse diese Methode implementiert hat.

Am GNU-Eiffel Compiler (mit dem Namen *SmallEiffel*) ist zu beachten, daß es sich eigentlich nur um eine Konvertierung des Codes in C handelt. Mit einem C-Compiler wird dann dieser Programmcode erst exekutierbar gemacht. Dadurch ist dieser Eiffelcompiler leicht portierbar und somit in mehreren Betriebssystemen einsetzbar.

3.5 Smalltalk

3.5.1 Geschichtliche Entwicklung

Die Entwicklung von Smalltalk beginnt in den frühen siebziger Jahren. Am Xerox Palo Alto Research Center (Parc) wurde ein neues Projekt namens „Dynabook“ vorgestellt: ein kostengünstiger PC in Größe eines Notebooks, in dem alle informationstechnischen Wünsche erfüllt sein sollten. Unter anderem war erwünscht, daß es einen Flachbildschirm mit Touch-Screen-Funktionalität besitzen sollte, Musikwiedergabe durch Ankoppelung an ein Sound-System ermöglicht wird und große Datenkapazitäten über Telefonleitungen importiert werden können.

Alan Kay, der Erfinder dieser Idee, glaubte, daß „Dynabooks“ eine revolutionäre Idee darstellen und viele Anwendungsbereiche haben werden. Unter anderem wollte er auch eine Programmiersprache in das System integrieren: damalige Sprachen waren allerdings eher für Programmierspezialisten entworfen worden und fast nur für numerische Aufgaben bestimmt. Eine neue Programmiersprache sollte daher entwickelt werden.

Zur selben Zeit wurde von S. Papert an der graphischen Programmiersprache *Logo* gearbeitet. Diese hatte die Aufgabe, Kindern das Erlernen von Programmiertätigkeiten zu erleichtern. Alan Kay hat mehrere Ideen in Smalltalk von Logo übernommen, da hier im Gegensatz zu prozeduralen Programmiersprachen einfach Erweiterungen im Funktionsumfang gemacht werden konnten, ohne komplexe Kombinationen von Prozeduren zu erhalten. Weiters hat er noch Vererbung integriert, um schließlich eine erste Version von Smalltalk zu erhalten.

Im Oktober 1972 bestand Smalltalk aus mehreren Zeilen Basic-Programm, das dann mit der Version Smalltalk-72 durch Assembler-Code abgelöst wurde. Smalltalk beinhaltete vor allem graphische Eingabeprogramme und hatte die Aufgabe, für Kinder als experimentelle Programmiersprache zu dienen.
Erst ab der Version Smalltalk-80 wurde die Sprache auch außerhalb von Xerox Parc verfügbar.

3.5.2 Sprachkonzepte

Die Implementation von Smalltalk wurde aus Portabilitätsgründen in zwei Teile geteilt:

- Virtual Machine
- Virtual Image

Während das *Virtual Image* die Klassen, Datenstrukturen, Compiler, Decompiler, Debugger und Benutzerschnittstelle von Smalltalk beinhaltet und systemunabhängig ist (da es direkt in Smalltalk geschrieben ist) besteht die *Virtual Machine* aus dem betriebssystemabhängigen Teil, einem Byte-Code Interpreter, einem Speichermanager und einigen primitiven Methoden. Dadurch ist es möglich, Smalltalk auf jede beliebige Plattform zu portieren.

Das Vokabular in Smalltalk kann durch fünf Worte – *object, message, class, instance* und *method* – vollständig beschrieben werden: Das zentrale Element der Sprache ist das Objekt und die Programmierung besteht grundsätzlich darin, Nachrichten an Objekte zu senden beziehungsweise Methoden von Instanzen aufzurufen.
Eigentlich ist Smalltalk eine interaktive Programmierumgebung, bestehend aus Objekten mit folgenden Funktionen: automatische Speicherverwaltung, ein Dateisystem, graphische Ein- und Ausgabemöglichkeiten (Eingaben über Tastatur beziehungsweise Maus), ein Debugger, eine Leistungsüberwachung, Kompilierungs- und Dekompilierungswerkzeuge, et cetera. Es ist für den Benutzer möglich, Informationen auf dem Bildschirm zu betrachten, interaktiv damit zu arbeiten, und einfach Nachrichten an die entsprechenden Objekte zu schicken.

Smalltalk benützt im Unterschied zu den bereits beschriebenen Sprachen keine statische Typisierung, sondern dynamische, wodurch beim Anlegen einer Variable nicht sofort deren Typ statisch festgelegt werden muß. Erst durch die spätere Verwendung ergibt sich dieser, der innerhalb des Programmes daher auch mehrfach gewechselt werden kann.
Da die Typen der Methodenparameter aus demselben Grund auch nicht statisch festgelegt werden, wird in Smalltalk automatisch ein Polymorphismus erzeugt: Abhängig von dem tatsächlichen verwendeten Parametertyp hat die Methode jeweils eine eigene Bedeutung. Zu beachten ist jedoch, daß die weitere Behandlung des Parameters in der Methode von dessen Klassentyp beziehungsweise seinen Superklassen auch zugelassen ist, da es sonst zu einem Laufzeitfehler kommt.

Bezüglich Vererbung unterstützt Smalltalk-80 keine *multiple inheritance*, sondern nur die Ableitung von jeweils einer Elternklasse. Obwohl Smalltalk oft als Paradebeispiel einer objektorientierten Sprache genannt wird, entspricht es nicht vollständig den Kriterien von Objektorientierung, die in [Meyer, 1988] genannt werden.

3.6 Zusammenfassung der Spracheigenschaften

In der folgenden Tabelle wird ein erster Vergleich über die Eigenschaften der oben vorgestellten Programmiersprachen gegeben. Zu berücksichtigen ist, daß dabei speziell auf die in Kapitel 4 genannten Übersetzer, die bei der Implementierung der Beispiele verwendet werden, eingegangen wird.

	C++	**Eiffel**	**Smalltalk**
Binden	statisch + dynamisch	dynamisch	dynamisch
Typisierung	statisch	statisch	dynamisch
Vererbung	mehrfach	mehrfach	einfach
Assertions	–	Preconditionen, Postconditionen, Invarianten	–
garbage collection	–	√	√

Wie in der obigen Tabelle ersichtlich ist, existieren grundlegende Unterschiede zwischen den drei Sprachen, die besonders beim Binden und der Typisierung hervortreten:

- In C++ werden Variablen und Methoden generell statisch gebunden. Es existiert allerdings das Schlüsselwort *virtual*, mit dem explizit dynamisches Binden verwendet werden kann.

- Außerdem gewährleistet C++ ein solches dynamisches Binden von Variablen dann allerdings nur über Zeiger. Das heißt, daß es nicht möglich ist, eine Variable, die als Typ eine abstrakte Klasse hat, einer Instanz der abgleiteten implementierten Klasse zuzuweisen. Stattdessen kann nur mit einem Zeiger oder einer Referenz diese Umlenkung vorgenommen werden.

- In Eiffel wird ein Überschreiben einer Methode anders angezeigt: Hier wird erst im Kopfteil der abgeleiteten Klasse angegeben, welche Methoden neu definiert werden beziehungsweise welche umbenannt werden.

- Im Gegensatz zu C++ gibt es in Eiffel überhaupt nur Referenzen auf Werte. Daher ist es dort nicht möglich, einen Parameter *byValue* mitzugeben.

- In Smalltalk werden alle Variablen und Methoden dynamisch gebunden, aber auch dynamisch typisiert. Dies hat nachteilige Wirkung auf die Sicherheit, da eher Laufzeitfehler entstehen können.

Die im ausführlichen Textteil über C++ beschriebenen Möglichkeiten von Invarianten in C++ reichen bei weitem nicht an die in Eiffel vorhandenen Assertions heran und können laut objektiver Definition auch nicht wirklich als solche klassifiziert werden. Daher sind in der Tabelle in der Spalte C++ auch keine Assertions angeführt.

Wie bereits in den vorigen Kapiteln angesprochen existiert für C++ schon eine Möglichkeit der *garbage collection*, diese ist jedoch nicht in der Sprache enthalten und kann daher nicht in diesem Vergleich berücksichtigt werden.

Im Implementierungsteil (ab dem 4. Kapitel) wird auf weitere spezielle Spracheigenschaften eingegangen. Die Kriterien, die dort verglichen werden und zur Charakterisierung eines Design Patterns verwendet werden, sind in der folgenden Tabelle als Vergleich zwischen den Programmiersprachen aufgesetzt. Eine nähere Beschreibung der Kriterien in der Tabelle[5] ist Kapitel 4 auf Seite 31 zu entnehmen.

Spracheigenschaft	**C++**	**Eiffel**	**Smalltalk**
Klassendefinition			
Einschränkung der Elementsichtbarkeit	++	+	−
abstrakte Klasse	++	++	++
Vererbung			
einfach (single inheritance)	++	++	++
mehrfach (multiple inheritance)	++	++	−−
Gemeinsame Superklasse			
Funktionalität	∼	+	+
Effizienz	∼	+	+
Instanzierte Objekte			
Aufwand der Instanzierung	−−	−	−
Möglichkeit der dynamischen Zuweisung	++	++	++
statischer Typ	++	++	−−

Die Bewertungen wurden aus Programmiererfahrung festgelegt und sind teilweise auch als subjektive Entscheidungen zu verstehen. Im folgenden werden einige Punkte erläutert:

- Eine Einschränkung der Elementsichtbarkeit ist nur in C++ ausreichend einsetzbar. Hier gibt es die Schlüsselwörter *private*, *protected* und *public*, um Variablen beziehungsweise Methoden vor einem Zugriff zu schützen, der von außerhalb der Klasse kommt. In Eiffel funktioniert der Mechanismus der Elementsichtbarkeit so, daß bei Variablen und Methoden angegeben werden kann, welche Klasse auf sie zugreifen darf. Falls dies für alle Klassen gelten soll, muß die allgemeine Superklasse *ANY* genannt werden. Für absolut privaten Zugriff ist *NONE* anzugeben. Der Nachteil an dieser Variante liegt allerdings darin, daß ein Zugriff von Subklassen der angegebenen Klasse ebenfalls nicht abwehrbar ist.

- In C++ existiert keine gemeinsame Superklasse, daher ist dieser Punkt für C++ nicht anwendbar.

- Das Instanzieren eines neuen Objekts ist in allen drei Sprachen einfach mit einem Konstruktor durchzuführen. C++ hat allerdings einerseits den Vorteil, daß der Kon-

[5]Die Ausprägung eines Kriteriums wird mit Noten angezeigt, diese gehen von −− über ∼ bis ++, wobei mit ∼ „nicht anwendbar“ ausgedrückt wird.

struktor immer einen genormten Namen hat (den Klassennamen) und weiters, daß eine Verschachtelung möglich ist.

3.7 Programmiersprachen und Design Patterns

Bezüglich der Implementation von Design Patterns in unterschiedlichen Programmiersprachen existiert das bereits angesprochene Standardwerk [Gamma et al., 1995], in dem Beispiele großteils in C++ und teilweise auch in Smalltalk geschrieben sind. Das zweite bei dieser Aufgabenstellung interessante Werk [Alpert et al., 1998] beschäftigt sich nur mit Smalltalk und möchte als Zielgruppe vor allem Smalltalk-Programmierer erreichen, die C++ nicht beherrschen.

Die Fragestellung dieser Diplomarbeit befaßt sich vor allem mit der Einsetzbarkeit der ausgesuchten Design Patterns in den drei Programmiersprachen. [Bosch, 1997] identifiziert in diesem Zusammenhang mehrere grundsätzliche Probleme:

Erkennbarkeit des Design Patterns: Meist kann ein Design Pattern in einer Implementierung nicht mehr erkannt werden, weil die jeweilige Programmiersprache ein entsprechendes Konzept nicht unterstützt. Daher muß der Programmierer dieses fehlende Konzept auf eine andere umständlichere Art implementieren, wodurch dann aber zusammengehörende Teile des Design Patterns über mehrere Objekte verstreut sein können.

Wiederverwendbarkeit: Design Patterns werden im Allgemeinen als Designstrukturen angesehen. Auf der Ebene der Implementation können sie daher meist nicht wiederverwendet werden, wodurch der Programmierer gezwungen ist, sie bei jedem Gebrauch neu zu implementieren. Dadurch wird zwar das Design wiederverwendet, nicht aber die Implementation.

Überschüssige Implementierung: Meist ist es bei Design Patterns wegen dem vorgegebenen Aussehen der Klassen und Objekte notwendig, Methoden zu implementieren, die nur triviale Aufgaben haben. Dies führt zu signifikanten Unkosten und weiters zu schlechterer Lesbarkeit des Programmcodes.

Ausgehend von diesen Problemen wird im nächsten Kapitel versucht, jeweils für jedes ausgesuchte Design Pattern die bestgeeignete Sprache für die Implementation herauszufinden.

Teil II
Implementierung

Kapitel 4

Allgemeines zur Implementierung

Wie bereits in der Einleitung angesprochen besteht die Aufgabe dieser Arbeit darin, einen Vergleich der drei Programmiersprachen C++, Eiffel und Smalltalk durchzuführen. Dieser soll durch Implementation von sechs Design Patterns, die sich mit Vererbung beschäftigen, gemacht werden.
Die Implementierung der Patterns wird unter Linux vorgenommen, um für alle Programmiersprachen die gleiche Umgebung zu haben. Die folgenden Compiler beziehungsweise Interpreter werden verwendet:

C++: GNU egcs-2.91.66: Dieser C++-Compiler beinhaltet bereits die STL (Standard Template Library).

Eiffel: GNU SmallEiffel Version -0.79: Leider entspricht dieser Compiler nicht ganz den in [Meyer, 1992] vorgestellten Erwartungen, da einige Klassen noch nicht implementiert aber im Standard vorgesehen sind.

Smalltalk: GNU Smalltalk Version 1.2.alpha5-43: Obwohl Smalltalk für interaktive Programme über eine graphische Oberfläche gedacht ist, wurde für diesen Vergleich entschieden, die Beispiele nur im Textmodus zu programmieren. In dieser Arbeit geht es ja nicht um Unterschiede in den Programmierumgebungen, sondern es werden die verschiedenen Implementierungsmöglichkeiten – also der produzierte Programmcode – untersucht.

Zu berücksichtigen ist, daß in den folgenden Beispielen immer die Implementierung in C++ als Referenz herangezogen wird, da diese Sprache der Autorin dieser Diplomarbeit am besten geläufig ist. Die Implementierungen in Eiffel und Smalltalk werden dann mit dieser bezüglich der in dem jeweiligen Design Pattern besonders auffälligen Spracheigenschaften verglichen, um schließlich eine Bewertung zu erhalten, welche der Sprachen am ehesten für das jeweilige Pattern geeignet ist. Dies kann auch dadurch erreicht werden, daß eine Sprache spezielle Voraussetzungen – durch bereits vorhandene Konstruktionselemente wie beispielsweise Container oder ähnliches – schafft.

Abgesehen von der allgemeinen Eignung eines Patterns für die Sprachen wird bei jedem

Beispiel eine spezielle Funktionalität oder ein Element der Sprache, das in dem jeweiligen Programm auch verwendet wird, genauer untersucht.

Weiters werden aber auch andere interessante Sprachelemente, die in fast jedem Pattern Anwendung finden, tabellarisch aufgelistet. In der Zusammenfassung jedes Kapitels wird dann in dieser Tabelle eine Bewertung über die Wichtigkeit dieses Konzepts für die Implementierung des aktuellen Design Patterns vorgenommen, die aus den Lösungen abgeleitet ist. Die dabei untersuchten Kriterien sind die folgenden:

- Klassendefinition

  Einschränkung der Elementsichtbarkeit: Dadurch wird bewirkt, daß von außen beispielsweise nicht einfach auf Instanzvariable einer Klasse zugegriffen werden kann.

  abstrakte Klasse: Eine abstrakte Klasse kann nicht instanziert werden; bei manchen Problemstellungen ist dies wichtig, weil sonst eventuell sinnlose Aufrufe von Methoden, die noch gar nicht ordentlich implementiert wurden, durchgeführt werden könnten.

- Vererbung

  einfach (single inheritance): Eine einfache Vererbung wird vermutlich in jedem Design Pattern benötigt.

  mehrfach (multiple inheritance)

- Gemeinsame Superklasse

  Funktionalität: In manchen Pattern ist es vorteilhaft, wenn eine im System vorhandene Superklasse auch eine gewisse reichhaltige Funktionalität mitbringt.

  Effizienz: Diese eventuell vorhandene Superklasse soll nicht zu viele unnötige Funktionen vererben, um abgeleitete Klassen nicht zu reichhaltig erscheinen zu lassen.

- Instanzierte Objekte

  Aufwand der Instanzierung (viele Instanzierungen): Als Aufwand wird angesehen, ob in einem Pattern viele Instanzvariablen notwendig sind. Bezüglich der Sprachen bedeutet das, daß eine Instanzierung kompliziert aussieht.

  Möglichkeit der dynamischen Zuweisung: Hier geht es um die Notwendigkeit des dynamischen Bindens.

  statischer Typ: Eine statische Typzuweisung ist bei manchen Design Patterns nicht nötig oder vielleicht sogar gar nicht erwünscht. Bei anderen Patterns würden vielleicht ohne statische Typisierung zuviele Fehler auftreten.

Bereits im letzten Kapitel auf Seite 27 wurde für jedes dieser Kriterien eingetragen, in welcher Programmiersprache es besonders gut ausgeprägt ist. Damit wird ein weiteres Indiz für die Verwendbarkeit eines Design Pattern in einer Sprache gezeigt.

Kapitel 5

Bridge

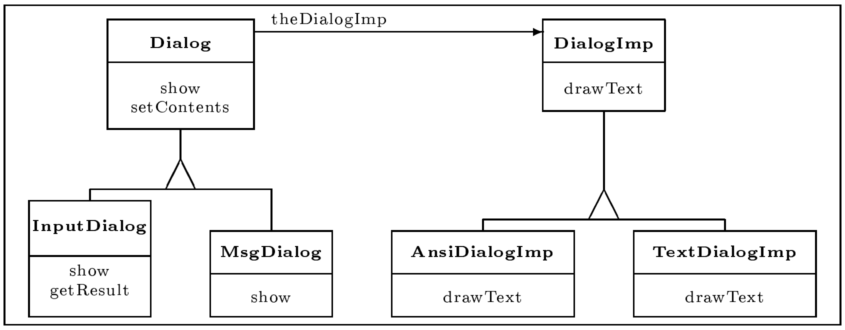

Abbildung 5.1: Implementierungsbeispiel einer Bridge

Für den Implementierungsvergleich des Pattern *Bridge* wird folgende Thematik als Beispiel herangezogen:
Eine Textausgabe auf den Bildschirm wird auf mehrere Arten implementiert und dann wird je nach der Umgebung, die zur Laufzeit aktuell ist, der jeweils entsprechend formatierte Text ausgegeben. Konkret wird auf die Umgebungsvariable *TERM* eingegangen, die für die Unterscheidung zwischen einem X-Terminal und einem ANSI-Terminal, in dem das Beispielprogramm ausgeführt wird, verantwortlich ist.

Die in Abbildung 5.1 schematisch gezeigten Klassenhierarchien geben ein einfaches Beispiel wieder, bei dem im linken Klassenbaum die Dialogarten und im rechten Teil die Implementierungshierarchie der Ausgabearten (Text mit ANSI-Steuerzeichen, Text ohne Steuerzeichen je nach verwendetem Terminal) abgebildet sind.

Instanziert wird durch den Benützer nur jeweils eine Klasse der Dialogarten (hier *InputDialog* oder *MsgDialog*), deren Methode *show* den Dialog am Bildschirm darstellt. In den Beispielprogrammen wird hier aus Einfachheitsgründen eine Textausgabe in ein Terminalfenster durchgeführt. Bei einem Inputdialog wird zusätzlich noch eine Benutzereingabe verarbeitet, die vom Objekt mittels der vorgesehenen Methode abgefragt werden kann. Je nach aktueller Terminalemulation wird entweder eine Instanz eines *AnsiDialogImp* oder eines *TextDialogImp* herangezogen.

5.1 C++

Die Main-Funktion befindet sich in *bridge.cpp* und beinhaltet die Instanzierung der benötigten Klassen sowie das Grundgerüst des Beispielprogramms:

```
/* File: bridge.cpp ***********************************************/

#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <fstream.h>

#include "inputdialog.h"
#include "msgdialog.h"

int main(int argc, char **argv) {
  cout << "Test of a  Bridge        " << endl
       << "(to be used in tcsh!)    " << endl << endl
       << "1) InputDialog:          " << endl;
  try {
    InputDialog *inDlg = new InputDialog("Please input a message:");
    inDlg->show();
    cout << "Your input was:" << inDlg->getResult()
         << endl << "2) MsgDialog " << endl;
    MsgDialog *msgDlg = new MsgDialog("This is a new message");
    msgDlg->show();
  } catch (const char *str) {
    cout << "Exception occurred: " << str << endl;
    return 1;
  }
  return 0;
}
```

Es wird in diesem Beispiel zuerst ein *Inputdialog* und danach ein *MsgDialog* aufgerufen. Interessant ist vor allem die Instanzierung eines Dialogs, da dabei festgelegt wird, welche Dialogvariante (*AnsiDialog* oder *TextDialog*) aktuell ist. Nachfolgend ist die Header-Datei der abstrakten Superklasse *Dialog* dargestellt:

```
/* File: dialog.h *************************************************/

#ifndef DIALOG
#define DIALOG

#include <stl.h>
#include "dialogimp.h"
```

```
class Dialog {
  public:
    Dialog(char *cont);
    ~Dialog();
    virtual void show() = 0;
    virtual void setContents(char *cont);
  protected:
    char *getContents();
    DialogImp *getDialogImp();
  private:
    char *theContents;
    DialogImp *theDialogImp;
};
#endif
```

Die privaten Variablen *theContents* und *theDialogImp* bezeichnen Zeiger auf den darzustellenden Text beziehungsweise die aktuelle Instanz eines *DialogImp*.
Im Konstruktor der Klasse *Dialog* wird die Variable *theDialogImp* mit einer aktuellen Implementierung eines Dialogs verbunden. Das sieht hier folgendermaßen aus:

```
/* File: dialog.cpp ************************************************/

#include "dialog.h"
#include "ansidlgimp.h"
#include "textdlgimp.h"

Dialog::Dialog(char *cont) {
  theContents = cont;
  if (strstr(getenv("TERM"), "ansi") != NULL)
     theDialogImp = new AnsiDialogImp();

  else theDialogImp = new TextDialogImp();
}
...
```

Es wird die aus C stammende Funktion *getenv(..)* verwendet, um den Inhalt der Umgebungsvariable TERM zu erhalten. Dementsprechend wird *theDialogImp* entweder ein Zeiger auf eine Instanz eines *AnsiDialogImp* oder eines *TextDialogImp*.

Ab dem Instanzierungszeitpunkt eines *MsgDialog* ist also festgelegt, wie sich dieser Dialog bezüglich seiner Umgebung verhalten soll. Die Implementierung von *MsgDialog::show* sieht folgendermaßen aus:

```
void MsgDialog::show() {
  if (getDialogImp() == NULL)
    throw "The variable theDialogImp was Null!";
  getDialogImp()->drawText(getContents());
}
```

In dieser kurzen Methode wird an die Instanz der aktuellen Dialogimplementationsklasse die Nachricht *drawText* (parametrisiert mit den darzustellenden Inhalten) gesendet. Eine Instanz eines *AnsiDialogImp* wird die Ausgabe dann eventuell mit ANSI-Steuerzeichen erweitern, während ein *TextDialogImp* nur den simplen Inhalt in das Terminalfenster schreibt.

An dieser Methode kristallisiert sich das Problem dieses Patterns heraus: Der Zeiger auf die Dialog-Implementation muß immer wirklich auf eine existierende Instanz zeigen und nicht etwa fälschlicherweise auf NULL. In C++ kann man etwaige diesbezügliche Laufzeitfehler mit Exceptions abfangen. Die Invariante, daß die Variable *theDialogImp* immer auf einen Wert ungleich NULL zeigt, wird in der Methode *MsgDialog::show* zu Beginn abgefragt.

5.2 Eiffel

Die Hauptklasse der Implementation dieses Patterns in Eiffel sieht folgendermaßen aus:

```
--* File: bridge.e *************************************************--

class BRIDGE
creation make
feature
  make is
    -- Constructor of class Bridge
  local
   indlg: INPUTDIALOG;
   msgdlg: MSGDIALOG;
   input: STRING
   do
    print("Test of a Bridge - Eiffel%N");
    !!indlg.make; !!msgdlg.make;
    indlg.setcontents("Please input a message:");
    indlg.show;
    input := indlg.getresult;
    print("Your input was:");
    print(input);
    print("%N");
    msgdlg.setcontents("%NThis is a new message");
    msgdlg.show;
  end;
end
```

An sich ist der Aufbau dieses Programmcodes dem aus C++ ziemlich ähnlich. Der Unterschied zwischen C++ und Eiffel liegt allerdings darin, daß in C++ der Hauptteil des Programms eine Funktion war, während er in Eiffel in eine eigene Klasse eingebaut wird. Beim Übersetzen wird diese Klasse explizit angegeben, damit der Compiler weiß, wo der Einstiegspunkt für den Ablauf ist. Diese unterschiedliche Vorgehensweise in Eiffel hat den Vorteil, daß beispielsweise zu Testzwecken auch andere Klassen als Einstiegspunkt gewählt werden können, während dies in C++ nicht möglich ist. Es wird sich in Smalltalk noch eine weitere Variante des Programmeinstiegs zeigen.

In der abstrakten Klasse *DIALOG* wird bestimmt, welchen implementierten Typ die Variable *thedialogimp* haben wird:

```
--* File: dialog.e **************************************************--

class DIALOG
creation {ANY} make
feature {ANY}
  make is
    -- Constructor of class DIALOG
  local
    ansidialogimp: ANSIDIALOGIMP
    textdialogimp: TEXTDIALOGIMP
    pnt: POINTER
    pnt2: POINTER
    str: STRING
  do
    !!ansidialogimp; !!textdialogimp;!!pnt;
      if (getterm = 'a') then
        thedialogimp := ansidialogimp;
      else
        thedialogimp := textdialogimp;
      end
  end;

  show is
    -- shows a dialog
  require
    thedialogimp /= Void
  deferred
  end;

  setcontents(cont: STRING) is
    -- set the contentsstring
  do
    contents := cont;
  end;

  getcontents : STRING is
    -- get the contentsstring
  do
    Result := contents;
  end;

  getterm: CHARACTER is
    -- get the environment variable TERM
   external "C_WithoutCurrent"
```

```
   end;

feature {DIALOG}
  contents: STRING;
  thedialogimp: DIALOGIMP;
end
```

Leider existiert in Eiffel keine Funktion, die den Inhalt von Umgebungsvariablen retourniert. Aber es ist möglich, einzelne Funktionen aus C einzubinden. Diese werden dann *external* genannt.
Diese Vorgangsweise wurde auch hier durchgeführt, indem in der Klasse *DIALOG* die Methode *getterm* wie oben angeführt definiert wurde. In einer zusätzlichen Datei *externalfunctions.c* wurde dann diese C-Funktion *getterm* implementiert und beim Kompilierungsvorgang miteingebunden.

```
/* File: externalfunctions.c                                      */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char getterm(void) {
  if (getenv("TERM") == NULL) return 'z';
  if (strlen(getenv("TERM")) != 0)
    return (getenv("TERM")[0]);
}
```

Eventuell erscheint es verwunderlich, warum nur der erste Buchstabe der Terminalbezeichnung (oder falls diese nicht vorhanden ist, ein ‚z') retourniert wird: Hier wurde deswegen der Typ *CHARACTER* gewählt, da es sich dabei um eine einfachen Typ in Eiffel handelt (weitere solche Typen sind *BOOLEAN*, *INTEGER*, *REAL*, *DOUBLE* und *POINTER*). Die Behandlung eines String wäre für diese einfache Funktionalität zu aufwendig.

Bei der Besprechung der Implementation in C++ wurde auch auf die Methode *MsgDialog::show* eingegangen: In Eiffel existieren eigene Syntaxelemente zur Beschreibung von Invarianten, Preconditions und Postconditions. Beispielsweise wird in der Methode *show* eine solche Precondition verwendet, in der festgelegt wird, daß es dort unbedingt erforderlich ist, daß die Variable *thedialogimp* auf einen Wert ungleich *Void* zeigt. Diese Vorgangsweise hat den Vorteil, daß solche Bedingungen auch für Methoden festgelegt werden können, die eigentlich noch als abstrakt gelten und erst in einer abgeleiteten Klasse implementiert werden. Falls die abgeleitete Klasse also von einem anderen Entwickler inhaltlich vervollständigt werden soll, ist diese Vorgangsweise bezüglich Schnittstellenbeschreibung korrekt und praktisch.

5.3 Smalltalk

In der Programmiersprache Smalltalk wird die Funktionalität und die Eigenschaften der Objekte in entsprechende Klassen geschrieben. Diese werden in der Smalltalk-Programmierumgebung in den existierenden Klassenbaum eingehängt und können dann mit gezielten Befehlen instanziert und verwendet werden. Diese gesammelten Befehle können als *Treiberprogramm* bezeichnet werden und sind für das aktuelle Pattern nachfolgend zu lesen:

```
"* File: bridge.st ***************************************************"

' Test of a Bridge                    ' printNl !
'1) InputDialog: ' printNl !
Smalltalk at:#indlg put:(InputDialog new: 'Please input a message')!
indlg show!
Smalltalk at:#input put:(String new:80)!
input := (indlg getResult) !
'Your input was' print !
input printNl !
'2) MsgDialog: ' printNl !
Smalltalk at:#msgdlg put:(MsgDialog new: 'This is a new message')!
msgdlg show !
```

Da es im GNU-Smalltalk derzeit leider nicht möglich ist, auf Umgebungsvariablen zuzugreifen, dies aber für den Vergleich nötig war, wird stattdessen die benötigte Information von einer Datei gelesen. Es gibt selbstverständlich Smalltalk-Umgebungen, in denen diese Funktionalität vorhanden wäre, aber der Sinn des Vergleichs besteht ja nicht darin, diese speziellen Eigenschaften zu untersuchen, sondern es sollen Differenzen gefunden werden, die unabhängig von der Smalltalk-Umgebung sind.. Die Klasse *Dialog* sieht wie folgt aus:

```
"* File: dialog.st ***************************************************"

Object subclass: #Dialog
        instanceVariableNames: 'theContents theDialogImp'
        classVariableNames: ''
        poolDictionaries: ''
        category: ''!

!Dialog class methodsFor: 'Instance Creation'!

new: contents
        | r |
        r := super new.
        r init: contents.
        ^r
!!

!Dialog methodsFor: 'abstract'!

init: contents
        | f line i |
        theContents := contents.
        f := (FileStream open: 'term.input' mode: 'r').
        line := (String new:80).
```

```
        i := 1.
        [ f atEnd ] whileFalse:
            [ line at: i put: (f next).
              i := i+1 ].
        (line sameAs: 'ansi') ifTrue: [ theDialogImp := (AnsiDialogImp new)]
                            ifFalse: [ theDialogImp := (TextDialogImp new)].
        f close
!

show
        self subclassResponsibility
!

setContents: contents
        theContents := contents
!

getContents
        ^theContents
!

getDialogImp
        ^theDialogImp
!!
"****************************************************************************"

Dialog subclass: #InputDialog
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: ''!

!InputDialog methodsFor: 'inputdialog'!

show
        self getDialogImp drawText: self getContents
!

getResult
        | line f c i |
        f := (FileStream open: '/dev/tty' mode: 'r').
        line := (String new:80).
        c := f next.
        i := 1.
        [(c isSeparator) not] whileTrue:
             [ line at: i put: c.
                   c := f next.
                   i := i+1 ].
        ^line
```

```
!!
"***************************************************************************"

Dialog subclass: #MsgDialog
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: ''!

!MsgDialog methodsFor: 'msgdialog'!

show
        self getDialogImp drawText: self getContents
!!
```

In der ersten dargestellten Klasse *Dialog* ist die Methode *init* bemerkenswert: Sie wird beim Instanzieren mit *new* aufgerufen und liest zunächst die Datei ein, in der die Umgebungsvariable steht. Je nachdem, ob die eingelesene Zeile aus dem Wort ‚ansi‘ besteht oder nicht, wird die Variable *theDialogImp* einem *AnsiDialogImp* oder einem *TextDialogImp* zugewiesen.
Während sowohl in C++ als auch in Eiffel versucht wurde, einen Programmabsturz in der Methode *show* zu vermeiden falls *theDialogImp* einen ungültigen Wert hat ist dies in Smalltalk nicht üblich. Ein Grund dafür ist vermutlich, daß die Klassen in Smalltalk für gewöhnlich in einer graphischen Programmierumgebung implementiert werden, in der solche Fehler eigentlich nicht möglich sind.

5.4 Zusammenfassung des Pattern *Bridge*

Anhand einer *Bridge* wurde vor allem untersucht, welche Möglichkeiten die drei Sprachen bieten, um Fehler zu vermeiden, die durch unerfüllte Vorbedingungen einer Methode oder nicht eingehaltenen Restriktionen einer ganzen Klasse passieren können.

- In C++ ist es üblich, in Methoden Variablenstati abzuprüfen und im ungünstigen Fall Exceptions zu werfen.

- Eiffel benützt spracheigene Konstrukte wie Preconditions und Postconditions. Der Vorteil dieser Vorgangsweise im Gegensatz zu der in C++ ist, daß hier die Bedingungen in der Schnittstellenbeschreibung (Methodendefinition) an einer vordefinierten Stelle und mit einem Schlüsselwort versehen vorhanden und daher einfacher zu finden sind. Dies ist besonders bei größeren Projekten wichtig, wenn mehrere Entwickler zusammen arbeiten.

- In Smalltalk steht Sicherheit nicht an erster Stelle, da eher das Augenmerk auf einfache Programmierung gelegt wird. Da allerdings von erfahrenen Smalltalk-Programmierern das textuelle Eintippen des Codes, wie es bei GNU-Smalltalk üblich ist, verachtet wird (stattdessen wird eine interaktive Programmierumgebung empfohlen, die die gewünschten Objekte in Text umsetzt), ist es vermutlich gar nicht so notwendig, diese Sicherheitsmaßnahmen zu setzen. Außerdem leidet natürlich auch die Performance unter den Überprüfungen bei jedem Methodenaufruf.

Eine weitere untersuchte Eigenschaft der Programmiersprachen in dieser Patternimplementation war der Aufbau des von außen aufrufbaren Programmteils. In der folgenden Tabelle befinden sich die augenscheinlichsten Unterschiede.

	C++	**Eiffel**	**Smalltalk**
Aufrufbares Hauptprogramm	main-Funktion	Hauptklasse	einzelne Befehle

Während in C++ die bereits aus C bekannte main-Funktion verwendet wird und einen eindeutigen Einstiegspunkt in das Programm darstellt, kann in Eiffel theoretisch jede beliebige Klasse dafür verwendet werden. Sie wird beim Kompilieren einfach als Hauptklasse ausgewiesen, wodurch ihr Konstruktor am Programmanfang aufgerufen wird.
In Smalltalk müssen explizit in der Befehlszeile der Programmierumgebung die gewünschten Objekte instanziert und verwendet werden. Obwohl Smalltalk als absolut objektorientierte Sprache gilt, ist es also doch möglich, einfache Befehle (ohne Berücksichtigung der zugrundeliegenden Objekte) auszuführen.

Im folgenden wird die in der Einleitung dieses Implementierungsaschnitts bereits angesprochene Tabelle, ausgefüllt für das Pattern *Bridge*, dargestellt.

Spracheigenschaft	**Wichtigkeit**
Klassendefinition	
Einschränkung der Elementsichtbarkeit	hoch
abstrakte Klasse	sehr hoch
Vererbung	
einfach (single inheritance)	sehr hoch
mehrfach (multiple inheritance)	niedrig
Gemeinsame Superklasse	
Funktionalität	sehr niedrig
Effizienz	sehr niedrig
Instanzierte Objekte	
Aufwand der Instanzierung (viele Instanzierungen)	durchschnittlich
Möglichkeit der dynamischen Zuweisung	sehr hoch
statischer Typ	durchschnittlich

Zum Pattern *Bridge* ist abschließend zu sagen, daß eigentlich alle drei Programmiersprachen dafür ganz gut geeignet sind und sich keine wirklich überzeugenden Argumente finden, warum eine Sprache bevorzugt werden soll.
In Bezug auf die Anwendung, die im Beispiel implementiert wurde, geht natürlich C++ als die bestgeeignete Sprache hervor, da nur dort sprachinterne Möglichkeiten vorhanden sind, die nötigen Umgebungsvariablen zu integrieren.

Kapitel 6

Chain of Responsibility

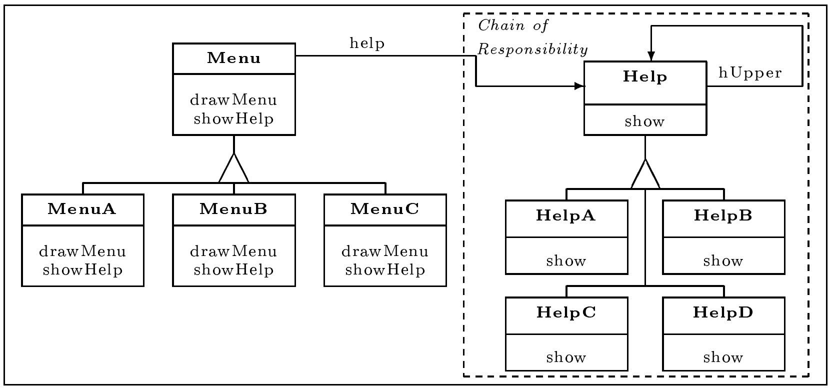

Abbildung 6.1: Implementierungsbeispiel einer Chain of Responsibility

Dieses Pattern ist dafür gedacht, dem Benutzer eine spezielle Klasse als Schnittstelle anzubieten, um deren Methoden zu verwenden. Diese Klasse ruft allerdings ihrerseits weitere interne Klassen auf (rekursiv entlang einer Kette), wodurch der Benutzer nicht weiß, welche interne Klasse ihm letztendlich das Resultat geliefert hat, das er von der Schnittstellenklasse bekommt.
Das implementierte Beispiel behandelt einen menügesteuerten Ablauf, wobei in einem einfachen Menü durch Eingabe eines Buchstabens navigiert werden kann. Die *Chain of Responsibility* befindet sich in den Hilfeklassen. Im Menü kann eine Hilfe über ein eingegebenes Wort angefordert werden, deren Ausgabe zunächst von der zum aktuellen Menü gehörenden Hilfeklasse erwartet wird. Kann diese Klasse nichts mit dem Wort anfangen, gibt sie den Auftrag zur nächsten allgemeineren Klasse weiter, bis er eventuell in der abstraktesten Hilfeklasse ankommt, die „Keine Hilfe verfügbar“ zurückgibt. Andernfalls wird ein jeweils passender Hilfetext retourniert.

Ein typischer Bildschirmausschnitt dieses Beispielprogramms (der bei allen drei Implementierungen gleich aussieht), ist in Abbildung 6.2 zu sehen. In diesem Programmablauf wird zunächst das Menü C aufgerufen, aus dem dann eine themenbezogene Hilfe angefordert wird. Als Thema wird ‚ac' eingegeben worauf folgende Kette durchlaufen wird:

```
> chain

*************************************************
*             Menu A                            *
*-----------------------------------------------*
* B.... Menu B                                  *
* C.... Menu C                                  *
* H.... Menuspecific Help                       *
* Q.... Quit                                    *
*************************************************

Input:c

*************************************************
*             Menu C                            *
*-----------------------------------------------*
* A.... Menu A                                  *
* B.... Menu B                                  *
* H.... Menuspecific Help                       *
* Q.... Quit                                    *
*************************************************

Input:h

Helptopic:ac
Help: 1 or 2 letters, including A-B-C

*************************************************
*             Menu C                            *
*-----------------------------------------------*
* A.... Menu A                                  *
* B.... Menu B                                  *
* H.... Menuspecific Help                       *
* Q.... Quit                                    *
*************************************************

Input:q

>
```

Abbildung 6.2: Typischer Bildschirmausschnitt dieses Beispiels

- Die Hilfeklasse des Menüs ‚C' bietet nur Hilfe für den einzelnen Buchstaben ‚c' an, daher wird das Hilfethema weitergereicht.
- Die nächste Klasse in der *Chain of Responsibility* retourniert Hilfe für Themen bis zu zwei Buchstaben, bestehend aus a, b, oder c.
- Für dieses Beispiel wurde also die passende Hilfeklasse gefunden und ein Hilfetext wird zurückgegeben.

6.1 C++

In C++ befindet sich die Hilfe-Funktionalität der entsprechenden Klassen in der Datei *help.cpp*. Die Definitionen dieser Klassen sehen folgendermaßen aus:

```
/* File: help.h **************************************************/

#include <iostream.h>

const unsigned short HELPLIMIT = 4;

class Help {
 public:
  Help();
  void show(char *topic);
 protected:
  Help *hUpper;          // Zeiger auf naechste Hilfeklasse in Kette
  char *HelpStr,         // Hilfetext dieser Klasse
       *TopicStr;        // vorgesehenes Hilfethema
  unsigned short HelpNr; // wie viele Buchstaben hat Hilfethema
};

class HelpA : public Help {
 public:
  HelpA();
};

class HelpB : public Help {
 public:
  HelpB();
};

class HelpC : public Help {
 public:
  HelpC();
};

class HelpD : public Help {
 public:
  HelpD();
};
```

Dabei gehören die Klassen *HelpA*, *HelpB* und *HelpC* zu ihren entsprechenden Pendants der Menüklassen, die hier allerdings aus Platzgründen nicht extra angeführt werden. Um sich deren Aufgaben besser vorstellen zu können – zu diesen Funktionen zählt das Darstellen der Menüs und das Aufrufen der Hilfefunktion – soll auf die Abbildung 6.2 verwiesen werden. Diese speziellen Hilfeklassen lassen nur einbuchstabige Themen zu, die entweder *a*, *b* oder *c* lauten. Die Klasse *HelpD* ist eine allgemeinere Hilfeklasse, die ein- bis zweibuchstabige Themen zuläßt; die ganz allgemeine Klasse, die auch nur allgemeine Hilfe anbietet, heißt *Help*.

```
/* File: help.cpp ****************************************************/

#include "help.h"
#include <string.h>
#include <ctype.h>

Help::Help() {
  hUpper = NULL;
  HelpStr = new char[81];
  strcpy(HelpStr, "General Help: 1-3 letters");
  TopicStr = new char[HELPLIMIT+1];
  strcpy(TopicStr, "ABC");
  HelpNr = 3;
}

void Help::show(char *topic) {
  for (unsigned int i=0; i < strlen(topic); i++)
     topic[i] = toupper(topic[i]);
  if (strlen(topic) >= HELPLIMIT)
    topic[HELPLIMIT] = '\0';
  if (strlen(topic) <= HelpNr && strcspn(TopicStr, topic) <= (HelpNr -
                                                          strlen(topic)))
    cout << HelpStr << endl;
  else if (hUpper != NULL)
    hUpper->show(topic);
  else cout << "No help available!" << endl;
}

HelpA::HelpA() {
  hUpper = new HelpD();
  HelpStr = new char[81];
  strcpy(HelpStr, "Help: 1 letter, A!");
  TopicStr = new char[81];
  strcpy(TopicStr, "A");
  HelpNr = 1;
}
...
```

In der oben auszugsweise dargestellten Implementierungsdatei findet sich zunächst der Konstruktor der allgemeinen Hilfeklasse beziehungsweise weiter unten der einer spezielleren Klasse *HelpA*. Dabei ist unter anderem darauf zu achten, daß in *Help* die Variable *hUpper* auf NULL gesetzt wird, während sie in *HelpA* auf eine Instanz von *HelpD* zeigt. Das liegt daran, daß es in *HelpA* einen Nachfolger gibt, *Help* jedoch befindet sich am Ende der Kette.
Interessant ist aber vor allem die Methode *Help::show(char *topic)*: In ihr wird entschieden, ob die aktuelle Klasse die am besten geeignete Antwort liefern kann, oder ob die Anfrage an die nächste weitergegeben werden soll.

Beim aktuellen Design Pattern soll zwischen den Programmiersprachen unterschieden werden, mit welchem Schlüsselwort ein ungültiger Wert dargestellt werden kann (die Variable hUpper soll entweder diesen Wert oder einen Zeiger auf eine gültige Instanz der nächsten Hilfeklasse beinhalten). In C++ wird ein ungültiger Wert durch NULL dargestellt.

6.2 Eiffel

Die abstrakte Klasse *HELP* hat in Eiffel ein strukturell ähnliches Aussehen wie in C++:

```
--* File: help.e ****************************************************--

class HELP
creation make
feature
  make is
    -- Creation of class HELP
  do
    upper := Void;
    !!helpstr.copy("General Help: 1-3 letters");
    !!topicstr.copy("ABC");
    helpnr := 3;
  end;

  show(topic: STRING) is
    -- shows the actual help for the topic
  local
    i: INTEGER;
    b: BOOLEAN;
  do
    topic.to_upper;
    if (topic.count >= 5) then
      topic.remove_last(topic.count-4);
    end;
    from
      i := 1;
      b := true;
    until
      i > topic.count
    loop
      b := (b and then (topicstr.has(topic.item(i))));
      i := i + 1;
    end
    if (topic.count <= helpnr and b) then
       print(helpstr);
       print("%N");
    elseif (upper /= Void) then
      upper.show(topic);
    else
      print("No help available!%N");
```

```
    end;
  end;

feature {HELP}
  upper: HELP;
  helpstr: STRING;
  topicstr: STRING;
  helpnr: INTEGER;
end
```

Wie auch oben in C++ wird hier in der Methode *show* entschieden, ob die aktuelle Klasse das gewünschte Hilfethema bearbeiten kann. Falls dies nicht der Fall ist, wird überprüft, ob sie die letzte Klasse in der Kette ist und im negativen Fall wird die gleiche Methode von dieser nächsten Klasse aufgerufen.
Auffallend bei der Implementation der Überprüfung des richtigen Hilfethemas ist vielleicht, daß es in C++ weitaus mehr Funktionen zur Veränderung beziehungsweise zum Vergleich zweier Zeichenketten gibt (die meisten dieser Funktionen gehen auf C zurück). Die hier für die Klärung der Frage, ob der mitgegebene String *topic* im klasseininternen *topicstr* enthalten ist, in Eiffel benutzte Methode *has* erfüllt daher auch nicht alle Einsatzmöglichkeiten des Beispielprogramms. Es wurde allerdings trotzdem so belassen, da die Aufgabe dieser Beispiele nicht die Funktionalität sondern die Vergleichbarkeit der Programmstrukturen ist.
Bezüglich ungültiger Werte ist für Eiffel zu sagen, daß es dafür den Wert *Void* gibt. Dieser ist Bestandteil der universellen Klasse *ANY*.

6.3 Smalltalk

Die Klasse *Help* und deren abgeleitete Klassen *HelpA*, *HelpB*, *HelpC* und *HelpD* sind im folgenden Ausschnitt der Datei *help.st* zu sehen:

```
"* File: help.st ***************************************************"

Object subclass: #Help
        instanceVariableNames: 'hUpper HelpStr TopicStr HelpNr'
        classVariableNames: ''
        poolDictionaries: ''
        category: '' !

!Help class methodsFor: 'instantiation'!

new
        | r |
        r := super new.
        r init.
        ^r
!!
```

```
!Help methodsFor: 'abstract'!

init
        hUpper := nil.
        HelpStr := 'General Help: 1-3 letters'.
        TopicStr := (String new:80).
        TopicStr at: 1 put: $a.
        TopicStr at: 2 put: $b.
        TopicStr at: 3 put: $c.
        HelpNr := 3
!

show: topic
        TopicStr printNl.
        (topic sameAs: TopicStr)
            ifTrue:  [ HelpStr printNl]
            ifFalse: [ hUpper isNil
                 ifFalse: [ hUpper show: topic]
                 ifTrue:  [ 'No help available!' printNl ]]
!!
"*******************************************************************************"

Help subclass: #HelpD
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: '' !

!HelpD methodsFor: 'HelpD'!

init
        hUpper := (Help new).
        HelpStr := 'Help: 1-2 letters, A-B-C!'.
        TopicStr := (String new:80).
        TopicStr at: 1 put: $a.
        TopicStr at: 2 put: $b.
        TopicStr at: 3 put: $c.
        HelpNr := 2
!!
"*******************************************************************************"

Help subclass: #HelpA
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: '' !

!HelpA methodsFor: 'HelpA'!
```

```
init
        hUpper := (HelpD new).
        HelpStr := 'Help: 1 letter, A!'.
        TopicStr := (String new:80).
        TopicStr at: 1 put: $a.
        HelpNr := 1
!!
....
```

Unterschiede zu den vorigen beiden Implementationen sind unter anderem in der Behandlung der Zeichenketten zu entdecken. Obwohl dies nur am Rande erwähnt sein soll, ist es doch ganz interessant, daß in Smalltalk ein deutlicher Unterschied zwischen einem echten String und einem Array of Character gemacht wird:
In der Datei *chain.st* befindet sich das Hauptprogramm. Hier werden zunächst alle Menüs instanziert und danach in einer Schleife die Benutzereingaben bei jedem Menü interpretiert. Dies sieht wie folgt aus:

```
"* File: chain.st ***************************************************"

Smalltalk at: #first put: (MenuA new)!
Smalltalk at: #second put: (MenuB new)!
Smalltalk at: #third put: (MenuC new)!
Smalltalk at: #current put: (Menu new)!
Smalltalk at: #c put: (Character new: 0)!

current := first!
c := (current drawMenu)!
[c = $q] whileFalse: [
    (c = $a) ifTrue: [current := first].
    (c = $b) ifTrue: [current := second].
    (c = $c) ifTrue: [current := third].
    (c = $h) ifTrue: [current showHelp].
    c := (current drawMenu)] !
```

Für jedes neue Menü wird einfach dessen Methode *drawMenu* aufgerufen; bei Eingabe von ‚q‘ endet die Schleife und die Eingabe von ‚h‘ ruft eine Methode *showHelp* auf.

```
...
showHelp
        | line f c i |
        'Helptopic: ' print.
        f := (FileStream open: '/dev/tty' mode: 'r').
        line := (String new:80).
        c := f next.
        i := 1.
        [(c isSeparator) not] whileTrue:
             [ line at: i put: c.
                  c := f next.
                  i := i+1 ].
        help show: line.
```

```
        f close
!
...
```

In dieser oben dargestellten Methode der Klasse *Menu* wird das Thema, das in der Hilfe behandelt werden soll, zeichenweise eingegeben. Danach wird die Methode *show* aus der zu diesem Menü gehörenden Hilfeklasse *help* aufgerufen, das dann eine weitere Behandlung des Themas wie bereits oben dargestellt, durchführt.
In Smalltalk wird ein deutlicher Unterschied zwischen der vorliegenden Vorgangsweise der Initialisierung der Variable *TopicStr* in der Methode *init* und einer Initialisierung der Art

```
TopicStr := 'abc'
```

gemacht. Bei letzterem wäre der Vergleich in der Methode *show*, ob die Zeichenkette *topic* gleich wie *TopicStr* ist niemals positiv. Aus diesem Grund wird die etwas umständlich wirkende Variante der Einzelzeichenzuweisung im *TopicStr* angewendet.

In Smalltalk gibt es den Wert *nil* als Referenz zu einem ungültigen Objekt, der auch in der Methode *init* der Klasse *Help* verwendet wird. Variablen, die noch nicht initialisiert worden sind, zeigen automatisch auf *nil*.

6.4 Zusammenfassung des Pattern *Chain of Responsibility*

Für dieses Design Pattern gibt es großteils Anwendungen, in denen Nachrichten innerhalb von Applikationen in einer eigens aufgebauten Hierarchie verschickt werden, um Anfragen von außen beantworten zu können. Da dies vermutlich meist über Zeichenketten geschieht, ist die Unterscheidung, wie gut eine Sprache Zeichenketten bearbeiten oder auch vergleichen kann, doch ein wichtiger Hinweis auf die Eignung dieser Sprache für eine *Chain of Responsibility*.
Und in diesem Punkt hat C++ am besten abgeschnitten. Der Grund dafür liegt darin, daß einerseits in C++ auch die in C sehr zahlreich vorhandenen Funktionen für Zeichenketten verwendet werden können und andererseits auch eigene Funktionen (für Streams) zur Verfügung stehen.

Aber es gibt noch einen weiteren Aspekt, der nicht unerwähnt bleiben darf und C++ deutlich von Eiffel und Smalltalk unterscheidet: In Eiffel und Smalltalk ist jedes Objekt von der Superklasse *ANY* beziehungsweise *Object* abgeleitet. In C++ existiert so eine vorgegebene Klasse nicht. Ein Vorteil der Superklassenvariante ist, daß in jeder neuen Klasse zumindest einige Methoden bereits existieren (wie beispielsweise die Variable *Void* in Eiffel) und auf diesen unabhängig von der aktuellen Implementation aufgebaut werden kann. Nachteilig ist aber sicherlich, daß jede Klasse relativ groß wird; eine wirklich kleine Klasse wie in C++ bauen zu können ist unmöglich. Dieser Unterschied wird auch erstmals in der abschließenden Tabelle über alle interessanten Spracheigenschaften erwähnt werden.

Die Ergebnisse der Sprachuntersuchung an den Beispielen der *Chain of Responsibility* sehen zusammengefaßt wie folgt aus:

	C++	**Eiffel**	**Smalltalk**
Zeichenkettenbearbeitung	viele Funktionen	wenig Funktionen	wenig Funktionen
Ungültiger Wert	NULL	Void	nil

Im folgenden wird wieder die tabellarische Darstellung der Wichtigkeit von Sprachkonzepten in einer *Chain of Responsibilitxy* gezeigt:

Spracheigenschaft	**Wichtigkeit**
Klassendefinition	
Einschränkung der Elementsichtbarkeit	hoch
abstrakte Klasse	sehr niedrig
Vererbung	
einfach (single inheritance)	sehr hoch
mehrfach (multiple inheritance)	sehr niedrig
Gemeinsame Superklasse	
Funktionalität	hoch
Effizienz	durchschnittlich
Instanzierte Objekte	
Aufwand der Instanzierung (viele Instanzierungen)	hoch
Möglichkeit der dynamischen Zuweisung	sehr hoch
statischer Typ	hoch

Zusammengefaßt sieht es so aus, daß C++ zwar die besten Eigenschaften für dieses Design Pattern (und vor allem eine Implementierung, die große Anforderungen an die Operatoren auf Zeichenketten) mitbringt. Es ist allerdings zu erwähnen, daß zumindest alle drei Sprachen die Möglichkeit haben, dieses Pattern kurz und einfach zu implementieren.

Kapitel 7

Composite

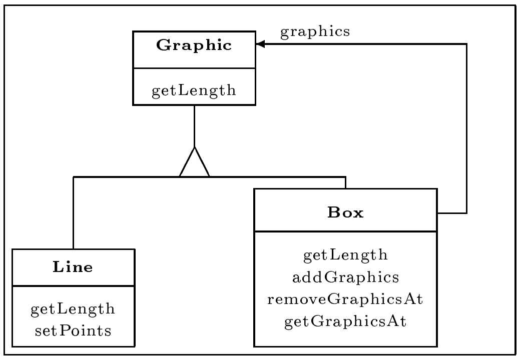

Abbildung 7.1: Implementierungsbeispiel eines Composites

Für dieses Design Pattern existieren viele Beispiele. Eines davon ist ein graphisches, in dem Vielecke, die aus mehreren Geraden bestehen, in Form eines *Composites* klassenhierarchisch abgebildet werden. Das bedeutet, daß ein Vieleck aus weiteren Vielecken und/oder aus einzelnen Geraden bestehen kann, wobei diese Varianten dynamisch durch den Anwender der Objekte veränderbar sind. Dadurch kann das gesamte Bild als ein einziges Vieleck betrachtet werden, um beispielsweise die benötigte Blattgröße zum Zeichnen errechnen zu könnnen. Für diesen Zweck ist es daher nicht notwendig, als Benutzer die enthaltenen Vielecke einzeln zu kennen.

7.1 C++

Im Implementierungsbeispiel dieses Patterns wird eine simple Variante gewählt, die nur aus den Klassen *Graphic*, *Line* und *Box* besteht (die Vererbung zwischen diesen Klassen ist der Abbildung 7.1 zu entnehmen).

In C++ befinden sich in dem in der STL bereits vorhandenen Container *vector* die einzelnen Bestandteile einer Box. Das können entweder Linien oder ebenfalls Boxen sein, daher wird als Inhalt des Vektors ein Zeiger auf ein Graphic-Objekt (*Graphic **) angegeben. Die Headerdatei dieses Patterns sieht folgendermaßen aus:

```
/* File: graphic.h *************************************************/

#include <stl.h>

class Graphic {
 public:
  Graphic() {}
  int getMinX() { return minX; }
  int getMinY() { return minY; }
  int getMaxX() { return maxX; }
  int getMaxY() { return maxY; }
  virtual double getLength() = 0;
 protected:
  void setMinX(int x) { minX = x; }
  void setMinY(int y) { minY = y; }
  void setMaxX(int x) { maxX = x; }
  void setMaxY(int y) { maxY = y; }
 private:
  int minX,
      minY,
      maxX,
      maxY;
};

class Line : public Graphic {
 public:
  Line();
  void setPoints(int x1, int y1, int x2, int y2);
  virtual double getLength();
};

class Box : public Graphic {
 public:
  Box() {}
  virtual ~Box();
  virtual double getLength();
  void addGraphics(Graphic*);
  void removeGraphicsAt(unsigned short);
  Graphic *getGraphicsAt(unsigned short);
 private:
  vector<Graphic*> graphics;
};
```

In der main-Funktion des Programms wird eine einfache Konstellation von Linien und Boxen aufgebaut: Zuerst werden vier Linien konstruiert, die sich dann zu einer Box verbinden lassen. Interessant bezüglich der Funktionalität ist die Methode *getLength*, die den

Umfang bei Boxen beziehungsweise die Länge bei Linien retourniert. Ihre Implementierung sieht folgendermaßen aus:

```
/* File: graphic.cpp **********************************************/

#include "graphic.h"
#include "math.h"
...
// return the length of the line
double Line::getLength() {
  return sqrt(pow(getMaxX()-getMinX(), 2) + pow(getMaxY()-getMinY(), 2));
}
...

// return the circumference of the box
double Box::getLength() {
  double l = 0;
  for (unsigned int i=0; i < graphics.size(); i++)
    l += graphics[i]->getLength();  // add every linelength
  return l;
}
```

Während bei der Klasse *Line* nur auf eigene Methoden und solche der Superklasse zugegriffen wird, berechnet sich bei der Klasse *Box* die Länge (d.h. der Umfang) durch Aufruf der Methode *getLength* aller *Graphic*-Objekte, die im Vektor *graphics* registriert sind, und Addieren der Ergebnisse.

Besonders zu berücksichtigen ist noch bei der Implementation in C++, daß die Klasse *Box* einen Destruktor benötigt, da es ja bei diesem Compiler keine garbage collection gibt. Der Destruktor leert den Vektor *graphics*.

7.2 Eiffel

Die Implementierung in Eiffel hat die gleiche Vererbungsstruktur und die gleichen Methoden wie die in C++. In Eiffel wird der Container der Klasse *Box*, in dem die in der Box enthaltenen Graphiken gespeichert werden, durch ein *ARRAY* dargestellt. Dieses besteht aus Objekten der Klasse *GRAPHIC* (im Gegensatz zu C++, wo es nur ein Zeiger auf ein *Graphic*-Objekt war).
Die Klasse *GRAPHIC* hat das folgende Aussehen:

```
--* File: graphic.e ************************************************--

class GRAPHIC
feature {ANY}
  getlength: DOUBLE is
   -- the graphic's length (min(XY) to max(XY))
  deferred
  end;

  minx: INTEGER;
  miny: INTEGER;
  maxx: INTEGER;
  maxy: INTEGER;
end
```

Hier wird also nur definiert, daß die Implementierungen in den abgeleiteten Klassen erfolgen sollen.
Die beiden Klassen *LINE* und *BOX* sind dann wie folgt implementiert:

```
--* File: line.e ***************************************************--

class LINE
inherit GRAPHIC
feature {ANY}
  getlength: DOUBLE is
   -- the graphic's length (min(XY) to max(XY))
  do
    Result := ((maxx - minx)^2 + (maxy - miny)^2).sqrt;
  end;

  setpoints(x1: INTEGER; y1: INTEGER; x2: INTEGER; y2: INTEGER) is
    -- set the points
  do
    minx := x1;
    miny := y1;
    maxx := x2;
    maxy := y2;
  end;
end

--* File: box.e ****************************************************--

class BOX
 inherit GRAPHIC;
creation make
feature {ANY}
  make is
   -- Constructor of class BOX
  do
```

```
   !!graphics.make(0,0);  -- construct the parentclass
  end

  getlength: DOUBLE is
   -- return the graphic's length
  local
    i: INTEGER;
    grp: GRAPHIC;
    result: DOUBLE;
  do
   result := 0.0;
   from
    i := graphics.lower;
   until
    i > graphics.upper
   loop
     grp := graphics.item(i);
     if (grp /= Void) then
       result := result + grp.getlength;
     end
     i:=i+1;
   end
   Result := result;
  end;

  addgraphics(grp: GRAPHIC) is
     -- add a new graphic to the box
   do
    graphics.add_last(grp);
    io.print(graphics.count);
   end

  removegraphicsat(i: INTEGER) is
     -- remove the specified graphic from the box
   do
    graphics.remove(i);
   end

  getgraphicsat(i: INTEGER) : GRAPHIC is
      -- return the specified graphic
   do
    Result := graphics.item(i);
   end

  graphics: ARRAY[GRAPHIC];
end
```

Besonders auffällig an der Methode *getlength* der Klasse *BOX* ist die Tatsache, daß im Vergleich zur Implementation in C++ um einiges mehr Codezeilen benötigt werden. Dies liegt einerseits sicher an der in Eiffel üblichen Formatierung, aber es sind auch andere Aspekte zu berücksichtigen:

- Bei der Abarbeitung des Arrays wird dieses in Eiffel nur von *graphics.lower* bis *graphics.upper* durchschritten. In C++ hingegen erfolgt das von *0* bis *graphics.size* (siehe dazu den entsprechenden Programmteil in Abschnitt 7.1). Dies bedeutet, daß dieser einfache Mechanismus, in Eiffel ein Array abzuarbeiten, für jedes Array anwendbar ist. In C++ ist es bei dieser Art notwendig zu wissen, mit welchem Index das Array beginnt. Eine weitere Möglichkeit bestünde darin, mit einem Iterator den Vektor in C++ durchzuarbeiten.

- Eine For-Schleife in C++ wirkt ebenfalls verkürzend auf den Programmcode, da in Eiffel der Schleifenzähler explizit weitergezählt werden muß. Dies würde einer While-Schleife in C++ entsprechen, die es allerdings in Eiffel nicht gibt.

7.3 Smalltalk

Auch in Smalltalk wurden die Klassen mit dem gleichen Vererbungsbaum strukturiert wie in Eiffel und Smalltalk, sodaß die Klassen *Line* und *Box* von *Graphic* abgeleitet sind. In jeder *Box* existiert eine Instanzvariable *graphics*, der bei der Instanzierung der Typ einer *OrderedCollection* zugewiesen wird. In Smalltalk gibt es viele unterschiedliche Collection-Klassen, die jeweils für verschiedene Zwecke gedacht sind [Goldberg, Robson, 1989]:. Die *OrderedCollection* hat gegenüber einem einfachen *Array* den Vorteil, daß die benötigte Größe nicht schon beim Programmieren angegeben werden muß, sondern sich der Container selbst beim Hinzufügen beziehungsweise Herausnehmen eines Elements größenmäßig dynamisch verändert.
Die Klasse *Graphic* beinhaltet nur die Variablendefinitionen sowie primitive Methoden für die Retournierung der Variableninhalte. Auch in der Klasse *Line* werden nur die Methoden in Smalltalk übersetzt wie sie bereits aus den Implementierungen in C++ und Eiffel bekannt sind.

```
"* File: line.st ***************************************************"

Graphic subclass: #Line
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: ''!
!Line methodsFor: 'line'!

setPoints: points
        minX := (points at: 1).
        minY := (points at: 2).
        maxX := (points at: 3).
        maxY := (points at: 4)
!

getLength
```

```
        ^(((self getMaxX - self getMinX) squared +
           (self getMaxY - self getMinY) squared) sqrt)
!!
```

Die *Box* besitzt die bereits angesprochene Variable *graphics*. Da in Smalltalk dynamische Typisierung durchgeführt wird, wird der Typ der Variable nicht in der Definition angeführt. Erst bei der Instanzierung wird er in der Methode *init* bekannt gegeben. Dadurch können dann in den Implementationen von *addGraphics* und *removeGraphics* auf die speziellen Methoden der Containerklasse zugegriffen werden.

```
"* File: box.st ****************************************************"

Graphic subclass: #Box
        instanceVariableNames: 'graphics'
        classVariableNames: ''
        poolDictionaries: ''
        category: ''!

!Box class methodsFor: 'Instance Creation'!

new: count
        | r |
        r := super new.
        r init: count.
        ^r
!!

!Box methodsFor: 'box'!

init: count
        graphics := OrderedCollection new
!

addGraphics: gr at: pl
        graphics add: gr
!

removeGraphics: gr
        graphics remove: gr
!

getGraphicsAt: pl
        ^(graphics at: pl)
!

getLength
        | result |
        result := 0.
        1 to:  graphics size do:
```

```
        [ :index | (graphics at: index) isNil
           ifFalse: [ result := result + (graphics at: index) getLength ]].
        ^result
!!
```

Eine interessante Facette an der dynamischen Typisierung in Smalltalk ist auch in der Methode *getLength* der Klasse *Box* zu entdecken: Obwohl im Gegensatz zu den Implementierungen in C++ oder Eiffel nirgends definiert wurde, welche Elemente in die *OrderedCollection* aufgenommen werden, wird in der Schleife von allen Objekten die Länge abgefragt (und addiert).

Daraus läßt sich eine Verwendungsmöglichkeit der Container in Smalltalk implizieren, die in C++ oder Eiffel in dieser Form nicht möglich ist: Es könnten auch verschiedene Elemente (Objekte unterschiedlicher Klassen, die nicht voneinander abgeleitet sind) in einen Container gesetzt werden.

7.4 Zusammenfassung des Pattern *Composite*

In diesem Kapitel wurde vor allem auf Container eingegangen, die im Pattern *Composite* benötigt werden. Dabei hat sich herausgestellt, daß in jeder der drei Programmiersprachen entsprechende Konstrukte vorhanden sind:

	C++	**Eiffel**	**Smalltalk**
Container	vector	ARRAY	OrderedCollection
Konstanter Elementtyp	$\surd$	$\surd$	–
Dynamische Größenänderung	$\surd$	$\surd$	$\surd$

Aufgrund der Klassen- und Vererbungsmechanismen ist zwischen den drei Implementierungen keinerlei Unterschied festzustellen, der sich für eine Sprache besonders begünstigend auswirkt. Sowohl in C++ als auch in Eiffel oder Smalltalk ist es möglich, den in Abbildung 7.1 schemenhaft dargestellten Objektzusammenhang zu implementieren.
Eine möglicherweise besondere Eignung der Sprache Eiffel für dieses Design Pattern ergibt sich aber bei Betrachtung des verwendeten Containers:

- Bezüglich Sicherheit wird nur hier bei der Arraydefinition der Typ der einzutragenden Elemente (die abstrakte Klasse *graphic*) angegeben (in C++ wird ein Pointer und in Smalltalk gar kein Typ definiert).
  Speziell zu Smalltalk ist zu bemerken, daß eigentlich die abstrakte Superklasse *Graphic* für die gewünschte Funktionalität unnötig ist: Im Gegensatz zu den anderen beiden Implementierungen wird wie eben erwähnt kein expliziter Typ der Containerelemente in Smalltalk definiert. Dadurch ist es aber möglich, sowohl Objekte der Klasse *Line* als auch der Klasse *Box* zu integrieren, ohne eine gemeinsame Elternklasse zu haben. Selbstverständlich gibt es auch negative Aspekte zu berücksichtigen, wie beispielsweise die Anforderung, daß alle im Container enthaltenen Objekte zumindest die definierte Methode *getLength* zur Verfügung stellen (dank dynamischer Typisierung wird das allerdings erst zur Laufzeit überprüft).

- C++ kann nicht als besonders geeignet angesehen werden, da es dort keine garbage collection gibt, die allerdings beim Löschen einer Instanz der Klasse *Box* notwendig wäre. Aus diesem Grund muß hier ein Destruktor eingesetzt werden.

Die Tabelle für alle betrachteten Sprachkonzepte dieses Pattern:

Spracheigenschaft	**Verwendung**
Klassendefinition	
Einschränkung der Elementsichtbarkeit	hoch
abstrakte Klasse	sehr hoch
Vererbung	
einfach (single inheritance)	sehr hoch
mehrfach (multiple inheritance)	sehr niedrig
Gemeinsame Superklasse	
Funktionalität	sehr niedrig
Effizienz	sehr niedrig
Instanzierte Objekte	
Aufwand der Instanzierung (viele Instanzierungen)	niedrig
Möglichkeit der dynamischen Zuweisung	sehr hoch
statischer Typ	durchschnittlich

Kapitel 8

Decorator

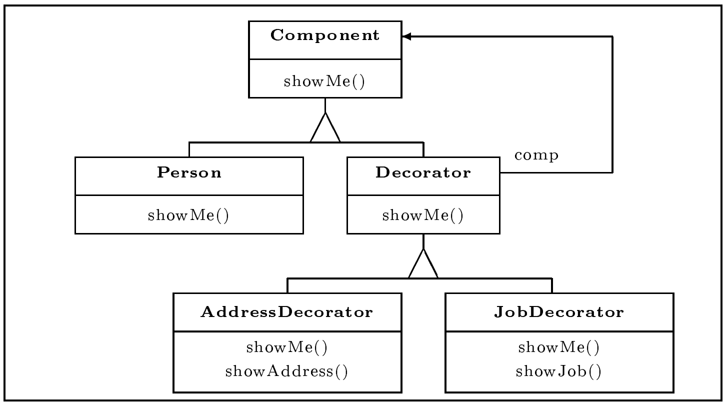

Abbildung 8.1: Implementierungsbeispiel eines Decorators

Dieses Pattern hat die Aufgabe, ein existierendes Objekt mit einer zusätzlichen Eigenschaft auszustatten. Beispielsweise könnte es sich dabei um ein Personenobjekt in einem Personenregister handeln, von dem man Name und Alter abfragen kann. Ein *Decorator* stattet dieses Objekt zusätzlich damit aus, daß der Beruf ausgegeben wird; ein anderer dient zur Darstellung der Wohnadresse. Dieses Beispiel ist beliebig erweiterbar, worin ja auch der Sinn eines *Decorators* besteht.

Um ein erweitertes Objekt zu erzeugen, wird ein *Decorator*, der eine einfache Person in der Variable *component* (siehe dazu Abbildung 8.1) eingetragen hat, instanziert.
Soll eine Person sowohl mit Beruf als auch mit Wohnadresse (sowie allen herkömmlichen Attributen) ausgegeben werden, muß ein *JobDecorator* mit einem *AddressDecorator* instanziert werden, der eine *Person* beinhaltet. Ein mögliche Darstellung dieser Zusammenhänge ist in Abbildung 8.2 zu sehen.

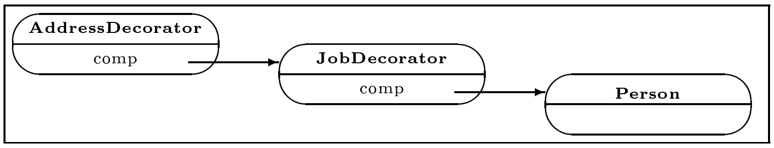

Abbildung 8.2: Objektzusammenhänge im Beispiel für einen Decorator

8.1 C++

Im folgenden wird die Headerdatei des Patterns dargestellt:

```
/* File: component.h ***********************************************/

class Component {
 public:
  virtual void showMe() = 0;
  virtual char *getName() = 0;
  virtual char *getJob() = 0;
  virtual char *getAddress() = 0;
  virtual unsigned short getAge() = 0;

 protected:
  char *name,
        *job,
        *address;
  unsigned short age;
};

class Person : public Component {
  public:
   Person(char *n, char *j, char *a, unsigned short ag);
   virtual void showMe();
   virtual char *getName() { return name; }
   virtual char *getJob() { return job; }
   virtual char *getAddress() { return address; }
   virtual unsigned short getAge() { return age; }
};

class Decorator : public Component {
 public:
  Decorator::Decorator(Component *c) { comp = c; }
  virtual void showMe();
  virtual char *getName() { return comp->getName(); }
  virtual char *getJob() { return comp->getJob(); }
  virtual char *getAddress() { return comp->getAddress(); }
  virtual unsigned short getAge() { return comp->getAge(); }
 protected:
  Component *comp;
```

```
};

class JobDecorator : public Decorator {
 public:
  JobDecorator(Component *c) : Decorator(c) {}
  virtual void showMe();
  void showJob();
};

class AddressDecorator : public Decorator {
 public:
  AddressDecorator(Component *c) : Decorator(c) {}
  virtual void showMe();
  void showAddress();
};
```

Bereits an der Headerdatei ist zu erkennen, daß die Methode *showMe* eine wichtige Rolle im Pattern spielt, da sie in jeder Klasse neu implementiert wird. In der Klasse *Person* werden damit die Attribute Name und Alter ausgegeben, in der Klasse *JobDecorator* der Beruf und in *AddressDecorator* die Wohnadresse. Genauer gesagt wird bei den beiden Dekoratoren zunächst die Methode *showMe* der abstrakten Superklasse *Decorator* aufgerufen und danach eine eigene Methode für das zusätzliche Attribut (*showJob*, *showAddress*). Im C++-Beispielprogramm sieht das folgendermaßen aus:

```
...

void Person::showMe() {
 cout << endl << "Hi, my name is " << name << ", I am " << age << " years old."
      << endl;
}

void Decorator::showMe() {
  if (comp != NULL)
    comp->showMe();
}

void JobDecorator::showMe() {
  Decorator::showMe();
  showJob();
}

void JobDecorator::showJob() {
  cout << "My job is " << comp->getJob() << endl;
}

void AddressDecorator::showMe() {
  Decorator::showMe();
  showAddress();
}
```

```
void AddressDecorator::showAddress() {
  cout << "My address is " << comp->getAddress() << endl;
}
```

Die Methode *Decorator::showMe()* ruft ihrerseits von der eingetragenen Instanz der Klasse *Component* wieder die gleichnamige Methode auf.
Das Aufrufen der gleichnamigen Methode aus der Superklasse ist in diesem Pattern ein wichtiger Schlüsselpunkt, an dem die Eignung beziehungsweise die Nicht-Eignung einer Programmiersprache gemessen werden soll.

Interessant ist aber auch die Kürze des Aufrufes in C++ aus der main-Funktion:

```
/* File: decorator.cpp *********************************************/

#include <iostream.h>
#include "component.h"

int main(int argc, char **argv) {
  cout << "A person with an address:" << endl;

  Component *cmp = new AddressDecorator(new
            JobDecorator(new Person("Inge", "Student", "Vienna", 24)));
  cmp->showMe();
}
```

Eine Verschachtelung mehrerer Objektkonstruktoren in einem Befehl trägt allerdings bei komplizierten Konstruktoren nicht unbedingt zur Lesbarkeit des erzeugten Programmcodes bei. Auf der anderen Seite kann man dadurch die Programmzeilenanzahl senken.

8.2 Eiffel

In Eiffel hat sich während der Implementation gezeigt, daß es mehrere Möglichkeiten gibt, die Methoden einer Superklasse von einer abgeleiteten Klasse aufzurufen.
Zunächst jedoch eine Vorstellung der Klassen:

```
--* File: component.e **********************************************--

class COMPONENT
feature {ANY}
  showme  is
   -- show a resulting string
  deferred
  end;

  getname: STRING is
   -- return the name
  deferred
```

```
  end;

  getjob: STRING is
   -- return the job
  deferred
  end;

  getaddress: STRING is
   -- return the address
  deferred
  end;

  getage: INTEGER is
   -- return the age
  deferred
  end;

  name: STRING;
  job: STRING;
  address: STRING;
  age: INTEGER;
end
```

Abgeleitet von dieser abstrakten Klasse sind, wie aus Abbildung 8.1 ersichtlich, einerseits die Klasse *PERSON* und andererseits *DECORATOR*, die weitere Subklassen besitzt.
Das Problem, das anhand dieses Design Patterns behandelt werden soll und bereits oben im Abschnitt über C++ angesprochen wurde, befaßt sich mit der Frage, wie Methoden einer Superklasse von der abgeleiteten Klasse aus erreicht werden können. Diese Aufgabe tritt in den Klassen *ADDRESSDECORATOR* und *JOBDECORATOR* in der Methode *showme* auf, in der zunächst die gleichnamige Methode aus der Superklasse *DECORATOR* sowie danach eine Erweiterungsmethode aufgerufen wird. Hier gibt es in Eiffel zwei Arten der Durchführung:

Precursor: Von Bertrand Meyer wird die Verwendung eines Zeigers auf Methoden der eigenen Superklasse zwar verschmäht – in [Meyer, 1992] wird erörtert, warum Benutzer einer Klasse nichts von deren Vorfahren erfahren brauchen – aber in GNU SmallEiffel ist ein derartiges Sprachelement vorhanden.

rename-redefine: Der Eiffel-Standard empfiehlt stattdessen, die gewünschte Methode der Superklasse einerseits mit *rename* umzubenennen (sodaß man auf den Originalinhalt zugreifen kann) und sie andererseits mit *redefine* inhaltlich an die neue Klasse anzupassen.

Beide Varianten wurden zwecks besserer Vergleichbarkeit implementiert. Mit der ersten Variante schaut die Klasse *JOBDECORATOR* folgendermaßen aus:

```
--* File: VersionA/jobdecorator.e ************************************--

class JOBDECORATOR
```

```
 inherit DECORATOR redefine showme end;
 creation make
 feature {ANY}

  showme is
   do
    Precursor;
    showjob;
   end;

  showjob is
   do
    io.print("My job is ");
    io.print(comp.getjob);
    io.print(".%N");
   end;
end
```

Und hier die zweite Version:

```
--* File: VersionB/jobdecorator.e ***********************************--

class JOBDECORATOR inherit
 DECORATOR
   rename showme as dec_showme
   end;
 DECORATOR
   redefine showme
   select showme
   end;
 creation make
 feature {ANY}

  showme is
   do
    dec_showme;
    showjob;
   end;

  showjob is
   do
    io.print("My job is ");
    io.print(comp.getjob);
    io.print(".%N");
   end;
end
```

Unterschiedlich an diesen beiden Varianten ist vor allem, daß die zweite Art sehr speziell auf Eiffel zugeschnitten ist und gemäß der ursprünglichen Absicht Bertrand Meyers beim Entwurf des Compilers vollkommen anders gestaltet ist als in C++. Nachteilig an der

ersten Variante mit dem *Precursor* ist aber sicherlich, daß man damit nur die gleichnamige Methode aus der Superklasse aufrufen kann; der Aufruf einer anderen Methode ist nicht möglich.
Es ist weiters vorstellbar, daß bei der Klassendefinition mit *rename-redefine* vor allem in größeren Klassen Probleme auftreten könnten, da dann die Übersichtlichkeit durch die Verdoppelung der Methodennamen verloren ginge.

8.3 Smalltalk

In Smalltalk wurde eine kleine Änderung der Methodennamen von *showMe* in C++ und Eiffel auf *printOn* vorgenommen. Die Methode *printOn* existiert bereits in der allgemeinen Superklasse *Object* (sie gibt gewöhnlich den Inhalt beziehungsweise den Zustand einer Klasse aus) und sie kann überschrieben werden, um beispielsweise den Inhalt von klassenspezifischen Variablen auszugeben.
Hier gibt es auch das Problem des Aufrufens einer Methode von einer Superklasse nicht. Es existiert das Schlüsselwort *super* mit dem eine Superklasse einfach angesprochen wird, wie im folgenden Programmteil zu sehen ist:

```
"* File: decorator.st ***********************************************"

Component subclass: #Decorator
        instanceVariableNames: 'comp'
        classVariableNames: ''
        poolDictionaries: ''
        category: ''!

!Decorator class methodsFor: 'instantiation'!

new: component
        | r |
        r := super new.
        r init: component.
        ^r
!!

!Decorator methodsFor: 'decorator'!

init: component
        comp := component
!

printOn: stream
        comp printOn: stream
!

getName
        ^(comp getName)
!
```

```
getJob
        ^(comp getJob)
!

getAddress
        ^(comp getAddress)
!

getAge
        ^(comp getAge)
!!
"****************************************************************"

Decorator subclass: #AddressDecorator
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: ''!
!AddressDecorator methodsFor: 'address'!

printOn: stream
        super printOn: stream.
        'My address is ' printOn: stream.
        comp getAddress printOn: stream.
        '.' printOn: stream
!!
"****************************************************************"

Decorator subclass: #JobDecorator
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: ''!
!JobDecorator methodsFor: 'job'!

printOn: stream
        super printOn: stream.
        'My job is ' printOn: stream.
        comp getJob printOn: stream.
        '.' printOn: stream
!!
```

Der Aufruf im Hauptteil des Programms ist sicherlich nicht so einfach und kurz wie in C++, da in Smalltalk jede Variable einzeln angelegt werden muß, um in die Smalltalk-Umgebung integriert werden zu können.

```
"* File: decmain.st **********************************************"

'Decorator:' printNl !
```

```
Smalltalk at: #pers put: (Person new: 'Inge' address: 'Vienna'
                              job: 'Student' age: 24) !
Smalltalk at: #jobdec put: (JobDecorator new: pers) !
Smalltalk at: #addec put: (AddressDecorator new: jobdec) !
addec printNl !
```

8.4 Zusammenfassung des Pattern *Decorator*

Das besprochene Design Pattern eignet sich bei der Implementation nicht sonderlich für Eiffel. Dies liegt an der bereits behandelten Problematik des Aufrufs einer Methode aus einer Superklasse. Im implementierten Beispiel hat es sich nur um *eine* Methode gehandelt, die in allen Klassen vorgekommen ist (*showMe*). Aber in größeren Anwendungen ist es durchaus vorstellbar, daß viele Methoden mitspielen. In Eiffel würden dann die vielen *redefine*- und *rename*-Klauseln derart zahlreich sein, daß ein Überblick eventuell verloren ginge.
Smalltalk und C++ sind an sich gleich gut für die Implementation dieses Patterns geeignet, wobei C++ noch einen kleinen Vorteil mitbringt: Der Konstruktor der Decorator-Klassen (der bei diesem Pattern immer zumindest einen Zeiger auf eine Komponente beinhaltet) kann hier verschachtelt werden, während dies in Smalltalk nicht möglich ist.

Es ergeben sich bei diesem Design Pattern also folgende Ergebnisse:

	C++	**Eiffel**	**Smalltalk**
Methode *abc* der Superklasse *xy* aufrufen	`xy::abc`	`Precursor` oder `rename abc as orig end; redefine abc select abc end;`	`super abc`
Verschachtelter Konstruktor	√	–	–

Die im folgenden dargestellte Tabelle bewertet die Wichtigkeit einiger Spracheigenschaften für die Implementierung eines *Decorators*:

Spracheigenschaft	Verwendung
Klassendefinition	
Einschränkung der Elementsichtbarkeit	niedrig
abstrakte Klasse	sehr hoch
Vererbung	
einfach (single inheritance)	sehr hoch
mehrfach (multiple inheritance)	sehr niedrig
Gemeinsame Superklasse	
Funktionalität	durchschnittlich
Effizienz	durchschnittlich
Instanzierte Objekte	
Aufwand der Instanzierung (viele Instanzierungen)	durchschnittlich
Möglichkeit der dynamischen Zuweisung	sehr hoch
statischer Typ	hoch

Kapitel 9

Observer

Ein Beispiel eines einfachen Observerobjektes ist das einer Klasse *TimeObserver*, das die aktuelle Uhrzeit überwacht und alle fünf Sekunden eine Nachricht an das *TimeSubject* schickt (der Objektzusammenhang im Beispiel ist aus Abbildung 9.1 ersichtlich). Daraufhin werden alle eingetragenen Observer (*FiveSecondsObserver*, *HalfMinuteObserver*, *TimeObserver*) benachrichtigt, daß sich die aktuelle Zeit geändert hat. Die Instanzen der Klassen *FiveSecondsObserver* und *HalfMinuteObserver* überprüfen, ob die aktuelle Uhrzeit ihren Kriterien entspricht (ob sie durch 5 beziehungsweise durch 30 teilbar ist), holen sich dann die neue Zeit und geben sie aus.
Der eigentliche Sinn dieses Design Patterns liegt darin, einem Objekt ein aufwendiges andauerndes Erfragen des Subjektzustands („Polling“) zu ersparen, um Veränderungen sofort zu erfahren.

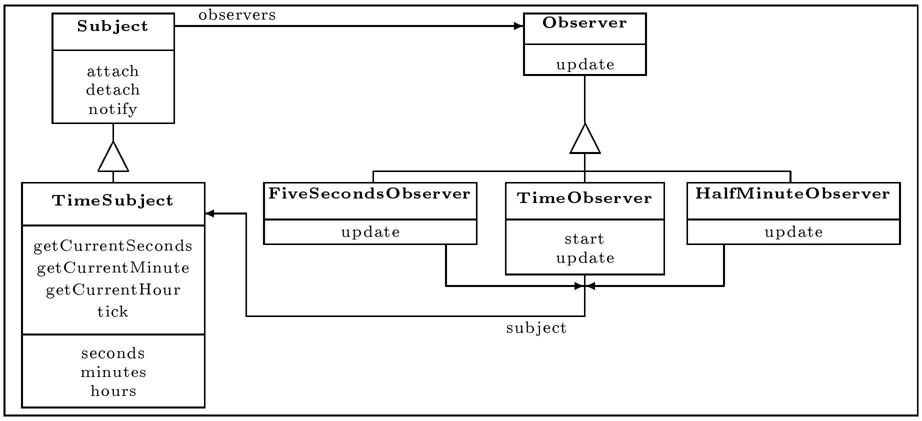

Abbildung 9.1: Objektzusammenhänge in einem Observer

9.1 C++

Das Headerfile der Objekte dieses Programmbeispiels sieht folgendermaßen aus:

```
/* File: objects.h ***************************************************/

#include <stl.h>

class Observer;

class Subject {
 public:
  void attach(Observer *);
  void detach(Observer *);
  void notify();
 private:
  vector<Observer*> observers;
};

class TimeSubject : public Subject {
  public:
   unsigned short getCurrentSecond() { return seconds; }
   unsigned short getCurrentMinute() { return minutes; }
   unsigned short getCurrentHour() { return hours; }
   void tick(int s, int m, int h);
  private:
    int seconds,
        minutes,
        hours;
};

class Observer {
  public:
   Observer(TimeSubject *s);
   virtual ~Observer();
   virtual void update(Subject *s) = 0;
  protected:
   TimeSubject *_pSubject;
};

class TimeObserver : public Observer {
  public:
   TimeObserver(TimeSubject *s) : Observer(s) {}
   ~TimeObserver() {}
   void start();
   virtual void update(Subject *s) {}
};

class FiveSecondsObserver : public Observer {
  public:
```

```
   FiveSecondsObserver(TimeSubject *s) : Observer(s) {}
   ~FiveSecondsObserver() {}
   virtual void update(Subject *s);
};

class HalfMinuteObserver: public Observer {
  public:
    HalfMinuteObserver(TimeSubject *s) : Observer(s) {}
    ~HalfMinuteObserver() {}
    virtual void update(Subject *s);
};
```

Die Klasse *Subject* verwaltet die Eintragungen von Observer-Objekten und besitzt auch eine Methode *notify*, um diese bei Veränderungen zu benachrichtigen. Die Klasse *Observer* auf der anderen Seite besitzt die Methode *update*, mit der die erneuerten Daten eingetragen werden und somit das Objekt aktualisiert wird.
Der Konstruktor der Klasse *Observer* trägt sich selbständig bei der mitgegebenen *Subject*-Instanz in der Liste der zu benachrichtigenden Objekte ein; im Destruktor löscht sie diesen Eintrag wieder.

Ein spezieller Unterschied zwischen den drei Programmiersprachen wird in diesem Design Pattern besonders untersucht, nämlich die Aufteilung des Programmcodes in Headerfile und Implementierungsteil. In C++ wird diese Aufteilung normalerweise durchgeführt, wobei es rein theoretisch auch möglich wäre, die Implementierung gemeinsam mit den Kopfdeklarationen in eine Datei zu schreiben. Dies ist jedoch aufgrund dadurch resultierender mangelhafter Übersichtlichkeit nicht üblich. Methoden mit kurzer Implementierung (wie z.B. solche, in denen nur sofort Variableninhalte retourniert werden) werden dagegegen im Allgemeinen schon im Headerfile implementiert. Beispielsweise wären das in diesem Beispiel Methoden wie *TimeSubject::getCurrentSecond()* oder ähnliche.
Ein Nachteil von Headerfiles ist allerdings, daß eventuell Vorwärtsdeklarationen von Klassen erstellt werden müssen. Das bedeutet, daß ein Klassenname beispielsweise in Parameter- oder Rückgabetypen vorkommt, diese Klasse aber erst nach der anderen deklariert wird. Eine solche Vorwärtsdeklaration von *Observer* ist auch im oben abgebildeten Programmcode zu sehen.
Ein weiterer Nachteil liegt darin, daß Headerfiles und damit Klassendefinitionen eventuell mehrmals im Kompiliervorgang berücksichtigt werden (weil sie in mehreren Implementierungsdateien inkludiert wurden). Daraus resultierende Fehler können meist nur durch bedingte Kompilierung gelöst werden (mit *#ifndef* et cetera, ein Beispiel dazu findet sich im Pattern *Dialog* in der C++-Implementation).

```
/* File: objects.cpp ***********************************************/

#include <time.h>
#include "objects.h"

void Subject::attach(Observer *o) {
  observers.push_back(o);
}

void Subject::detach(Observer *o) {
```

```
  for (unsigned int i=0; i < observers.size(); i++)
    if (observers[i] == o) observers.erase(observers.begin()+i);
}

void Subject::notify() {
  for (unsigned int i = 0; i < observers.size(); i++)
    observers[i]->update(this);
}

void TimeSubject::tick(int s, int m, int h) {
  seconds = s;
  minutes = m;
  hours = h;
  notify();
}

Observer::Observer(TimeSubject *s) {
  _pSubject = s;
  _pSubject->attach(this);
}

Observer::~Observer() {
  _pSubject->detach(this);
}

void TimeObserver::start() {
  int oldSeconds = 61,
      newSeconds = 0;

  while(1) {
    newSeconds = localtime(new time_t(time(NULL)))->tm_sec;
    if (newSeconds % 5 == 0)
      if (oldSeconds != newSeconds) {
        _pSubject->tick(newSeconds,
                        localtime(new time_t(time(NULL)))->tm_min,
                        localtime(new time_t(time(NULL)))->tm_hour);
        oldSeconds = newSeconds;
      }
  }
}

void FiveSecondsObserver::update(Subject *s) {
  if (s == _pSubject)
    if (_pSubject->getCurrentSecond() % 5 == 0)
       cout << "FiveSeconds: " << _pSubject->getCurrentSecond() << endl;
}

void HalfMinuteObserver::update(Subject *s) {
  if (s == _pSubject)
```

```
    if (_pSubject->getCurrentSecond() % 30 == 0)
       cout << "HalfMinute: " << _pSubject->getCurrentSecond() << endl;
}
```

An der Implementierung der Methoden ist allerdings auch zu bemerken, daß in diesem Beispiel kein besonders großer Wert auf die Verwendbarkeit und die Einsatzmöglichkeiten des Beispielprogramms gedacht wurde. Um diesen Mangel zu beseitigen, wäre es nötig, beispielsweise die Klassen *FiveSecondsObserver* und *HalfMinuteObserver* zusätzlich von einem graphischen Uhrenobjekt abzuleiten, da diese Mehrfachvererbung in C++ kein Problem darstellt. Sie könnten dann in den jeweiligen Zeitabständen das Aussehen dieses Objekts verändern.

9.2 Eiffel

Das Problem dieses Design Patterns in Eiffel liegt in der Tatsache, daß im Beispiel eine Zeitfunktion notwendig wäre, um die Zeitabstände für die Klassen *FIVESECONDS-OBSERVER* und *HALFMINUTEOBSERVER* richtig einzusetzen. Daher wurde hier – wie bereits im Pattern *Bridge* – auf eine externe C-Funktion zugegriffen, die diese Zeitfunktionalität bereitstellt.
Die beiden abstrakten Klassen *SUBJECT* und *OBSERVER* sehen folgendermaßen aus:

```
--* File: subject.e ***************************************************--

class SUBJECT
creation make
feature
  make is
   -- Constructor of class SUBJECT
  do
    !!observers.make(0,0);
  end;

  attach(ob: OBSERVER) is
  do
    observers.add_last(ob);
  end;

  detach(ob: OBSERVER) is
  local
    i: INTEGER;
  do
    from
      i := observers.lower;
    until
      i > observers.upper
    loop
      if (observers.item(i) = ob) then
        observers.remove(i);
```

```
      end
      i := i+1;
    end
  end;

  notify is
  local
    i: INTEGER;
  do
    from
      i := observers.lower;
    until
      i > observers.upper
    loop
      if (observers.item(i) /= Void) then
        observers.item(i).update(Current);
      end
      i := i+1;
    end
  end;

feature {NONE}
  observers: ARRAY[OBSERVER];
end

--* File: observer.e ************************************************--

class OBSERVER
creation make
feature
  make(s: TIMESUBJECT) is
    -- Constructor of class OBSERVER
  do
    psubject := s;
    psubject.attach(Current);
  end;

  update(s: SUBJECT) is
    -- updates the observer
  deferred
  end;

feature {NONE}
  psubject: TIMESUBJECT;
end
```

Interessant ist vor allem die Klasse *TIMEOBSERVER*, da dort auf die bereits angesprochenen externen Funktionen zugegriffen wird.

```
--* File: timeobserver.e *******************************************--
```

```
class TIMEOBSERVER inherit
OBSERVER redefine update end;
creation make
feature
  update(s: SUBJECT) is
    -- updates the observer
  do
  end;

  start is
    -- set a new time in the subject
  local
    oldseconds: INTEGER;
    newseconds: INTEGER;
  do
    from
      oldseconds := 61;
    until
      oldseconds < 0
    loop
      newseconds := localseconds;
      if (newseconds \\ 5 = 0) then
        if (oldseconds /= newseconds) then
           psubject.tick(newseconds, localminutes, localhour);
           oldseconds := newseconds;
          end;
      end;
     end;
  end;

feature {NONE}

  localseconds: INTEGER is
    -- get the current seconds
   external "C_WithoutCurrent"
   end;

  localminutes: INTEGER is
    -- get the current minutes
   external "C_WithoutCurrent"
   end;

  localhour: INTEGER is
    -- get the current hours
   external "C_WithoutCurrent"
   end;
end
```

Diese externen Funktionen sind in der Datei *timefunctions.c* zu finden und sehen wie folgt aus:

```
/* File: timefunctions.c                                              */

#include <time.h>

int localseconds(void) {
  time_t tm = time(NULL);
  return localtime(&tm)->tm_sec;
}

int localminutes(void) {
  time_t tm = time(NULL);
  return localtime(&tm)->tm_min;
}

int localhour(void) {
  time_t tm = time(NULL);
  return localtime(&tm)->tm_hour;
}
```

Im Vergleich zu den für C++ geltenden Regeln der Modulaufteilung sieht es in Eiffel so aus, daß grundsätzlich jede Klasse in einer eigenen Datei stehen muß. Dort wird dann allerdings Definition und Implementation zusammengefaßt, sodaß eine völlig andere Aufteilung als in C++ erreicht wird. Probleme wie die Notwendigkeit von Vorwärtsdeklarationen gibt es in Eiffel nicht. Es werden hier auch keine Dateien im Code eingebunden, da der Compiler sich diese anhand der Klassennamen selbst sucht (Klassenname = Dateiname + „.e").

Bezüglich der Mehrfachvererbung gibt es in Eiffel wie auch in C++ keinerlei Probleme.

9.3 Smalltalk

Wie in [Alpert et al., 1998] bereits ausführlich beschrieben, existiert das Observer-Pattern in der Sprache Smalltalk bereits in mehreren zur Smalltalk-Umgebung gehörenden Objekten, die einander bei Veränderungen benachrichtigen. Wie dort beschrieben, stehen Objekte in Smalltalk-80 in drei Arten an Relationen zueinander:

- Ein Objekt wird in einem anderen referenziert (beispielsweise durch Instanzvariable).
- Eine Relation einer Klasse zu seiner Superklasse und seiner Metaklasse.
- Die Möglichkeit eines Objektes, sich als abhängig von einer anderen Klasse zu deklarieren und bei Änderungen der anderen Klasse benachrichtigt zu werden, wird von der absoluten Superklasse in Smalltalk, *Object*, bereitgestellt (näheres zu diesen Methoden in [Goldberg, Robson, 1989], Seite 240f.).

Bei Verwendung dieser Methoden (*addDependent: anObject*, *changed* und *update*) wird die Implementation des Beispiels deutlich verkürzt, da ja die musterspezifischen Teile bereits vorhanden sind. Die abstrakte Klasse *Subject*, die in den beiden anderen Implementierungen die Aufgaben des Design Patterns inne hatte (das Merken aller aktuellen Observer-Objekte und die Methode *notify*), erspart man sich hier, wodurch der Programmcode wie folgt aussieht:

```
"* File: objects.st ************************************************"

Object subclass: #TimeSubject
        instanceVariableNames: 'seconds minutes hours'
        classVariableNames: ''
        poolDictionaries: ''
        category: '' !

!TimeSubject methodsFor: 'timesubject'!

getCurrentSecond
        ^seconds
!

getCurrentMinute
        ^minutes
!

getCurrentHour
        ^hours
!

tick: sec min: min hour: hour
        seconds := sec.
        minutes := min.
        hours := hour.
        self changed
!!
"*******************************************************************"

Object subclass: #Observer
        instanceVariableNames: 'subject'
        classVariableNames: ''
        poolDictionaries: ''
        category: '' !
!Observer class methodsFor: 'Instance creation'!

new: subj
        | r |
        r := super new.
        r init: subj.
       ^r
!!
```

```
!Observer methodsFor: 'observer'!

init: subj
        subject := subj.
        subject addDependent: self
!

update: subj
        self subclassResponsibility
!!
"****************************************************************"

Observer subclass: #TimeObserver
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: '' !
!TimeObserver methodsFor: 'timeobserver' !

start
        [[true] whileTrue:
           [subject tick: (Time now seconds) min: Time now hours
                    hour: (Time now hours).
                    (Delay forSeconds: 1) wait]] value
!

update: sub
!!
"****************************************************************"

Observer subclass: #FiveSecondsObserver
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: '' !
!FiveSecondsObserver methodsFor: 'fivesecondsobserver' !

update: subj
        subj = subject
        ifTrue: [ ((subject getCurrentSecond) \\ 5) = 0
           ifTrue: [ 'FiveSeconds:' print.
              subject getCurrentSecond printNl]]
!!
"****************************************************************"

Observer subclass: #HalfMinuteObserver
        instanceVariableNames: ''
        classVariableNames: ''
```

```
        poolDictionaries: ''
        category: '' !
!HalfMinuteObserver methodsFor: 'halfminuteobserver' !

update: subj
        subj = subject
        ifTrue: [ ((subject getCurrentSecond) \\ 30) = 0
           ifTrue: [ 'HalfMinute:' print.
              subject getCurrentSecond printNl]]
!!
```

Der in C++ und Eiffel begonnene Vergleich der Modulaufteilung wird hier so fortgesetzt, daß es in Smalltalk dem Programmierer freisteht, in welchen Dateien er welche Klassen schreibt (jedoch müssen Definition und Implementation zusammen bleiben). Der Grund dafür ist der, daß der Interpreter von stdin liest beziehungsweise die beim Aufruf als weitere Parameter angegebenen Dateien als Eingabe heranzieht.

Ein großes Problem in Smalltalk ist bei diesem Programmbeispiel die fehlende Mehrfachvererbung. Dadurch ist es im vorliegenden Programmcode nicht einfach möglich, für eine graphische Oberfläche im Programm zusätzlich von Klassen mit entsprechender Funktionalität abzuleiten. Dies ist aber gar nicht notwendig: Da die für einen Observer benötigten Methoden *changed* und *update* bereits in der allgemeinen Superklasse *Object* vorhanden sind, wird jede abgeleitete Klasse – und damit auch solche mit eigenen graphischen Oberflächenelementen – dieses Pattern unterstützen.

9.4 Zusammenfassung des Pattern *Observer*

Bei der Beschreibung der Ergebnisse in diesem Kapitel wurden folgende weitere Spracheigenschaften genauer untersucht:

	C++	**Eiffel**	**Smalltalk**
Modulaufteilung	eigene Definitions- und Implementationsdatei	jede Klasse ein Modul	Aufteilung selbst wählbar
Vorhandene Observermethoden	–	–	`changed` und `update` in Object

Bei der Tabelle der wichtigen Sprachkonstrukte für das Design Pattern *Observer* wird eine Unterscheidung zwischen den Implementierungen für C++ und Eiffel einerseits und Smalltalk andererseits nötig, da sie vollkommen unterschiedlich durchgeführt wurden:

Spracheigenschaft	Verwendung
Klassendefinition	
Einschränkung der Elementsichtbarkeit	hoch
abstrakte Klasse	hoch
Vererbung	
einfach (single inheritance)	sehr hoch
mehrfach (multiple inheritance)	sehr hoch[1]/ sehr niedrig[2]
Gemeinsame Superklasse	
Funktionalität	durchschn. / sehr hoch
Effizienz	durchschn. / sehr hoch
Instanzierte Objekte	
Aufwand der Instanzierung (viele Instanzierungen)	hoch
Möglichkeit der dynamischen Zuweisung	hoch
statischer Typ	hoch

Bei diesem Design Pattern geht *Smalltalk* als eindeutiger Sieger hervor, weil dort die notwendigen Observer-Methoden bereits in GNU-Smalltalk vorhanden sind. Wie in [Alpert et al., 1998] erwähnt, ist ein Observer allerdings kein Teil der Sprache, sondern nur in den meisten Smalltalk-Bibliotheken vorhanden.

[1] C++, Eiffel
[2] Smalltalk

Kapitel 10

Strategy

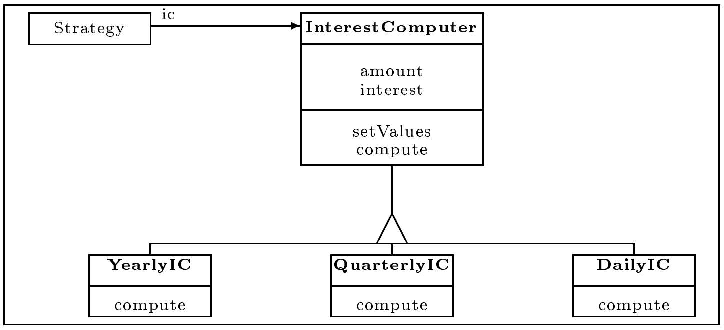

Abbildung 10.1: Implementierungsbeispiel einer Strategy

Als Beispiel der Einsatzmöglichkeiten einer *Strategy* soll ein einfaches Berechnungsprogramm für Zinsen dienen. Mehrere Objekte unterstützen unterschiedliche Arten der Zinsenberechnung (z.B. jährliche, vierteljährliche, tägliche Verzinsung), die in einem Buchhaltungsprogramm Verwendung finden könnten.
Ohne das Design Pattern *Strategy* würde der Algorithmus so aussehen, daß vor jeder Zinsberechnung konditional abhängig das gewünschte Zinsberechnungsverfahren angewandt würde. Mit einer *Strategy* kann diese Unterscheidung bereits bei der Instanzierung des Kontextes gemacht werden, wodurch dann ein Zeiger zum aktuellen Algorithmus-Objekt zeigt. Der Vorteil der zweiten Variante liegt darin, daß der Programmcode vor allem in größeren Projekten übersichtlicher und daher leichter wartbar wird.

10.1 C++

In der abstrakten Klasse *InterestComputer*, die als Superklasse aller Algorithmen dient, wird unter anderem die Berechnungsmethode *compute* definiert:

```
/* File: InterestComputer.h ****************************************/

// Time will be in years
class InterestComputer {
 public:
  InterestComputer() : amount(0), interest(0.0) {}
  void setValues(double amount, double interest);
  virtual double compute(unsigned int time) = 0;
 protected:
  double amount,
         interest;
};
```

Neben dem Konstruktor existiert noch die Methode *setValues*, die bereits in der abstrakten Klasse implementiert ist und mit der die Werte der zu verzinsenden Summe sowie des aktuellen Jahreszinssatzes einzutragen sind.
In den abgeleiteten Klassen *YearlyIC*, *QuarterlyIC* und *DailyIC* wird dann die Methode *compute* jeweils passend implementiert.

```
/* File: SpecialIC.h ***********************************************/

#include "InterestComputer.h"

class YearlyIC : public InterestComputer {
  virtual double compute(unsigned int time);
};

class QuarterlyIC : public InterestComputer {
  virtual double compute(unsigned int time);
};

class DailyIC : public InterestComputer {
  virtual double compute(unsigned int time);
};
```

Am Pattern *Strategy* ist vor allem die Instanzierung der Algorithmenklassen im aufrufenden Programmteil (in C++ in der main-Funktion) interessant: In diesem Beispiel erfolgt die Auswahl der aktuellen Zinsberechnungsmethode als Programmparameter (in C++ und Eiffel; in Smalltalk gibt es die Möglichkeit von Programmparametern nicht). Abhängig von diesem Parameter wird eine der drei oben genannten Klassen instanziert, sodaß der Programmcode folgendermaßen aussieht:

```
/* File: strategy.cpp **********************************************/

#include <iostream.h>
#include "SpecialIC.h"

int main(int argc, char **argv) {
  InterestComputer *ic = NULL;
```

```
  if (argc < 2 || argc > 2) {
    cout << "Syntax: " << argv[0] << " <[year|quarter|day]>" << endl;
    exit(1);
  }
  if (!strcmp(argv[1], "year"))
     ic = new YearlyIC();
  else if (!strcmp(argv[1], "quarter"))
     ic = new QuarterlyIC();
  else if (!strcmp(argv[1], "day"))
     ic = new DailyIC();

  if (ic == NULL) {
    cout << "Syntax: " << argv[0] << "<[year|quarter|day]>" << endl;
    exit(1);
  }
  ic->setValues(1000, 0.02);
  cout.setf(ios::fixed, ios::floatfield);
  cout << "Compute the capital of 1000 ATS bearing 2 percent interest for"
          " 2 years" << endl;
  cout << "Result:" << ic->compute(2) << endl;
  return 0;
}
```

Im Rahmen dieses Design Patterns soll ein wichtiger Unterschied innerhalb der drei zu behandelnden Programmiersprachen bezüglich dynamischem Binden aufgezeigt werden: Wie in der oben dargestellten Datei *strategy.cpp* zu sehen, handelt es sich bei der Variable *ic* um einen Zeiger auf eine Instanz eines *InterestComputer*. Diese kann in der C++ Notation gleich direkt dem Speicherplatz der neu angelegten Instanz zugewiesen werden.

Eine weitere Spracheigenschaft, die hier kurz angesprochen werden soll, ist die der abstrakten Klasse. In C++ wird eine solche in der Form kenntlich gemacht, daß eine oder mehrere Methoden, die mit dem Schlüsselwort *virtual* versehen und damit für ein Überschreiben der Implementierung in einer abgeleiteten Klasse freigegeben sind zusätzlich noch mit der Zuweisung „= 0“ versehen werden (dann werden sie *pure virtual* genannt). Ein Beispiel für eine derartige Methode ist *InterestComputer::compute*, die auf Seite 84 im Abdruck der Datei *InterestComputer.h* zu sehen ist.

10.2 Eiffel

Die abstrakte Klasse *INTERESTCOMPUTER* hat die gleichen Variablen und Methoden wie in C++ und sieht folgendermaßen aus:

```
--* File: interestcomputer.e ****************************************--

class INTERESTCOMPUTER
feature {ANY}
  setvalues(a: REAL;i: REAL) is
   do
```

```
    amount := a;
    interest := i;
   end;

  compute(time: INTEGER): DOUBLE is
   deferred
   end;
feature {NONE}
  amount: DOUBLE;
  interest: DOUBLE;
end
```

Gleich in dieser Klasse kann ein Unterschied in der Notation einer abstrakten Klasse zwischen C++ und Eiffel erkannt werden: In Eiffel existiert das Schlüsselwort *deferred*, mit dem angezeigt wird, daß die Implementierung einer Methode an eine abgeleitete Klasse weitergegeben wird und dort erfolgen soll. An dieser Stelle ist eventuell interessant, daß in der Subklasse diese Methode dann nicht in der *redefine*-Klausel vorkommen darf (da sie ja auch nicht neu sondern erstmals implementiert wird).
Diesen Sachverhalt kann man beispielsweise in der Klasse *QUARTERLYIC* beobachten, in dem eine vierteljährliche Zinsabrechnung durchgeführt wird.

```
--* File: quarterlyic.e ********************************************--

class QUARTERLYIC
inherit INTERESTCOMPUTER
feature
  compute(time: INTEGER): DOUBLE is
   do
    Result := (interest+1.0)^(time*4)*amount;
   end
end
```

Die Hauptklasse dieses Beispiels *STRATEGY* erhält wie auch in C++ eine Unterscheidungsmöglichkeit aus den Programmparametern. Wie im folgenden dargestellt, funktioniert die dynamische Zuweisung der gewünschten Klasseninstanz nicht ganz so elegant wie in C++[1]:

```
--* File: strategy.e ***********************************************--

class STRATEGY
creation make
feature
  make is
   -- Constructor of class STRATEGY
  require
    argument_count = 1 and (argument(1).is_equal("year") or
                            argument(1).is_equal("quarter") or
                            argument(1).is_equal("day"))
```

[1] Unter „elegant“ wird in diesem Zusammenhang verstanden, daß möglichst gut lesbarer Programmcode mit wenig verwendeten Variablen erzeugt wird.

```
  local
    yic: YEARLYIC;
    qic: QUARTERLYIC;
    dic: DAILYIC;
    ic: INTERESTCOMPUTER;
  do
    if argument(1).is_equal("year") then
      !!yic;
      ic := yic;
    elseif argument(1).is_equal("quarter") then
      !!qic;
      ic := qic;
    elseif argument(1).is_equal("day") then
      !!dic;
      ic := dic;
    end
   io.print("Compute the capital of 1000 ATS bearing 2 percent interest for"
            " 2 years%N");
   ic.setvalues(1000, 0.02);
   print("Result:");
   print(ic.compute(2).to_string);
   print("%N");
  end;
end
```

Zur Argumentbehandlung ist zu erwähnen, daß es im Eiffel-Standard auch die Möglichkeit gibt, der Methode *make* der Hauptklasse einen Parameter zu geben. Der GNU-Compiler unterstützt diese Funktion allerdings nicht. Stattdessen kann man die Programmparameter aus festgelegten Variablen (*argument_count*, *argument(1)*, et cetera) entnehmen.
Bezüglich der dynamischen Zuweisung zur Variable *ic* muß in Eiffel zunächst die Instanz erzeugt und dann erst zugewiesen werden. Verglichen mit C++ ist diese Variante also nicht ganz so elegant.

10.3 Smalltalk

In Smalltalk existiert auch eine Methodik, wie abstrakte Klassen als solche gekennzeichnet werden können: Mit der Phrase *self subclassResponsibility* wird jede Methode versehen, deren Implementation erst in der abgeleiteten Klasse eingefügt werden soll.

```
"* File: InterestComputer.st ****************************************"

Object subclass: #InterestComputer
        instanceVariableNames: 'amount interest'
        classVariableNames: ''
        poolDictionaries: ''
        category: '' !
!InterestComputer methodsFor: 'abstract'!
```

```
setValues: am and: in
        amount := am.
        interest := in
!

compute: time
        self subclassResponsibility
!!
"****************************************************************"

InterestComputer subclass: #YearlyIC
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: ''!
!YearlyIC methodsFor: 'year'!

compute: time
        ^(amount*((1+interest) raisedTo: (time)))
!!
"****************************************************************"

InterestComputer subclass: #QuarterlyIC
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: ''!
!QuarterlyIC methodsFor: 'quarter'!

compute: time
        ^(amount*((1+interest) raisedTo: (time*4)))
!!
"****************************************************************"

InterestComputer subclass: #DailyIC
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: ''!
!DailyIC methodsFor: 'day'!

compute: time
        ^(amount*((1+interest) raisedTo: (time*360)))
!!
```

In der obigen Dateiansicht ist die Klasse *InterestComputer* als abstrakt gekennzeichnet. Die weiteren Klassen dienen zur unterschiedlichen Implementation der Zinsberechnung (jährlich, vierteljährlich, täglich).

In Smalltalk gibt es leider keine Programmparameter, da sämtliche Befehlszeilen des Programms direkt dem Interpreter übergeben werden, der sie sofort behandelt. Daher wird im folgenden Dateiabdruck des „Treiberprogramms" als Parameter *quarter* gesetzt. Abhängig von diesem Wert wird danach – mit einer ähnlich eleganten Syntax wie in C++ der Variable *ic* eine aktuelle Instanz der Zinsberechnungsklasse zugewiesen.

```
"* File: strategy.st ************************************************"

'Strategy:' printNl !

Smalltalk at: #param put: 'quarter'!
Smalltalk at: #ic put: 0!

(param sameAs: 'year')    ifTrue: [ ic := YearlyIC new ] !
(param sameAs: 'quarter') ifTrue: [ ic := QuarterlyIC new ] !
(param sameAs: 'day')     ifTrue: [ ic := DailyIC new ] !

ic setValues: 1000 and: 0.02 !
'Compute the capital of 1000 ATS bearing 2 percent interest for 2 years'
 printNl!
'Result: ' print !
^(ic compute: 2) printNl !
```

10.4 Zusammenfassung des Pattern *Strategy*

Dieses Design Pattern ist grundsätzlich eigentlich sowohl für C++, als auch für Eiffel und Smalltalk gleichermaßen geeignet, da es in allen diesen Sprachen das notwendige Sprachkonstrukt – dynamisches Binden von Variablen zu Objekten – gibt.
Unterschiede in den Sprachen sind während der Besprechung dieses Patterns aber trotzdem herausgekommen, wie in der folgenden Tabelle ersichtlich:

	C++	**Eiffel**	**Smalltalk**
Programmparameter	$\surd$	$\surd$	–

Einige Worte sollen noch zum Thema des dynamischen Bindens gesagt werden, die die Frage der Eleganz des Programmcodes von einer unterschiedlichen Seite beleuchten als in den Beispielen dieses Patterns:
Ausgehend von der Annahme, daß im Design Pattern *Strategy* nicht wie in den obigen Beispielen die Entscheidung für das aktuelle Algorithmusobjekt aus den Programmparametern hervorgehen muß und daher zu Laufzeitbeginn feststeht, sondern eventuell im Programm mehrfach wechselt, wird folgendes deutlich:

- In C++ könnte der Programmcode so aussehen, daß zwar die erste Zuweisung elegant erscheint; bei einer erneuten Zuweisung verfällt allerdings der Zeiger zur vorigen Instanz und sollte daher zuvor explizit gelöscht werden (in C++ gibt es im Allgemeinen keine *garbage collection*), sodaß der Programmcode beispielsweise folgendermaßen aussehen könnte (die Klassennamen sind die gleichen wie im obigen Programmbeispiel):

```
..
InterestComputer *ic;
...
ic = new QuarterlyIC();
...
if (ic != NULL) delete ic;
ic = new DailyIC();
...
```

- In Eiffel ist für diesen Fall eine andere Lösung üblich, bei der zu Programmbeginn alle in Frage kommenden Klassen instanziert werden und dann je nach Bedarf die aktuelle zugewiesen wird:

```
...
local
  ic: INTERESTCOMPUTER;
  yic: YEARLYIC;
  qic: QUARTERLYIC;
  dic: DAILYIC;
end;
...
!!yic; !!qic; !!dic;
...
ic := qic;
...
ic := dic;
...
```

- In Smalltalk existiert die für diesen Gebrauch eleganteste Lösung, da sie wie in C++ nur eine Variable benutzt, aber auch keinen expliziten Aufruf eines Destruktors benötigt:

```
...
Smalltalk at: #ic put: 0!
...
ic := QuarterlyIC new!
...
ic := DailyIC new!
...
```

Und schlußendlich wird in der folgenden Tabelle noch die Wichtigkeit einiger Sprachkonstrukte für dieses Design Pattern angegeben:

Spracheigenschaft	**Verwendung**
Klassendefinition	
Einschränkung der Elementsichtbarkeit	niedrig
abstrakte Klasse	sehr hoch
Vererbung	
einfach (single inheritance)	sehr hoch
mehrfach (multiple inheritance)	niedrig
Gemeinsame Superklasse	
Funktionalität	durchschnittlich
Effizienz	durchschnittlich
Instanzierte Objekte	
Aufwand der Instanzierung (viele Instanzierungen)	sehr niedrig
Möglichkeit der dynamischen Zuweisung	sehr hoch
statischer Typ	durchschnittlich

Kapitel 11

Zusammenfassung

In dieser Diplomarbeit wurde anhand von Design Patterns versucht, die drei Programmiersprachen C++, Eiffel und Smalltalk miteinander bezüglich ihrer unterschiedlichen Spracheigenschaften zu vergleichen.
Mittel zum Zweck war die Implementierung ausgesuchter Design Patterns, die sich mit Vererbungsmechanismen beschäftigen. Eine Zusammenfassung der Ergebnisse für jedes Design Pattern wird im folgenden wiedergegeben:

11.1 Design Patterns

Bridge Bei der Implementierung dieses Beispiels wurde vor allem darauf geachtet, welche Sprache Fehler vermeiden hilft, die durch unerfüllte Bedingungen in einer Klasse passieren können. In der konkreten Anwendung handelt es sich um einen Zeiger auf eine Klasseninstanz, die im Konstruktor richtig zugewiesen werden muß. Im negativen Fall könnten später in anderen Methoden schwere Laufzeitfehler auftreten. Weiters wurde anhand dieses Design Patterns untersucht, welche Form das von außen aufrufbare Hauptprogramm hat, um daran den Unterschied zwischen *hybriden* und *reinen* objektorientierten Programmiersprachen zu sehen.

	C++	**Eiffel**	**Smalltalk**
Fehlervermeidung	Exceptions	Assertions	–
Hauptprogramm	main-Function	Hauptklasse	einzelne Befehle

Chain of Responsibility In der Diskussion über die Implementation der *Chain of Responsibility* wurde unter anderem untersucht, welche Möglichkeiten in den drei Sprachen bezüglich Zeichenkettenbearbeitung liegen. Hier hat C++ eindeutig am besten abgeschnitten, da es auch die bereits große Auswahl an Instrumenten aus C besitzt. Eine weitere Untersuchung bezog sich auf die Darstellung eines ungültigen Werts. Das wenig überraschende Ergebnis hat ergeben, daß alle drei Sprachen dies unterstützen (mit unterschiedlichen Schlüsselwörtern).
Schließlich wurde noch bemerkt, daß es in Smalltalk und Eiffel eine allgemeine Superklasse gibt, von der immer abgeleitet wird, beziehungsweise die eine implizite Su-

perklasse aller definierten Klassen ist. In Smalltalk ergibt sich dadurch ein System von Klassen, und jede neue Klasse muß an einer Stelle in diesen Baum eingehängt werden.

	C++	**Eiffel**	**Smalltalk**
Zeichenkettenbearbeitung	viele Funktionen	wenig Funktionen	wenig Funktionen
Ungültiger Wert	NULL	Void	nil

Composite Im Pattern *Composite* wurde die Aufmerksamkeit auf die Container gelegt. Hier hat sich herausgestellt, daß es sowohl in C++, als auch in Eiffel und Smalltalk zumindest eine dynamisch erweiterbare Containerklasse gibt. In C++ und Eiffel ist sie allerdings zusätzlich an einen konstanten Elementtyp gebunden, während in Smalltalk durch die dynamische Typisierung Elemente unterschiedlichen Typs in einen Container eingepackt werden können.
Schließlich hat sich bei diesem Design Pattern herausgestellt, daß Eiffel am besten für die Implementation geeignet ist, da in C++ aufgrund mangelnder automatischer Speicherverwaltung (*garbage collection*) ein Destruktor in der Klasse notwendig ist, um den Container explizit zu entleeren. In Smalltalk dagegen ist keine Typsicherheit gegeben.

	C++	**Eiffel**	**Smalltalk**
Container	vector	ARRAY	OrderedCollection
Konstanter Elementtyp	√	√	–
Dynamische Größenänderung	√	√	√

Decorator Auffällig an diesem Beispiel war beim Programmieren, daß es in Eiffel (genauer in GNU SmallEiffel) zwei Arten gibt, um eine Methode einer Superklasse aufzurufen.

- Mit einem Precursor, der allerdings im Eiffel-Standard nicht vorgesehen ist, wird einfach die gleichnamige Methode der nächsten Superklasse aufgerufen.
- Durch Verwendung von *rename* und *redefine* wird die abgeleitete Methode umbenannt und weiters neu implementiert, sodaß in der aktuellen Klasse alle beide Varianten vorhanden sind.

In diesem Design Pattern wurde weiters untersucht, ob es möglich ist, einen Konstruktor zu verschachteln. Dies würde eine Kompaktheit des Programmcodes mit sich bringen, hätte aber auch Nachteile wie beispielsweise schlechtere Lesbarkeit.

	C++	**Eiffel**	**Smalltalk**

Methode *abc* der Superklasse *xy* aufrufen	`xy::abc`	Precursor oder
		`rename abc as`
		`orig end;`
		`redefine abc`
		`select abc`
		`end;`
			`super abc`

Verschachtelter Konstruktor	√	–	–

Observer Da in GNU Smalltalk die notwendigen Methoden für einen Observer bereits in der Superklasse *Object* enthalten sind und daher in dieser Sprache nicht mehr extra programmiert werden müssen, ist sicherlich Smalltalk am besten für dieses Design Pattern geeignet.

Ansonsten wurde bei den Beispielen noch untersucht, wie es in den drei Programmiersprachen um die Modulaufteilung steht. Hier haben sich drei völlig unterschiedliche Konzepte gezeigt.

Die Mehrfachvererbung, die gerade bei diesem Design Pattern in einem wirklichen Projekt brauchbar wäre (Ableitung einerseits von der abstrakten Klasse – entweder Subject oder Observer – und andererseits von einer graphischen Oberflächenklasse), ist in C++ und Eiffel enthalten, in Smalltalk leider nicht. Da die Methoden für einen Observer aber in jeder neuen Klasse bereits enthalten sind (sie ist auf jeden Fall von *Object* abgeleitet), genügt es, wenn sie sich von einer Klasse mit Oberflächenelementen ableitet.

	C++	**Eiffel**	**Smalltalk**
Modulaufteilung	eigene Definitions- und Implementationsdatei	jede Klasse ein Modul	Aufteilung selbst wählbar
Vorhandene Observermethoden	–	–	`changed` und `update` in Object

Strategy Dieses Design Pattern ist für alle drei Sprachen gleichermaßen geeignet. Das nötige dynamische Binden der Variablen (die auf die Klasse mit dem aktuellen Algorithmus zeigt), wird sowohl in C++ als auch in Eiffel und Smalltalk unterstützt. Unterschiede bestehen nur hinsichtlich der Tatsache, daß es sich in C++ bei dieser Variable um einen Zeiger handeln muß; ansonsten ist nur eine Referenz nötig.

In diesem Implementierungsbeispiel wurde auch untersucht, ob Programmparameter in den Sprachen vorgesehen sind. Hier hat sich gezeigt, daß C++ und Eiffel diesbezüglich gut ausgestattet sind; in Smalltalk existieren aufgrund des Interpreters keine Programmparameter (das Hauptprogramm besteht ja auch, wie oben angeführt, nur aus mehreren Befehlen, die dem Interpreter eingegeben werden, nachdem er die Klassen in seinen Objektbaum integriert hat).

	C++	**Eiffel**	**Smalltalk**
Programmparameter	√	√	–

11.2 Programmiersprachen

Die im Kapitel 3 über Programmiersprachen bereits ausgefüllte Tabelle wird hier noch einmal wiedergegeben. Damit soll ein abschließender Vergleich zu den Tabellen in den Zusammenfassungen der Design Patterns hergestellt werden.

Spracheigenschaft	**C++**	**Eiffel**	**Smalltalk**
Klassendefinition			
Einschränkung der Elementsichtbarkeit	++	+	−
abstrakte Klasse	++	++	++
Vererbung			
einfach (single inheritance)	++	++	++
mehrfach (multiple inheritance)	++	++	−−
Gemeinsame Superklasse			
Funktionalität	∼	+	+
Effizienz	∼	+	+
Instanzierte Objekte			
Aufwand der Instanzierung	−−	−	−
Möglichkeit der dynamischen Zuweisung	++	++	++
statischer Typ	++	++	−−

Zur Abrundung dieser Arbeit werden die Durchschnittswerte über die Wichtigkeit der Kriterien bei den Design Patterns berechnet. Dadurch kann festgestellt werden, ob die untersuchten Programmiersprachen für diese Probleme bei objektiver Betrachtung geeignet sind. Diese Durchschnittswerte wurden jeweils zur nächsten, höherwertigen Bewertung aufgerundet:

Spracheigenschaft	**Durchschnitt**
Klassendefinition	
Einschränkung der Elementsichtbarkeit	hoch
abstrakte Klasse	sehr hoch
Vererbung	
einfach (single inheritance)	sehr hoch
mehrfach (multiple inheritance)	durchschn.[1]/ niedrig[2]
Gemeinsame Superklasse	
Funktionalität	durchschn. / hoch
Effizienz	durchschn. / hoch
Instanzierte Objekte	
Aufwand der Instanzierung	niedrig
Möglichkeit der dynamischen Zuweisung	sehr hoch
statischer Typ	hoch

[1] C++, Eiffel
[2] Smalltalk

An dieser zusammenfassenden Tabelle ist zu sehen, daß bei drei Kriterien zwischen den hier eingetragenen Werten und denen bei den Eigenschaften der Programmiersprachen Übereinstimmung herrscht. Dabei handelt es sich um die Sprachkonzepte von *abstrakten Klassen*, *Vererbung* und *dynamischem Binden*. Auch die Kriterien, bei denen eine Unterscheidung zwischen C++ und Eiffel einerseits und Smalltalk andererseits durchgeführt wurde (diese Trennung stammt aus der Bewertung des Design Patterns *Observer*), weisen im Vergleich zwischen den Tabellen ähnliche Werte auf, die eine Implementation möglich machen. Einzig bei den Kriterien der *Einschränkung der Elementsichtbarkeit* und dem *statischen Typ* bietet Smalltalk nicht die geforderten Fähigkeiten an. Diese beiden Kriterien dienen im allgemeinen der Sicherheit und sind daher eventuell ein entbehrlicher Bestandteil.

Die in den vorigen Kapiteln untersuchten und beschriebenen Unterschiede lassen allerdings trotzdem öfters die Wahl auf eine Sprache fallen, die bei erster Betrachtung gar nicht so speziell geeignet erscheint. Daher muß abschließend noch bemerkt werden, daß der Einsatz einer Sprache abgesehen vom verwendeten Design Pattern und der Klassenstruktur auch von der tatsächlichen Aufgabe abhängt. Dies liegt daran, daß viele Entscheidungen zugunsten einer Sprache aufgrund von zusätzlichen Spracheigenschaften (Methoden zur Zeichenkettenbearbeitung, et cetera) getroffen wurden.

Literaturverzeichnis

[Alexander et al., 1977] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel: *A Pattern Language*, Oxford University Press, New York, 1977.

[Alpert et al., 1998] Sherman R.Alpert, Kyle Brown, Bobby Woolf: *The Design Patterns Smalltalk Companion*, Addison Wesley Longman Inc., ISBN 0-201-18462-1, 1998

[Bosch, 1997] Jan Bosch: *Design Patterns & Frameworks: On the Issue of Language Support*, ECOOP 97 Workshop Reader, Proceedings, June 1997

[Coplien, 1994] James O.Coplien: *Software Design Patterns: Common Questions and Answers*, Proceedings of Object Expo New York, Juni 1994

[Cunningham, Beck, 1987] Ward Cunningham, Kent Beck: *Using Pattern Languages for Object-Oriented Programs*, OOPSLA 87 in Orlando, 1987

[Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley Longman, Inc., ISBN 0-201-63361-2, 1995.

[Ghezzi, Jazayeri, 1996] Carlo Ghezzi, Mehdi Jazayeri: *Programming language concepts*, John Wiley & Sons, Third edition 1996

[Goldberg, Robson, 1989] Adele Goldberg and David Robson: *Smalltalk-80: The Language*, Addison Wesley Publishing Company, ISBN 0-201-13688-0, 1989

[Hetzel-Herzog, 1994] Walter Hetzel-Herzog: *Objektorientierte Softwaretechnik: Integration und Realisierung in der betrieblichen DV-Praxis*, Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, ISBN 3-528-05341-0, 1994

[LaLonde, Pugh, 1991] Wilf LaLonde, John Pugh: *Subclassing ≠ subtyping ≠ Is-a*, Journal of Object-Oriented Programming, Vol. 3, No. 5, January 1991

[Meyer, 1988] Bertrand Meyer: *Object-oriented Software construction*, Prentice Hall International (UK) Ltd., ISBN 0-13-629049-3, 1988

[Meyer, 1992] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall Europe, ISBN 0-13-247925-7, 1992

[Meyer, 1995] Bertrand Meyer: *Static Typing*, OOPS Messenger Vol. 6, No. 4, October 1995

[Riehle, Zullighoven, 1996] Dirk Riehle, Heinz Züllighoven: *Understanding and Using Patterns in Software Development*, aus: *TAPOS-Theory and Practice of Object Systems*, Vol. 2, John Wiley New York, 1996

[Singh, 1995] Ghan Bir Singh: *Single Versus Multiple Inheritance in Object Oriented Programming*, OOPS Messenger Vol.6, No. 1, January 1995

[Stroustrup, 1995] Bjarne Stroustrup: *Why C++ is not just an Object-Oriented Programming Language*, OOPS Messenger 6(4) Oct. 1995

[Stroustrup, 1997] Bjarne Stroustrup: *The C++ Programming Language*, Third Edition, Addison Wesley, 1997, ISBN 0-201-88954-4

[Wegner, 1990] Peter Wegner: *Concepts and Paradigms of Object-Oriented Programing*, OOPS Messenger Vol. 1, No. 1, August 1990

[Weidl, 1996] Johannes Weidl: *The Standard Template Library Tutorial*, Information Systems Institute, Distributed Systems Department, Technical University Vienna, 1996

[Wirfs-Brock et al, 1990] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener: *Designing Object-Oriented Software*, Prentice Hall, Inc., ISBN 0-13-629825-7, 1990

Index