Skriptum zu

Objektorientierte Programmierung

Sommersemester 2009

Raimund Kirner
Technische Universität Wien
Institut für Technische Informatik
http://ti.tuwien.ac.at/rts/teaching/courses/oop

Institut für Technische Informatik

Dieses Skriptum für *Objektorientierte Programmierung* mit C++ wurde im Jahre 2009 erstellt, indem ein bereits bestehendes, auf Java basierendes, Skriptum adaptiert wurde. Die Vorlage dieses Skriptums wurde von Franz Puntigam erstellt:

Franz Puntigam
Technische Universität Wien
Institut für Computersprachen
http://www.complang.tuwien.ac.at/franz/objektorientiert.html

Das Umschreiben der Beispiele und Konzepte von Java nach C++ wurde im März 2009 von Markus Raab durchgeführt:

Markus Raab
http://www.markus-raab.org

Anregungen und Kritiken bitte an: oop@markus-raab.org

Inhaltsverzeichnis

1 Grundlagen und Ziele **11**

1.1 Konzepte objektorientierter Programmierung 12

1.1.1 Objekte 12

1.1.2 Klassen 15

1.1.3 Werkzeuge für C++ 21

1.1.4 Polymorphismus 27

1.1.5 Vererbung 30

1.2 Qualität in der Programmierung 35

1.2.1 Qualität von Programmen 35

1.2.2 Effizienz der Programmerstellung und Wartung . . 39

1.3 Rezept für gute Programme 41

1.3.1 Zusammenhalt und Kopplung 42

1.3.2 Wiederverwendung 45

1.3.3 Entwurfsmuster 47

1.4 Paradigmen der Programmierung 49

1.4.1 Imperative Programmierung 49

1.4.2 Deklarative Programmierung 50

1.4.3 Paradigmen für Modularisierungseinheiten 52

1.5 Wiederholungsfragen 55

2 Enthaltender Polymorphismus und Vererbung **57**

2.1 Das Ersetzbarkeitsprinzip 57

2.1.1 Untertypen und Schnittstellen 58

2.1.2 Untertypen und Codewiederverwendung 63

2.1.3 Dynamisches Binden 67

2.2 Ersetzbarkeit und Objektverhalten 71

2.2.1 Client-Server-Beziehungen 71

2.2.2 Untertypen und Verhalten 77

2.2.3 Abstrakte Klassen 82
2.3 Vererbung versus Ersetzbarkeit 85
2.3.1 Reale Welt versus Vererbung versus Ersetzbarkeit . 85
2.3.2 Vererbung und Codewiederverwendung 88
2.4 Exkurs: Klassen und Vererbung in C++ 93
2.4.1 Speicherplatzverwaltung 93
2.4.2 Klassen in C++ 94
2.4.3 Benutzerdefinierte Typen 100
2.4.4 Zugriffskontrolle in C++ 102
2.5 Wiederholungsfragen 104

3 Generizität und Ad-hoc-Polymorphismus 107
3.1 Generizität 107
3.1.1 Wozu Generizität? 108
3.1.2 Funktions-Templates 109
3.1.3 Klassen-Templates 115
3.1.4 Iteratoren 123
3.2 Verwendung von Generizität im Allgemeinen 126
3.2.1 Richtlinien für die Verwendung von Generizität . . 126
3.2.2 Arten der Generizität 131
3.3 Typabfragen und Typumwandlungen 135
3.3.1 Explizite Typkonvertierung 135
3.3.2 Verwendung dynamischer Typinformation 135
3.3.3 Kovariante Probleme 139
3.4 Überladen versus Multimethoden 143
3.4.1 Unterschiede zwischen Überladen und Multimethoden 143
3.4.2 Simulation von Multimethoden 147
3.5 Ausnahmebehandlung 150
3.5.1 Ausnahmebehandlung in C++ 150
3.5.2 Einsatz von Ausnahmebehandlungen 155
3.6 Wiederholungsfragen 159

4 Softwareentwurfsmuster 161
4.1 Erzeugende Entwurfsmuster 162
4.1.1 Factory Method 162
4.1.2 Prototype 166
4.1.3 Singleton 170
4.2 Strukturelle Entwurfsmuster 173
4.2.1 Decorator 173

4.2.2 Proxy 177
4.3 Entwurfsmuster für Verhalten 180
4.3.1 Iterator 180
4.3.2 Template Method 185
4.4 Wiederholungsfragen 187

Vorwort

„Objektorientierte Programmierung“ ist eine Vorlesung mit Laborübung im Umfang von zwei Semesterwochenstunden an der TU Wien. Unter anderem werden folgende Themenbereiche der objektorientierten Programmierung an Hand von C++ behandelt:

- Datenabstraktion, Klassenhierarchien, Polymorphismus
- Objektschnittstellen und Zusicherungen (Schwerpunkt)
- Vererbung und Untertyprelationen (Schwerpunkt)
- Generizität (Schwerpunkt)
- Ausnahmebehandlung
- Implementierung einiger gängiger Entwurfsmuster

TeilnehmerInnen an der Lehrveranstaltung sollen einen Überblick über die wichtigsten Konzepte objektorientierter Programmierung bekommen und diese Konzepte so einzusetzen lernen, dass qualitativ hochwertige und gut wartbare Software entsteht. Subtyping (auf der Basis von Objektschnittstellen mit Zusicherungen) und Generizität bilden Schwerpunkte, die am Ende der Lehrveranstaltung jedenfalls beherrscht werden müssen. Praktische Programmiererfahrung in einer beliebigen Programmiersprache wird vorausgesetzt. C++-Vorkenntnisse sind sehr hilfreich. C-Vorkenntnisse können auch helfen, dann sind aber Anmerkungen zu C++ unbedingt zu lesen, da es in C++ oftmals bessere und einfachere Möglichkeiten gibt das selbe zu tun. Das Erlernen von C++ im Selbststudium parallel zur Lehrveranstaltung ist möglich.

Das erste Kapitel dieses Skriptums

- führt grundlegende objektorientierte Programmierkonzepte ein,
- gibt einen Überblick über Qualität in der Programmierung,

- weist darauf hin, mit welchen Problemen man in der objektorientierten Programmierung rechnen muss und wie man diese lösen kann,
- und klassifiziert Programmiersprachen anhand ihrer Paradigmen, um eine Einordnung der objektorientierten Sprachen in die Vielfalt an Programmiersprachen zu erleichtern.

Das zweite Kapitel beschäftigt sich mit dem besonders wichtigen Themenkomplex des enthaltenden Polymorphismus zusammen mit Klassenhierarchien, Untertypbeziehungen und Vererbung. Vor allem das Ersetzbarkeitsprinzip und Zusicherungen (Design by Contract) werden ausführlich behandelt. Eine Beschreibung der Umsetzung entsprechender Konzepte in C++ rundet das zweite Kapitel ab.

Das dritte Kapitel ist neben weiteren Formen des Polymorphismus vor allem der Generizität gewidmet. Es werden Programmiertechniken vorgestellt, die entsprechende Problemstellungen auch bei fehlender Sprachunterstützung für Generizität, kovariante Spezialisierungen und mehrfaches dynamisches Binden lösen können. Das dritte Kapitel wird mit dem, nicht direkt im Zusammenhang stehenden, Thema Ausnahmebehandlung abgeschlossen.

Das letzte Kapitel stellt eine Auswahl an häufig verwendeten Entwurfsmustern vor. Nebenbei werden praktische Tipps und Tricks in der objektorientierten Programmierung gegeben.

Die Lehrveranstaltung soll einen Überblick über Konzepte der objektorientierten Programmierung, Zusammenhänge zwischen ihnen, mögliche Schwierigkeiten sowie Ansätze zu deren Beseitigung vermitteln. Keinesfalls soll sie als C++ -Kurs verstanden werden. Insbesondere die umfangreichen Klassenbibliotheken, die in der C++ -Programmierung Verwendung finden, werden nicht behandelt. Es ist aber erlaubt, und sogar erwünscht, die Standardlibrary und die `boost` Library `http://www.boost.org/` intensiv einzusetzen.

C++ ist eine von ANSI/ISO standardisierte Sprache. In der Vorlesung und im Skriptum wird der aktuelle Sprachstandard `ANSI ISO IEC 14882 2003` verwendet.

Informationen zu C++ gibt es unter anderem im world wide web, zum Beispiel unter `http://www.research.att.com/~bs/C++.html`. Auch mehrere Bücher, beispielsweise „Thinking in C++“, sind im www gratis verfügbar.

Als Einführungsliteratur kann [KM00], welches sehr mit Beispielen arbeitet oder [LLM05], welches eher einen traditionellen und umfassenden

Zugang zu der Sprache C++ gibt, verwendet werden. Die ursprüngliche C++ Sprachbeschreibung [Str00] wurde mittlerweilen sehr stark überarbeitet und ist auch als anspruchsvolles Lehrbuch verwendbar. Bücher für Tipps wie C++ Programme verbessert werden können [Mey05][Mey95][MS01] sollten beim ernsthaften Einsatz von C++ unbedingt bekannt sein, sind aber für die Lehrveranstaltung keine Voraussetzung. Vertiefend wird zusätzlich noch das Buch [Wil06] empfohlen. Der abgeprüfte Inhalt kann vollständig aus diesem Skriptum entnommen werden, aber um tatsächlich C++ zu erlernen und die Beispielaufgaben zu lösen werden zusätzlich ein oder mehrere Bücher der Literatur unbedingt benötigt. Direkte Hilfe bekommen Studenten in den Tutorenstunden an der Universität und im Forum. Allgemein wird aber auch in Medien wie Newsgroups (`de.comp.lang.iso-c++`) oder IRC (`##c++` auf `freenode`) bei C++-spezifischen Problemen weitergeholfen. Eine umfangreiche Sammlung von Verweisen auf für die objektorientierte Programmierung relevante Seiten befindet sich unter `http://www.cetus-links.org/`.

Viel Erfolg bei der Teilnahme an der Lehrveranstaltung!

Raimund Kirner, Markus Raab

Kapitel 1

Grundlagen und Ziele

Immer mehr Unternehmen der Softwarebranche steigen auf objektorientierte Programmierung um. Ein großer Teil der SoftwareentwicklerInnen verwendet derzeit bereits Methoden der objektorientierten Programmierung. Dabei stellt sich die Frage, welche Vorteile die objektorientierte Programmierung gegenüber anderen Paradigmen bietet oder zumindest erwarten lässt, die den umfangreichen Einsatz in der Praxis rechtfertigen. Solche erhofften Vorteile sowie mögliche Gefahren wollen wir in diesem Kapitel betrachten. Die Stellung der objektorientierten Programmierung unter der Vielzahl existierender Programmierparadigmen wollen wir durch eine Klassifizierung der Paradigmen veranschaulichen. Außerdem soll das Kapitel einen ersten Überblick über objektorientierte Programmiersprachkonzepte sowie die später im Detail behandelten Themen geben und nebenbei einige häufig verwendete Begriffe einführen.

In Abschnitt 1.1 werden die wichtigsten Konzepte objektorientierter Programmiersprachen angesprochen. Viele dieser Konzepte werden in den folgenden Kapiteln genauer behandelt.

In Abschnitt 1.2 beschäftigen wir uns damit, welche Ziele durch die Programmierung im Allgemeinen erreicht werden sollen und was gute Programmierung von schlechter unterscheidet.

In Abschnitt 1.3 werden wir untersuchen, wie man gute objektorientierte Programme erkennt bzw. schreibt und welche Schwierigkeiten dabei zu überwinden sind.

Abschnitt 1.4 gibt eine Klassifizierung von Programmiersprachen anhand ihrer üblichen Verwendungen. Diese Klassifizierung soll Zusammenhänge mit anderen Paradigmen aufzeigen und helfen, den Begriff der objektorientierten Programmierung abzugrenzen.

1.1 Konzepte objektorientierter Programmierung

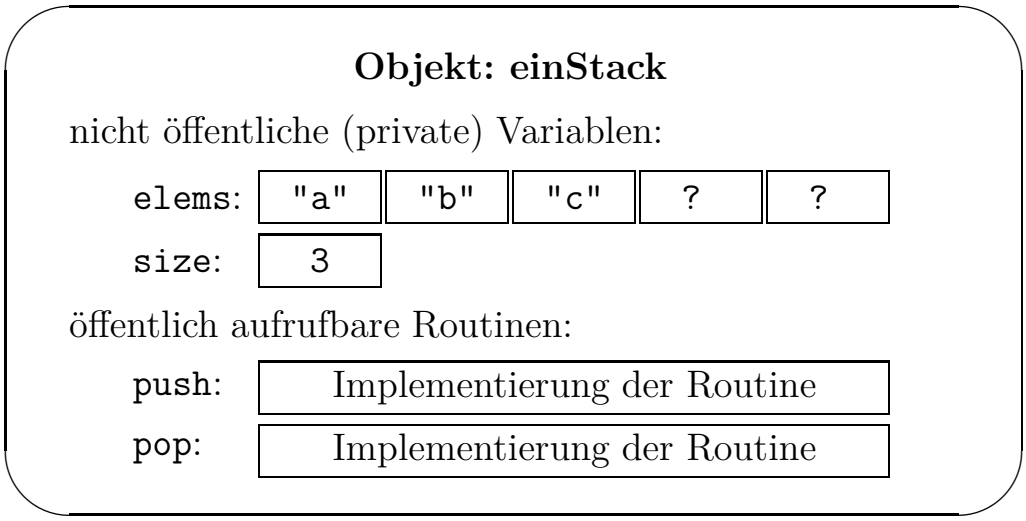
Wir wollen zunächst einige Grundkonzepte betrachten. Die objektorientierte Programmierung will vor allem Softwareentwicklungsprozesse, die auf inkrementelle Verfeinerung aufbauen, unterstützen. Gerade bei diesen Entwicklungsprozessen spielt die leichte Wartbarkeit der Programme eine große Rolle. Im Wesentlichen will die objektorientierte Programmierung auf die einfache Änderbarkeit von Programmen achten, und objektorientierte Programmiersprachen geben EntwicklerInnen Werkzeuge in die Hand, die sie zum Schreiben leicht wartbarer Software brauchen.

1.1.1 Objekte

Das wichtigste Konzept der objektorientierten Programmierung ist, wie der Name schon sagt, das des *Objekts*. Ein Objekt ist eine grundlegende Einheit in der Ausführung eines Programms. Zur Laufzeit besteht die Software aus einer Menge von Objekten, die einander teilweise kennen und untereinander *Nachrichten* (messages) austauschen.

Man kann ein Objekt am ehesten als *Kapsel* verstehen, die zusammengehörende *Variablen* und *Routinen* (ausführbare Einheiten wie z.B. Funktionen, Prozeduren und Methoden) enthält. Gemeinsam beschreiben die Variablen und Routinen eine Einheit in der Software. Von außen soll man auf das Objekt nur zugreifen, indem man ihm eine Nachricht schickt, das heißt, eine nach außen sichtbare Routine des Objekts aufruft.

Die folgende Abbildung veranschaulicht ein Objekt:



Dieses Objekt mit der Funktionalität eines Stacks fügt zwei Variablen und zwei Routinen zu einer Einheit zusammen und grenzt die Einheit so weit

wie möglich vom Rest des Systems ab. Die beiden öffentlichen Routinen sind von überall aufrufbar. Auf die privaten Variablen kann nur durch die beiden Routinen innerhalb des Objekts zugegriffen werden. Eine Variable enthält ein Array mit dem Inhalt des Stacks, eine andere die aktuelle Anzahl der Elemente am Stack. Das Array kann höchstens fünf Stackelemente halten. Zurzeit sind drei Einträge vorhanden.

Das Zusammenfügen von Daten und Routinen zu einer Einheit nennt man *Kapselung* (encapsulation). Daten und Routinen in einem Objekt sind untrennbar miteinander verbunden: Die Routinen benötigen die Daten zur Erfüllung ihrer Aufgaben, und die genaue Bedeutung der Daten ist oft nur den Routinen des Objekts bekannt. Routinen und Daten stehen zueinander in einer engen logischen Beziehung. In Abschnitt 1.2 werden wir sehen, dass eine gut durchdachte Kapselung ein wichtiges Qualitätsmerkmal ist. In Abschnitt 1.3 werden wir Faustregeln zur Unterstützung der Suche nach geeigneten Kapselungen kennen lernen. In Abschnitt 1.4 werden wir feststellen, dass die Kapselung von Daten und Routinen zu Objekten ein entscheidendes Kriterium zur Abgrenzung der objektorientierten Programmierung von anderen Programmierparadigmen ist.

Jedes Objekt besitzt folgende Eigenschaften[24]:

Identität (identity): Seine Identität kennzeichnet ein Objekt eindeutig. Sie ist unveränderlich. Über seine Identität kann man das Objekt ansprechen, ihm also eine Nachricht schicken. Vereinfacht kann man sich die Identität als die Adresse des Objekts im Speicher vorstellen. Dies ist aber nur eine Vereinfachung, da die Identität erhalten bleibt, wenn sich die Adresse ändert – zum Beispiel beim Verschieben des Objekts bei der garbage collection oder beim Auslagern in eine Datenbank. Jedenfalls gilt: Gleichzeitig durch zwei Namen bezeichnete Objekte sind *identisch* (identical) wenn sie am selben Speicherplatz liegen, es sich also um nur ein Objekt mit zwei Namen handelt.

Zustand (state): Der Zustand setzt sich aus den Werten der Variablen im Objekt zusammen. Er ist in der Regel änderbar. In obigem Beispiel ändert sich der Zustand durch Zuweisungen neuer Werte an die Variablen `elems` und `size`. Zwei Objekte sind *gleich* (equal) wenn sie denselben Zustand und dasselbe Verhalten haben. Objekte können auch gleich sein, wenn sie nicht identisch sind; dann sind sie *Kopien* voneinander. Zustände gleicher Objekte können sich unabhängig voneinander ändern; die Gleichheit geht dadurch verloren. Identität kann durch Zustandsänderungen nicht verloren gehen.

Verhalten (behavior): Das Verhalten eines Objekts beschreibt, wie sich das Objekt beim Empfang einer Nachricht verhält, das heißt, was das Objekt beim Aufruf einer entsprechenden Routine macht. Routinen, die beim Empfang von Nachrichten ausgeführt werden, nennt man häufig *Methoden* (methods). Das Verhalten ist von der Nachricht – also dem Methodennamen zusammen mit den *aktuellen Parametern*, auch *Argumente* der Nachricht genannt –, der entsprechenden aufgerufenen Methode und dem Zustand des Objekts abhängig. In obigem Beispiel wird die Methode `push` beim Empfang der Nachricht `push("d")` das Argument `"d"` in den Stack einfügen (falls es noch einen freien Platz gibt), und `pop` wird beim Empfang von `pop()` ein Element entfernen (falls eines vorhanden ist) und an den Absender der Nachricht zurückgeben.

Unter der *Implementierung* einer Methode verstehen wir den Programmcode, der festlegt, was genau beim Aufruf der Methode zu tun ist. Die Implementierungen aller Methoden eines Objekts und die Deklarationen der Variablen des Objekts bilden zusammen die Implementierung des Objekts. Die Implementierung beschreibt das Verhalten des Objekts bis ins kleinste Detail. Für die Programmausführung ist diese genaue Beschreibung ganz essentiell; sonst wüsste der Computer nicht, was er tun soll. Aber für die Wartung ist es günstiger, wenn das Verhalten eines Objekts nicht jedes Detail der Implementierung widerspiegelt. Wir fordern (neben obigen drei Eigenschaften, die zur Definition des Begriffs Objekt unbedingt notwendig sind) eine weitere Eigenschaft, die es ermöglicht, den Detailiertheitsgrad des Verhaltens nach Bedarf zu steuern:

Schnittstelle (interface): Eine Schnittstelle eines Objekts beschreibt das Verhalten des Objekts in einem Detailiertheitsgrad, der für Zugriffe von außen notwendig ist. Ein Objekt kann mehrere Schnittstellen haben, die das Objekt aus den Sichtweisen unterschiedlicher Verwendungen beschreiben. Oft enthalten Schnittstellen nur die Köpfe der überall aufrufbaren Routinen ohne weitere Beschreibung des Verhaltens. Manchmal enthalten sie auch Konstanten. Wie wir in Kapitel 2 sehen werden, kann man das Verhalten in Schnittstellen (zumindest verbal) beliebig genau beschreiben. Ein Objekt *implementiert* seine Schnittstellen; das heißt, die Implementierung legt das in den Schnittstellen unvollständig beschriebene Verhalten im Detail fest. Jede Schnittstelle kann das Verhalten beliebig vieler Objekte beschreiben. Schnittstellen entsprechen den Typen des Objekts.

Häufig verwendet man ein Objekt als *black box* oder *grey box*; das heißt, der Inhalt des Objekts ist von außen zum Großteil nicht sichtbar. Nur das, was in den Schnittstellen beschrieben ist, ist von außen sichtbar. Schnittstellen dienen dazu, den Inhalt des Objekts von dessen verschiedenen Außenansichten klar zu trennen. ProgrammiererInnen, die ein Objekt verwenden wollen, brauchen nur eine Schnittstelle des Objekts kennen, nicht aber dessen Inhalt. Man spricht daher von *data hiding*, dem Verstecken von Daten und Implementierungen. Kapselung zusammen mit data hiding heißt *Datenabstraktion*, da die Daten in einem Objekt nicht mehr direkt sichtbar und manipulierbar, sondern abstrakt sind. Im Beispiel sieht man die Daten des Objekts nicht als Array von Elementen zusammen mit der Anzahl der gültigen Einträge im Array, sondern als abstrakten Stack, der über zwei Methoden zugreifbar und manipulierbar ist. Diese Abstraktion bleibt unverändert, wenn wir das Array gegen eine andere Datenstruktur, sagen wir eine Liste, austauschen. Datenabstraktionen helfen bei der Wartung: Details von Objekten sind änderbar, ohne deren Außenansichten und damit deren Verwendungen zu beeinflussen.

1.1.2 Klassen

Viele objektorientierte Sprachen beinhalten ein Klassenkonzept: Jedes Objekt gehört zu genau einer Klasse, die die Struktur des Objekts – dessen Implementierung – im Detail beschreibt. Außerdem beschreibt die Klasse *Konstruktoren* (constructors), das sind Routinen zur Erzeugung und Initialisierung neuer Objekte. Alle Objekte, die zur Klasse gehören, wurden durch Konstruktoren dieser Klasse erzeugt. Man nennt diese Objekte *Instanzen* der Klasse. Genauer gesagt sind die Objekte Instanzen der durch die Klasse beschriebenen Schnittstellen bzw. Typen. Die Klasse selbst ist die spezifischste aller dieser Schnittstellen, die das Verhalten am genauesten beschreibt. (Anmerkung: Man sagt manchmal, ein Objekt gehöre zu mehreren Klassen, der spezifischsten Klasse und deren Oberklassen; wir verstehen im Skriptum unter der Klasse eines Objekts immer dessen spezifischste Klasse beziehungsweise Schnittstelle und sprechen von der Schnittstelle eines Objekts wenn wir eine beliebige Schnittstelle meinen.) Am Ende der Laufzeit einer Instanz wird für diese der Destruktor der Klasse aufgerufen.

Alle Instanzen einer Klasse haben dieselben Implementierungen und dieselben Schnittstellen. Aber unterschiedliche Instanzen haben immer unterschiedliche Identitäten und unterschiedliche Variablen – genauer: In-

stanzvariablen – obwohl diese Variablen gleiche Namen und Typen tragen. Auch die Zustände können sich unterscheiden.

In einer objektorientierten Programmiersprache mit Klassen schreiben ProgrammiererInnen hauptsächlich Klassen. Objekte werden nur zur Laufzeit durch Verwendung von Konstruktoren erzeugt. Nach dem Ende der Laufzeit des Objektes, können im Destruktor die Aktionen des Konstruktors rückgängig gemacht werden. Oft gibt es in diesen Sprachen gar keine Möglichkeit, Objekte direkt auszuprogrammieren.

Ein kleines Beispiel in C++ soll demonstrieren, wie Klassen aussehen:

Listing 1.1: stack1.h

```
#ifndef CSTACK_H
#define CSTACK_H

#include <string>

class CStack
{
private:
    std::string* m_elems;
    int m_size;
    int m_maxsize;
public:
    CStack (int sz) : m_size(0), m_maxsize(sz)
    {
        m_elems = new std::string[sz];
    }
    virtual ~CStack ()
    {
        delete [] m_elems;
    }
    virtual void push (std::string const& elem);
    virtual std::string pop();
    bool empty() const
    {
        return m_size == 0;
    }
    bool full() const
    {
        return m_size == m_maxsize;
    }
};

#endif
```

Folgende Beispielerklärung ist für Leser gedacht, die noch nicht genug Erfahrung mit C++ gesammelt haben. Erfahrene ProgrammiererInnen in C++ können solche speziell gekennzeichneten Textstellen überspringen.

(Anmerkungen zu C++)

Jede Instanz der Klasse `CStack` enthält die Variablen `m_elems` vom Typ Zeiger

auf `std::string` sowie `m_maxsize` und `m_size` vom Typ `int` (ganze Zahl). Alle Variablen sind in einer `private` Sektion definiert. Diese können nur von Instanzen von `CStack` zugegriffen werden. Jede Instanz unterstützt die Methoden `push`, `pop`, `empty` und `full`. Diese Methoden sind `public`, also überall sichtbar, wo eine Instanz von der Klasse `CStack` bekannt ist. Der Ergebnistyp `void` bedeutet, dass `push` kein Ergebnis zurück gibt. Der formale Parameter `elem` von `push` ist vom Typ eine konstante Referenz auf `std::string`. Durch `const` wird verhindert, dass, obwohl eine Referenz übergeben wird, `elem` unabsichtlich verändert wird. Die Methode `pop` liefert ein Ergebnis vom Typ `std::string`, hat aber keine formalen Parameter – ausgedrückt durch ein leeres Klammerpaar oder alternativ und gleichbedeutend mit `(void)`. Daneben gibt es einen Konstruktor. Syntaktisch sieht ein Konstruktor wie eine Methode aus, abgesehen davon, dass der Name immer gleich dem Namen der Klasse ist und kein Ergebnistyp angegeben wird. Der Konstruktor im Beispiel ist `public`, also überall sichtbar. Nach dem Doppelpunkt folgt eine Initialisierungsliste, die vor dem Codeblock abgearbeitet wird. Es ist auch ein Destruktor mit dem Namen `~CStack` definiert, welcher sich um die Freigabe der Ressourcen des Objektes kümmert.

In dieser Klasse wird die für C++ wichtige Programmiertechnik **RAII** (Ressourcenbelegung ist Initialisierung) verwendet, indem im Konstruktor die Ressource (das Array) alloziert und im Destruktor wieder freigegeben wird. Dadurch kann nicht auf Freigabe von Ressourcen (Speicher, aber auch Zugriff auf Dateien, Prozesse, Netzwerk usw.) vergessen werden! Diese Programmiertechnik führt aber auch zu ausnahmefestem Code (Siehe Kapitel 3.5.1). Diese Technik sollte immer wenn Ressourcen belegt werden, verwendet werden. Sie findet auch in der STL (Standard Template Library) intensiv Anwendung.

Neue Objekte können durch den Operator `new` erzeugt werden. Sie werden automatisch durch den Aufruf eines Konstruktors initialisiert. Danach muss der Operator `delete` verwendet werden um das Objekt wieder freizugeben. Für Arrays hingegen gibt es den Operator `new[]` für die Allokation. Diese müssen allerdings immer durch ein entsprechendes `delete[]`, ohne Größenangabe, freigegeben werden.

> **Tipp:** Für jedes `new` oder `new[]` in einem C++ Programm muss ein entsprechendes `delete` oder `delete[]` exisiteren. RAII ist die einfachste Möglichkeit dieses Ziel zu erreichen.

Zum Beispiel erzeugt `CStack s(5);` eine neue Instanz von `CStack` auf dem Stack mit neuen Variablen `m_elems`, `m_maxsize` und `m_size` und ruft den Konstruktor in `CStack` auf, wobei der formale Parameter `sz` an `5` gebunden ist. Bei der Ausführung des Konstruktors wird durch `new std::string[sz]` eine neue Instanz eines Arrays von Zeichenketten auf dem Heap erzeugt. Im Array finden 5 Zeichenketten Platz. Dieses Array wird an die Variable `m_elems` zugewiesen. Durch die Initialisierungsliste wurde die Variable `m_size` auf 0 und `m_maxsize` auf `sz` bereits zu Beginn initialisiert.

Statt Arrays können auch dynamische Datenstrukturen der STL, wie etwa `std::deque`, verwendet werden. Diese bieten auch einen Zugriff auf Elemente mit dem Operator `[]` an, kümmern sich aber im Gegensatz zu den Arrays automatisch um die Speicherverwaltung. Sie werden automatisch und dynamisch vergrößert und verkleinert. Der

kleinste Index ist immer 0. Ungültige Zugriffe dürfen nicht getätigt werden. Um in diesem Fall undefiniertes Verhalten zu vermeiden, kann stattdessen `at(size_type)` verwendet werden.

> **Tipp:** Verwende bevorzugt Container der Standardlibrary.

Die in diesem Beispiel verwendete `#ifndef`, `#define` und `#endif` Klausel sollte sich in jeder C/C++ Header Datei wiederfinden. Sie verhindert mehrmaliges indirektes Inkludieren, indem ein bestimmter String überprüft, und wenn dieser noch nicht gesetzt, definiert wird. Dieser String kann nahezu beliebig sein. Er muss aber im gesamten Programm eindeutig sein. In der Praxis verwendet man deshalb meistens (Teile von) den Dateinamen.

Um einen logischen Zusammenhang von Funktionen und Klassen auszudrücken, gibt es *Namensbereiche*. Innerhalb diesen werden Namen, wie beispielsweise Funktionen, im eigenen Namensbereich gesucht. Um von außerhalb zuzugreifen, muß allerdings angegeben werden, dass dieser Namensbereich gemeint ist. Um Teile der STL zu verwenden, müssen entsprechende Definitionen mit `#include` in den eigenen Code eingebunden werden. Diese Klassen sind in dem Namensbereich `std`, deshalb muss bei deren Verwendung in Header-Files `std::` geschrieben werden.

Die in diesem Beispiel verwendeten Arrays wurden von C übernommen und haben weder die Schutzmechanismen noch die Möglichkeit, vergrößert oder verkleinert zu werden. Bei der Objekterzeugung mit `new[]` enthalten die eckigen Klammern die Anzahl der Array-Einträge, bei einem Zugriff den Index. Die Freigabe muss mit `delete[]` erfolgen - eine Angabe der Größe ist hier aber nicht notwendig. Der unterste Index ist auch hier immer 0. Am Anfang sind die Array-Einträge nicht initialisiert.

Zeiger und ihre Arithmetik wurden in C++ mit ihrer Mächtigkeit und Gefährlichkeit vollständig von C übernommen. Sie sollten entweder auf ein Objekt oder auf `0` zeigen. 0 hat eine spezielle Bedeutung und sagt aus, dass der Zeiger auf nichts zeigt. Das Makro `NULL` von C sollte hingegen nicht mehr verwendet werden. Es kann aber auch gänzlich auf Zeiger verzichtet werden und stattdessen mit *Referenzen* gearbeitet werden. Diese können im Gegensatz zu Zeigern weder mit 0, noch mit sonst irgendeiner ungültigen Adresse initialisiert werden. Wichtig ist, dass man keine Referenzen (oder Zeiger) auf lokale Objekte (Objekte am Stack) zurückgibt, da diese beim Verlassen der Methode freigegeben werden.

> **Tipp:** Verwende Referenzen statt Zeiger. Keine Referenzen von lokalen Objekten zurückgeben!

Wie wir später sehen werden, ist das Schlüsselwort `virtual` sehr wichtig. Es ermöglicht die so deklarierten Funktionen so zu überschreiben, dass zur Laufzeit nach der richtigen Methode gesucht wird. Das nennt man *dynamisches Binden* und wird im Kapitel 1.1.5 detailliert erklärt.

Zu beachten ist, dass, wann immer irgendeine Methode `virtual` deklariert ist, muss auch der Destruktor `virtual` deklariert sein [Str00], da sonst nicht von dem richtigen

Objekt zur Laufzeit die Ressourcen freigegeben werden! Da der Defaultdestruktor (dieser wird verwendet wenn keiner angegeben wird), **nicht** `virtual` ist, muss dann ein eigener Destruktor definiert werden.

Die Implementierung der zwei in `stack1.h` deklarierten Methoden erfolgt in einer eigenen Datei:

Listing 1.2: stack1.cpp

```
#include "stack1.h"

#include <stdexcept>

void CStack::push (std::string const& elem)
{
    if (full()) throw std::out_of_range
        ("CStack::push(): Cant push on full stack");
    m_elems[m_size] = elem;
    m_size = m_size + 1;
}

std::string CStack::pop()
{
    if (empty()) throw std::out_of_range
        ("CStack::pop(): Cant pop on empty stack");
    m_size = m_size - 1;
    return m_elems[m_size];
}
```

(Anmerkungen zu C++)

Signatur und Implementierung, auch Deklaration und Definition genannt, werden in C++ grundsätzlich getrennt. *Deklaration* ist die Ankündigung, dass später eine *Definition* folgen wird. Es ist mindestens eine Deklaration notwendig, damit ein Name verwendet werden kann. Die Deklaration einer Funktion ist deren Signatur, die Definition ist die Signatur mit der Implementierung. Bei einer Klasse ist die Deklaration `class CStack;`, die Definition der Code von `stack1.h`. Bei einer Methode ist die Deklaration die Signatur die in einer Klassendefinition steht. Die Definition ist der Code von `stack1.cpp`. Wird, wie in diesem Beispiel der Konstruktor und Destruktor und die Methoden `empty` und `full`, direkt in der Klassendefinition angegeben, so ist der Code implizit `inline`. Das bedeutet, dass der Compiler versucht, diese Codesegmente an den verwendeten Stellen einzufügen, statt eine Funktion aufzurufen. Längere Teile sollten deshalb und auch der Übersichtlichkeit wegen, getrennt definiert werden, wie in dem Beispiel bei `push` und `pop` geschehen.

Ausnahmen ermöglichen eine zentrale Fehlerbehandlung, die sehr wünschenswert sein kann. Der Hauptvorteil der Ausnahmen ist, dass im Gegensatz zu einem Fehlerstatus im Rückgabewert, ein ignorierter Fehler das Programm sauber beendet und nicht undefiniert weiterlaufen lässt. Das ist aber nicht immer erwünscht. Wie Ausnahmen genau eingesetzt werden, wird im Kapitel 3.5 detailliert beschrieben.

> **Tipp:** Verwende eine Ausnahme, wenn ein Fehler nicht einfach ignoriert werden kann.

Ein Aufruf von `push` stellt mit Hilfe von `full()` fest, ob es im Array noch einen freien Eintrag gibt. Da `full()` implizit inline ist, wird der Compiler die Bedingung an dieser Stelle direkt einfügen. In diesem Fall wird der Parameter als neues Element in das Array eingetragen und `m_size` erhöht: Sollte es bereits voll besetzt sein wird die Ausnahme `out_of_range` geworfen.

Ein Aufruf von `pop` verringert `m_size` um 1 und liefert durch eine `return`-Anweisung eine Kopie des Array-Eintrag an der Position `m_size` zurück. Es wird hier keine Referenz verwendet, da ein anschließendes `push` den Array-Eintrag überschreibt und damit die Referenz auf einen falschen - aber gültigen - Eintrag zeigen würde. Bei einem leerem Stack, abgefragt über die Methode `empty()`, wird wiederum die Ausnahme `out_of_range` geworfen.

Da jede Instanz von `CStack` ihre eigenen Variablen hat, stellt sich die Frage, zu welcher Instanz von `CStack` die Variablen gehören, auf die die Methoden zugreifen. In der Klasse selbst steht nirgends, welches Objekt das ist. Die Instanz von `CStack`, die dabei verwendet wird, ist im Aufruf der Methode eindeutig festgelegt, wie wir an folgendem Beispiel sehen:

Listing 1.3: stacktest1.cpp

```
#include "stack1.h"

#include <iostream>
#include <stdexcept>

using namespace std;

int main(int argc, char **argv) try
{
    CStack s (5);
    int i;

    for (i=1; i<argc; i++) s.push (argv[i]);
    for (;i>1;i--) cout << s.pop() << endl;
} catch (out_of_range const& oor) {
    cerr << argv[0] << ": " << oor.what() << endl;
    return 1;
}
```

(Anmerkungen zu C++)

In dem Code von `stacktest1.cpp` wird nur die Funktion `main` definiert, welche die Klasse `CStack` verwendet. Die Funktion hat die Anzahl von Argumenten und ein Array von Zeichenketten als Parameter. Beim Programmstart enthält dieses Array die Argumente (command line arguments), die im Programmaufruf angegeben werden.

Damit hat `argc` den Wert 4 und `argv` ist ein Array von vier Zeichenketten: dem Programmnamen, `"a"`, `"b"` und `"c"`. Es wird wie in `stack1.cpp` wieder die Definition von der Klasse `CStack` durch `#include "stack1.h"` in die Datei inkludiert. Zu Beachten ist, dass nicht die Implementationen von `push` und `pop` inkludiert, sondern zusammen gelinkt werden.

`<iostream>` stellt ein auf *Streams* basierendes Ein/Ausgabekonzept in C++ zur Verfügung. Da `cout`, `endl` und `out_of_range` im *Namensbereich* `std` definiert sind und jedesmal `std::` davor zu schreiben mühsam wird, kann mit `using namespace std;` für diese Datei der Namensbereich direkt verwendet werden. Allerdings darf diese Methode nicht in Header-Dateien angewendet werden, da diese Dateien zum Inkludieren gedacht sind und in anderen Dateien zu Namenskonflikten führen können.

Tipp: Verwende keine `using` Deklarativen in Headerdateien.

Die Funktion `main` hat zwei lokale Variablen. Die Variable `s` wird mit einer neuen Instanz von `CStack` initialisiert und `i` mit dem Wert 1, da wir den Programmnamen nicht in den Stack aufnehmen wollen. Die erste Schleife wird für die restlichen Zeichenketten in `argv` einmal durchlaufen. In jedem Schleifendurchlauf wird die Nachricht `push(argv[i])` an das Objekt `s` gesendet; es wird also `push` in `s` mit der Zeichenkette `argv[i]` als Argument aufgerufen. Bei der Ausführung von `push` ist bekannt, dass die Variablen des Objekts `s` zu verwenden sind. Die zweite Schleife wird gleich oft durchlaufen wie die erste. Die Anweisung `cout << s.pop() << endl` gibt das oberste Element vom Stack auf die Standardausgabe – normalerweise das Terminal – aus und entfernt dieses Element vom Stack. Die globale Variable `cout` enthält ein Objekt, den output stream für die Standardausgabe. In diesem Objekt wird der Operator `<<` aufgerufen, die eine Zeile mit dem Argument in den output stream schreibt. Als Argument wird dem Operator das Ergebnis eines Aufrufs von `pop` in `s` übergeben.

1.1.3 Werkzeuge für C++

Um effektiv C++ programmieren zu können, benötigt man mehr als einen Editor und Compiler. Viele der hier beschriebenen Werkzeuge werden auch in IDEs (z.b. Code::Blocks, kdevelop) integriert, sie können aber auch wie hier beschrieben direkt auf der Konsole aufgerufen werden. Als Shell-Prompt wird `>` verwendet. Zudem werden in diesem Abschnitt Hintergründe erklärt, die jeder C++ Programmierer unbedingt wissen sollte.

Als durchgängiges Beispiel wird in diesem Abschnitt der im vorigen Abschnitt 1.1.2 eingeführte Code für `CStack` verwendet.

Make

Abgesehen davon, dass es sehr umständlich wäre jedesmal den Compiler manuell für das Übersetzen von Code aufzurufen, hat man das Problem,

dass es für einen Menschen schwierig ist festzustellen, wo sich etwas geändert hat und welche Teile deswegen neu compiliert werden müssen.

Das Programm `make` bestimmt, ob die Quelldateien neuer sind als die Zieldateien und stoßt dann automatisch einen standardmäßigen oder vom Benutzer bestimmten Übersetzungsschritt an. Wegen vielen vordefinierten Standardregel ist folgendes bereits ein vollständiges Makefile:

Listing 1.4: Makefile

```
all:program
```

Existiert eine Datei `program.cpp`, so wird dieses beim Aufruf von `make` übersetzt. Für das `CStack` Beispiel, mit den richtigen Compilerflags und Targets für die Übung, wird das `Makefile` allerdings ein wenig länger:

Listing 1.5: Makefile

```
LD=g++

CXXFLAGS=-g -ansi -W -Wall -pedantic-errors

PROG=stack1
OBJ=stack1.o stacktest1.o

.PHONY: all
all:$(PROG)

$(PROG):$(OBJ)
    $(LD) -o $(PROG) $(OBJ)

stacktest1.o:stacktest1.cpp stack1.h
stack1.o:stack1.cpp stack1.h

.PHONY:run
run: $(PROG)
    ./$(PROG) a b c

.PHONY:clean
clean:
    rm -f $(OBJ) $(PROG)
```

Wir übersetzen das Programm nun mit der Eingabe von:

```
> make
```

Es werden folgende Übersetzungsschritte ausgegeben:

```
g++ -g -ansi -W -Wall -pedantic-errors\
    -c -o stack1.o stack1.cpp
g++ -g -ansi -W -Wall -pedantic-errors\
    -c -o stacktest1.o stacktest1.cpp
g++ -o stack1 stack1.o stacktest1.o
```

Nachdem `CStack` und das Hauptprogramm mit Hilfe des Makefiles übersetzt wurden, können wir das Programm zum Beispiel so aufrufen:

```
> ./stack1 a b c
```

Es kann auch stattdessen `make run`, welches den gleichen Aufruf durchführt, verwendet werden.

Am Bildschirm werden dann folgende drei Zeilen ausgegeben:

```
c
b
a
```

Werden hingegen mehr als fünf Elemente übergeben, so terminiert das Programm mit einer Fehlermeldung und dem Rückgabewert 1. Ein Wert ungleich 0 signalisiert einen Fehler. Der Rückgabewert ist in einer Shell in der Variable `$?` verfügbar. Man beachte, dass auch der Programmname ausgegeben wird, um die Programmierrichtlinien der Übung zu erfüllen:

```
> ./stack1 a b c d e f g
./stack1: CStack::push(): Cant push on full stack
> echo $?
1
```

Valgrind

Um bei einem laufenden Programm zu überprüfen, ob wirklich alle allozierten Ressourcen auch wieder freigegeben werden, wird `valgrind memcheck` verwendet. Ein Aufruf sieht so aus:

```
> valgrind ./stack1 --leak-check=full a b c d e f
```

In der Ausgabe sollte idealerweise „no leaks are possible“ erscheinen. Das erleichtert sehr, und gibt gute Gewissheit, dass bei der Heapverwaltung – für diesen Aufruf – alles geklappt hat. Es ist aber unbedingt anzumerken, dass Fehler auf dem Stack nicht überprüft werden. Außerdem wird keine Aussage über *alle mögliche* Ausführungen des Programmes gemacht, es sollte also zumindest mit unterschiedlicher Parameteranzahl getestet werden.

Fehler werden allerdings sehr effektiv gefunden, entfernt man z.B. das

```
delete [] m_elems;
```

in stack1.h, so erscheint folgende Ausgabe (wird mit `-g` compiliert, so werden auch die Zeilennummern angezeigt):

```
==5982== 178 (48 direct, 130 indirect) bytes in 1 blocks are definitely lost..
==5982==    at 0x4C2188C: operator new[](unsigned long) (..)
==5982==    by 0x401BB7: CStack::CStack(int) (stack1.h:16)
==5982==    by 0x4018D8: main (stacktest1.cpp:8)
```

Diese Ausgabe bedeutet, dass im `stack1.h`, Zeile 16, ein Array mit `new[]` alloziert, aber bis zum Programmende nicht mehr freigegeben wurde. Auch wenn statt `delete[]` beispielsweise `delete` oder `free` verwendet wird, was beides nicht erlaubt ist, so findet `valgrind` diesen Fehler zuverlässig.

Geordi

Bei dynamischen Sprachen werden Einzeiler, um beispielsweise schnell einen Sachverhalt testen, sehr gerne verwendet. In C++ ist es auch möglich sehr kompakten Code zu schreiben. Es wurde auch ein ähnliches Environment wie für dynamische Sprachen entwickelt: Geordi, ein C++ eval-bot. Er kann lokal installiert oder in IRC Channels wie `#geordi` auf `freenode` verwendet werden.

Möchte man beispielsweise testen, wie sich Gleitkommazahlen verhalten, so ist das einfach möglich:

```
geordi << (3 * 0.1 == 0.3)
Output: false
```

geordi verwendet im Hintergrund `g++` mit gut gewählten Compilerflags. Da in nur einer Zeile programmiert wird, ist bereits alles vom C++ Standard (mit C++0x Erweiterungen) und `boost` inkludiert und deren Namensbereiche in Verwendung. Es sei hier aber darauf hingewiesen, dass `geordi` – wie jeder Compiler – kein Ersatz ist, um in Detailfragen den Standard zu konsultieren. Es ist grundsätzlich empfehlenswert nach dem Standard zu programmieren und für fehlerhafte Compiler entsprechende Workarounds und Fixes zu verwenden. Das ist in dieser Übung aber sicher nicht notwendig.

Debugging

Sollte das Programm nicht das erwartete Verhalten an den Tag legen, so muss es debuggt werden. Eine sehr einfache, aber meistens ausreichende Möglichkeit ist es mittels `cout` die Variablen auszugeben von denen man glaubt sie haben einen bestimmten Wert, der aber in Diskrepanz zu dem Verhalten steht. Unter Umständen kann die Ausgabe im Code gelassen werden, wenn sie später noch von Nutzen ist. Am besten gibt man sie dann in einen `#ifdef` Block um sie mit einem Compilerflag nach Belieben

an- und ausschalten zu können. Compilerflags übergibt man mit `-DNAME`. Das entspricht `#define NAME` am Beginn des Codes.

Für komplexe Probleme – wobei hier oft zu überlegen ist, ob man das Programm nicht hätte einfacher schreiben können – ist dann ein Debugger wie beispielsweise `gdb` – unverzichtbar. Damit kann man nicht nur Variablen inspizieren, sondern sie auch ändern, den *Stacktrace* ausgeben usw. Es wird hier nicht darauf eingegangen wie das gemacht wird, es soll nur ein sehr häufiger Anwendungsfall gezeigt werden. Verändern wir unser Beispiel so, dass

```
argv=0;
```

am Beginn der `main` Funktion in `stacktest1.cpp` steht. Nun führen wir das Programm in gdb aus:

```
> gdb ./stack
(gdb) run a b c
Program received signal SIGSEGV, Segmentation fault.
(gdb) bt full
...
```

Wir sehen dass Nullzeiger in C++ nicht dereferenziert werden dürfen. Das Programm stürzt ab, wenn es doch gemacht wird. Deshalb sind auch nicht Zeiger, sondern Referenzen, die bevorzugte Variante um in C++ zu einem Objekt zu zeigen. `bt full` kann aber auf jeden Fall anzeigen an welcher Stelle, mit welchen Variablen es zu dem Absturz kam.

Doxygen

Das Dokumentationswerkzeug doxygen bietet die Möglichkeit aus speziellen Kommentaren, Klassen, Funktionen und Methoden eine Dokumentation in einer Vielzahl von Formaten zu erzeugen. Doxygen unterstützt alles was in der Übung nach den Richtlinien dokumentiert werden muss, unter anderem auch Pre- und Postconditions (siehe Kapitel 2.2.1).

Versionsverwaltung

Es sollte kein Programm ohne Verwendung von Versionsverwaltung geschrieben werden. Zum Einen ermöglicht es zu älteren Versionen zurückzugehen, unabsichtliche Löschungen rückgängig zu machen, aber auch in verschiedenen Entwicklungszweigen entweder zu experimentieren oder die

Software zu warten. Neben den bekannten Versionsverwaltungswerkzeugen wie `svn` sind vor allem dezentrale Versionsverwaltungswerkzeuge wie `git` sehr beliebt, da bei diesen das Aufsetzen eines Servers entfällt und ein wesentlich schwächer gekoppeltes und flexibleres Entwicklungsmodell ermöglicht wird. Anstatt nur von einer zentralen Instanz können von jedem Repository Änderungen bezogen werden. Um auf den Laborrechnern git zu verwenden, ist folgendes durchzuführen:

```
> mkdir bsp1
> cd bsp1
> git init-db
> # Dateien bearbeiten
> git add .
> git commit -a
```

Die letzen beide Befehle werden nun immer dann durchgeführt, wenn neue Dateien hinzugekommen sind oder Änderungen aufgezeichnet werden sollen.

(für Interessierte)

Zum Abschluß noch ein paar Grundlagen, wie C++ Programme übersetzt werden: Nachdem der Programmierer in den `.h` Dateien die Klassendefinitionen und Funktionsdeklarationen, in den `.cpp` Dateien die Methoden- und Funktionsdefinitionen und im `Makefile` die Übersetzungschritte angegeben hat, wird folgendes gemacht: Für jede `.cpp` Datei wird eine `.o` Datei compiliert. In diesen Objekt-Dateien sind alle *Symbole*, das sind Implementationen von Funktionen, Methoden, globale Variablen usw. vorhanden. Diese können mit dem Tool `nm` aufgelistet werden:

```
> nm stack1.o
000000000000016e T _ZN6CStack3popEv
0000000000000000 T _ZN6CStack4pushERKSs
```

Wir sehen, dass vor (und nach) den Methodennamen `pop` und `push` zusätzliche Informationen über Namensbereiche, Klassen, aber auch Parameter, da C++ Überladen unterstützt, hinzugekommen sind. Diesen Prozess nennt man *Mangling*. Mit `c++filt` kann wieder eine lesbare Information erzeugt werden:

```
> nm stack1.o | c++filt
000000000000016e T CStack::pop()
0000000000000000 T CStack::push(std::basic_string<...> const&)
```

Diese Symbole in den `.o` Dateien werden schließlich mit dem Linker `ld` (wird von `g++` aufgerufen) zusammen zu einer ausführbaren Datei oder Shared Library (`.so`) gelinkt. Eine Shared Library kann von mehreren Programmen gemeinsam benutzt werden, liegt aber nur einmal im Speicher. Des weiteren können die `.so` Dateien auch dynamisch zur Laufzeit zusätzlich geladen werden. Dazu wird unter Linux `dlopen` verwendet.

1.1.4 Polymorphismus

Das Wort *polymorph* kommt aus dem Griechischen und heißt „vielgestaltig“. Im Zusammenhang mit Programmiersprachen spricht man von *Polymorphismus*, wenn eine Variable oder eine Routine gleichzeitig mehrere Typen haben kann. Ein formaler Parameter einer polymorphen Routine kann an Argumente von mehr als nur einem Typ gebunden werden. Objektorientierte Sprachen sind polymorph. Im Gegensatz dazu sind konventionelle typisierte Sprachen wie C und Pascal im Großen und Ganzen *monomorph*: Jede Variable oder Routine hat einen eindeutigen Typ.

In einer polymorphen Sprache hat eine Variable (oder ein formaler Parameter) meist gleichzeitig folgende Typen:

Deklarierter Typ: Das ist der Typ, mit dem die Variable deklariert wurde. Dieser existiert natürlich nur bei expliziter Typdeklaration.

Statischer Typ: Der statische Typ wird vom Compiler (statisch) ermittelt und kann spezifischer sein als der deklarierte Typ. In vielen Fällen ordnet der Compiler ein und derselben Variablen an verschiedenen Stellen verschiedene statische Typen zu. Solche Typen werden beispielsweise für Programmoptimierungen verwendet. Es hängt von der Qualität des Compilers ab, wie spezifisch der statische Typ ist. In Sprachdefinitionen kommen statische Typen daher nicht vor.

Dynamischer Typ: Das ist der spezifischste Typ, den der in der Variable gespeicherte Wert tatsächlich hat. Dynamische Typen sind oft spezifischer als deklarierte Typen und können sich mit jeder Zuweisung ändern. Dem Compiler sind dynamische Typen nur in dem Spezialfall bekannt, in dem dynamische und statische Typen einander stets entsprechen. Dynamische Typen werden unter anderem für die Typüberprüfung zur Laufzeit verwendet.

Man kann verschiedene Arten von Polymorphismus unterscheiden [6]:

- Polymorphismus
  - universeller Polymorphismus
    - Generizität
    - enthaltender Polymorphismus
  - Ad-hoc-Polymorphismus
    - Überladen
    - Typumwandlung

Nur beim universellen Polymorphismus haben die Typen, die zueinander in Beziehung stehen, eine gleichförmige Struktur:

Generizität (genericity): Generizität wird auch als *parametrischer Polymorphismus* bezeichnet, weil die Gleichförmigkeit durch Typparameter erreicht wird. Das heißt, Ausdrücke können Parameter enthalten, für die Typen eingesetzt werden. Zum Beispiel kann im Ausdruck `std::list<T>` der Typparameter `T` durch den Typ `std::string` ersetzt werden. Das Ergebnis der Ersetzung, `std::list<std::string>`, ist der (generierte) Typ einer Liste von Zeichenketten. Ein Ausdruck mit freien Typparametern bezeichnet die Menge aller Ausdrücke, die durch Einsetzen von Typen generiert werden können. Typparameter werden als universell über die Menge aller Typen quantifizierte Variablen betrachtet. Daher wird Generizität dem universellen Polymorphismus zugerechnet. Wir beschäftigen uns in Kapitel 3 mit Generizität.

Enthaltender Polymorphismus (inclusion polymorphism): Diese Art, auch *subtyping* genannt, spielt in der objektorientierten Programmierung eine wichtige Rolle. Angenommen, der Typ `Person` hat die *Untertypen* (subtypes) `Student` und `Angestellter`. Dann ist jedes Objekt vom Typ `Student` oder `Angestellter` auch ein Objekt vom Typ `Person`. An eine Routine mit einem formalen Parameter vom Typ `Person` kann auch ein Argument vom Typ `Student` oder `Angestellter` übergeben werden. Die Menge der Objekte vom Typ `Person` enthält alle Objekte der Typen `Student` und `Angestellter`. Die Routine akzeptiert alle Argumente vom Typ t, wobei t universell über `Person` und dessen Untertypen quantifiziert ist. Daher ist auch enthaltender Polymorphismus ein universeller Polymorphismus.

Eine Schnittstelle entspricht im Wesentlichen einem Typ. Wenn die Schnittstelle eine Methode beschreibt, dann müssen auch alle Schnittstellen, die Untertypen davon sind, dazu kompatible Methoden beschreiben. Eine Methode ist kompatibel, wenn sie überall dort verwendbar ist, wo die ursprüngliche Methode erwartet wird. Diese Einschränkung kann man zur Definition von enthaltendem Polymorphismus durch das *Ersetzbarkeitsprinzip* verwenden [25]:

Definition: Ein Typ U ist ein Untertyp eines Typs T (bzw. T ist ein Obertyp von U) wenn eine Instanz von U überall verwendbar ist, wo eine Instanz von T erwartet wird.

Der Compiler sieht nur den statischen Typ einer Variablen oder eines formalen Parameters. Der dynamische Typ steht erst während der Ausführung fest. Daher kann der Compiler auch nicht immer feststellen, welche Implementierung einer Methode des in der Variable enthaltenen Objekts ausgeführt werden muss, da ja nur eine Schnittstelle, die zu Instanzen unterschiedlicher Klassen gehören kann, bekannt ist. Die auszuführende Methode wird erst während der Programmausführung festgestellt. Dies ist als *dynamisches Binden* (dynamic binding) bekannt. *Statisches Binden* (static binding) bedeutet, dass bereits der Compiler die auszuführende Methode festlegt. Im Zusammenhang mit enthaltendem Polymorphismus ist man auf dynamisches Binden angewiesen. Enthaltenden Polymorphismus und dynamisches Binden werden wir in Kapitel 2 behandeln.

Überladen (overloading): Eine Routine heißt *ad-hoc-polymorph*, wenn sie Argumente mehrerer unterschiedlicher Typen, die in keiner Relation zueinander stehen müssen, akzeptiert und sich für jeden dieser Typen anders verhalten kann. Beim Überladen bezeichnet ein und derselbe Name verschiedene Routinen, die sich durch die deklarierten Typen ihrer formalen Parameter unterscheiden. Die deklarierten Typen der übergebenen Argumente entscheiden, welche Routine ausgeführt wird. Überladen dient häufig nur der syntaktischen Vereinfachung, da für Operationen mit ähnlicher Funktionalität nur ein gemeinsamer Name vorgesehen zu werden braucht. Zum Beispiel bezeichnet „/“ sowohl die ganzzahlige Division als auch die Division von Fließkommazahlen, obwohl diese Operationen sich im Detail sehr stark voneinander unterscheiden. Überladen ist nicht spezifisch für die objektorientierte Programmierung.

Typumwandlung (type coercion): Typumwandlung ist eine semantische Operation. Sie dient zur Umwandlung eines Wertes in ein Argument eines Typs, der von einer Routine erwartet wird. Zum Beispiel wird in C jede Instanz von `char` oder `short` bei der Argumentübergabe implizit in eine Instanz von `int` umgewandelt, wenn der Parametertyp `int` ist. Sprachen wie C++ definieren durch diffizile Regeln,

wie Typen umgewandelt werden, wenn zwischen mehreren überladenen Routinen gewählt werden kann. Auch die Typumwandlung ist nicht spezifisch für die objektorientierte Programmierung.

1.1.5 Vererbung

Die *Vererbung* (inheritance) in der objektorientierten Programmierung ermöglicht es, neue Klassen aus bereits existierenden Klassen abzuleiten. Dabei werden nur die Unterschiede zwischen der *abgeleiteten Klasse* (derived class) und der entsprechenden *Basisklasse* (base class), von der abgeleitet wird, angegeben. Die abgeleitete Klasse heißt auch *Unterklasse* (subclass), die Basisklasse *Oberklasse* (superclass). Vererbung erspart ProgrammiererInnen Schreibaufwand. Außerdem werden einige Programmänderungen vereinfacht, da sich Änderungen von Klassen auf alle davon abgeleiteten Klassen auswirken.

In populären objektorientierten Programmiersprachen können bei der Vererbung Unterklassen im Vergleich zu Oberklassen aber nicht beliebig geändert werden. Eigentlich gibt es nur zwei Änderungsmöglichkeiten:

Erweiterung: Die Unterklasse erweitert die Oberklasse um neue Variablen, Methoden und Konstruktoren.

Überschreiben: Methoden der Oberklasse werden durch neue Methoden überschrieben, die jene in der Oberklasse ersetzen. Meist gibt es eine Möglichkeit, von der Unterklasse aus auf überschriebene Routinen der Oberklasse zuzugreifen.

Diese beiden Änderungsmöglichkeiten sind beliebig kombinierbar.

Im folgenden Beispiel leiten wir eine Klasse `CounterStack` von der in Abschnitt 1.1.2 definierten Klasse `CStack` ab:

Listing 1.6: counterstack1.h

```
#ifndef COUNTERSTACK_H
#define COUNTERSTACK_H

#include "stack1.h"

class CounterStack : public CStack
{
private:
    int m_counter;
public:
    CounterStack (int sz, int c=0)
        : CStack(sz), m_counter(c)
```

```
    {}
    virtual void push (std::string const& elem);
    void count();
};

#endif
```

(Anmerkungen zu C++)

Wir beginnen wieder mit einer `#ifndef` Klausel, um das mehrmalige Inkludieren der Datei zu verhindern. Dieser Prüfmechanismus wird in diesem Skriptum bei den weiteren Beispielen aus Platzgründen weggelassen. Der nächste Schritt ist das Inkludieren der Klassendefinition von `CStack` um davon ableiten zu können.

Der Konstruktor für `CounterStack` verwendet die Initialisierungsliste, um den Konstruktor der Oberklasse `CStack` mit dem ersten Argument aufzurufen. Das zweite, optionale Argument wird zur Initialisierung von `m_counter` verwendet. Die Methode `push` wurde überschrieben. Die neue Methode erhöht `m_counter` und ruft anschließend die überschriebende Methode auf. Die Methode `pop` ist nicht überschrieben, wird also von `CStack` geerbt. `CounterStack` erweitert `CStack` um `count`. Diese Methode wandelt den Wert von `m_counter` in eine Zeichenkette um und fügt sie in den Stack ein.

Listing 1.7: counterstack1.cpp

```
#include "counterstack1.h"

#include <sstream>

void CounterStack::push (std::string const& elem)
{
    ++ this->m_counter;
    CStack::push(elem);
}

void CounterStack::count()
{
    std::stringstream s;
    s << m_counter;
    CStack::push(s.str());
}
```

(Anmerkungen zu C++)

In der Implementierungsdatei `counterstack1.cpp` die zwei ausstehenden Methoden definiert. Bei der Implementation von `count` wird der Wert von `m_counter` in eine Zeichenkette umgewandelt. Dies wird realisiert, indem ein `std::stringstream` verwendet wird. Es handelt sich hierbei um einen Stream, der in eine Zeichenkette schreibt. Um auf den geschriebenen String zugreifen zu können, wird die Methode `str()` verwendet. Um `std::stringstream` verwenden zu können, muss `<sstream>` inkludiert werden.

Der *Bereichsoperator* `::` dient zur genauen Spezifikation von Namen. Wie wir bereits gesehen haben, kann damit der Name mit einem Namensbereich, beispielsweise

`std` qualifiziert werden. Bei der Methodendefinition, wie bei `CounterStack::push`, wird der Bereichsoperator ähnlich verwendet um die Klasse zu spezifizieren. Innerhalb des Rumpfes der Implementierung wird `CStack::push` verwendet. Damit wird ausgesagt, dass an dieser Stelle ein Aufruf der Methode `push` in der Oberklasse erwünscht ist. Ohne dieser Qualifizierung würde eine endlose Rekursion entstehen.

Das aktuelle Objekt ist immer durch das Schlüsselwort `this` in jeder Methode erreichbar. Der `this`-Zeiger wird in `push` verwendet. Allerdings hätte auf die Variable `m_counter` auch einfach so zugegriffen werden können, da es hier keine Mehrdeutigkeit gibt. Der `this`-Zeiger ist aber manchmal durchaus notwendig, da beispielsweise ein Parameter einer Methode eine Variable verdeckt.

In Programmiersprachen wie C++ besteht ein enger Zusammenhang zwischen `public` Vererbung und enthaltendem Polymorphismus: Eine Instanz einer Unterklasse kann, zumindest soweit es vom Compiler überprüfbar ist, überall verwendet werden, wo ein Zeiger oder Referenz auf eine Instanz einer Oberklasse erwartet wird. Änderungsmöglichkeiten bei der Vererbung sind, wie oben beschrieben, eingeschränkt, um die Ersetzbarkeit von Instanzen der Oberklasse durch Instanzen der Unterklasse zu ermöglichen. Es besteht eine direkte Beziehung zwischen Klassen und Typen: Die Klasse eines Objekts ist gleichzeitig der spezifischste Typ bzw. die spezifischste Schnittstelle des Objekts. Dadurch entspricht eine Vererbungsbeziehung einer Untertypbeziehung. Im Beispiel ist `CounterStack` ein Untertyp von `CStack`. Eine Instanz von `CounterStack` kann überall verwendet werden, wo eine Instanz von `CStack` erwartet wird. Jede Variable vom Typ `CStack` kann auch eine Instanz vom Typ `CounterStack` enthalten.

In C++ wird sowohl statisches als auch dynamisches Binden unterstützt. Beim statischen Binden wird immer die Methode von der deklarierten Klasse aufgerufen. Statisches Binden wird verwendet, wenn eine Methode von einem Objekt auf dem Stack aufgerufen wird:

```
void push1 (CStack s) { s.push("foo"); }
```

Da das Objekt `s` auf dem Stack liegt, findet keine dynamische Bindung statt. Der Stack `CStack`, so wie er im vorigen Beispiel implementiert wurde, darf hier nicht verwendet werden, weil der Kopierkonstruktor `m_elems` der Kopie auf das gleiche Array wie im ursprünglichen Objekt zeigen lässt. Der Destruktor am Ende der Funktion `push1` gibt das Array dann frei. Außerhalb der Funktion wird das Array aber nochmals freigegeben, wir haben hier einen schweren Programmierfehler eingebaut, welcher zu undefiniertem Verhalten führt!

Dieses Problem wird ganz einfach durch die explizite Angabe eines Kopierkonstruktors in der Klasse `CStack` gelöst:

```
CStack (CStack const&cs) : m_size(0), m_maxsize(cs.m_maxsize)
{
    m_elems = new std::string[m_maxsize];
    for (int i=0; i<m_size; i++) m_elems[i] = cs.m_elems[i];
}
```

Ein Kopierkonstruktor wird immer dann verwendet, wenn ein Objekt durch ein anderes bei der Erzeugung initialisiert wird. Dabei ist zu beachten, dass hier zwei Schreibweisen unterstützt werden:

```
CStack orig (5);
CStack copy1 (orig);
CStack copy2 = orig;
```

Eine spätere Zuweisung mit `=` ruft hingegen den Operator `operator=` auf.

Beim Aufruf von Methoden welche nicht `virtual` deklariert wurden, findet ebenfalls keine dynamische Bindung statt:

```
void empty (CStack const& s) { s.empty(); }
```

Da `CStack::empty` nicht `virtual` deklariert wurde, wird hier keine dynamische Bindung verwendet. Bei der Übergabe als Referenz – wie hier – oder als Zeiger ist es möglich, dynamisch zu binden. In diesen Fällen wird auch keine Kopie des Objektes angelegt, wodurch dann die Angabe des Kopierkonstruktors entfallen könnte. Aber man sollte allgemein immer bestrebt sein, dass die selbst geschriebenen Typen sich möglichst so verhalten wie eingebaute Typen. Sollten sie es nicht tun, muss das einen guten Grund haben und das muss auch dokumentiert werden.

> **Tipp:** Definiere einen eigenen Kopierkonstruktor, wenn das default Verhalten unerwünscht ist.

Es findet also immer dann eine dynamische Bindung statt, wenn die Methode `virtual` deklariert wurde und das Objekt über eine Referenz oder Zeiger angesprochen wird:

```
void push2 (CStack& s) { s.push("foo"); }
void push3 (CStack* s) { s->push("foo"); }
```

In beiden Fällen wird durch `CounterStack::push` die Variable `m_counter` inkrementiert, wenn der tatsächliche dynamische Typ ein `CounterStack` ist, anderenfalls wird `CStack::push` ausgeführt. Das Konzept ist vor allem deshalb interessant, weil der Methode auch beliebige andere, eventuell

derzeit noch gar nicht geschriebene, Subtypen übergeben werden können ohne die Methoden `push2` und `push3` ändern zu müssen.

Neben der *Einfachvererbung* (single inheritance) die bei `CounterStack` verwendet wurde, unterstützt C++ auch deren Verallgemeinerung, die *Mehrfachvererbung* (multiple inheritance). Die Motivation dafür war hauptsächlich, um abhängige Konzepte, welche in Klassen realisiert sind, gemeinsam zu repräsentieren [Str87]. Wir wollen uns ein kleines Beispiel anschauen:

```
class Satellite: public Task, public Displayed {};
```

Durch die `public` Vererbung wurde wie in `CounterStack` eine Untertypbeziehung eingeführt, ein `Satellite` *ist* somit auch ein `Task` und `Displayed`. Der unmittelbare Vorteil ist, dass die Funktionalität von `Task` und `Displayed` nicht mit vielen neuen Funktionen nach außen delegiert werden muss – dadurch wird Tipparbeit gespart. Der wesentliche Vorteil aber ist, dass bei dem Prozessscheduler und die GUI, welche nur die Typen `Task` und `Displayed` kennen, ein `Satellite` übergeben werden kann. Diese Funktionalität ist nicht anders als mit Mehrfachvererbung lösbar. Mehrdeutigkeiten in den Methoden werden wie bei `CounterStack` durch qualifizierte Namen aufgelöst. Eine andere Taktik muß allerdings für Variablen verwendet werden, wenn mehrfach die selbe Basisklasse (das sogenannte *Diamond Problem*) vorkommt. In diesem Fall kann durch die *virtuelle Vererbung* das Duplizieren der Variablen verhindert werden:

```
class Tier { protected: std::string m_name; };
class Pferd : public virtual Tier { };
class Vogel: public virtual Tier { };
class Pegasus : public Pferd, public Vogel { };
```

Mittlerweile hat Mehrfachvererbung viel Anklang gefunden und wird verwendet, um eigene Klassen mit Klassen, von denen man den Source-Code nicht hat, zu kombinieren, Policy-Based Design mit Templates umzusetzen (hier wird von jeder Policy abgeleitet) und um Interfaces beliebig kombinieren zu können.

Interfaces sind Klassen welche nur rein virtuelle Methoden anbieten. Bei rein virtuellen Methoden fehlt die Implementation. Der Compiler weiß, dass eine Methode rein virtuell ist, wenn `=0` nach der Methodendeklaration erfolgt. Diese Klassen können nicht instanziert werden, man nennt sie deshalb auch abstrakte Klassen (siehe Kapitel 2.2.3). Hier ist ein Beispiel für eine Interface Klasse:

Listing 1.8: interface.h

```
#include <string>
```

```
class IStack
{
    virtual void push (std::string const& elem) = 0;
    virtual std::string pop() = 0;
    virtual ~IStack() {}
};
```

Solche Interfaces sind Schnittstellen, wie wir in Abschnitt 2.4 sehen werden.

(Wir verwenden in diesem Skriptum für die Konvention der Klassen den englischen Begriff, während wir mit dem gleichbedeutenden deutschen Begriff Schnittstellen im Allgemeinen bezeichnen.)

1.2 Qualität in der Programmierung

Die Qualität in der Programmierung gliedert sich in zwei Bereiche:

- die Qualität der erstellten Programme
- sowie die Effizienz der Erstellung und Wartung der Programme.

Nur wenn die Qualität beider Bereiche zufriedenstellend ist, kann man brauchbare Ergebnisse erwarten. Diese beiden Bereiche sind eng ineinander verflochten: Ein qualitativ hochwertiges Programm erleichtert die Wartung, und eine effiziente Programmerstellung lässt im Idealfall mehr Zeit zur Verbesserung des Programms.

Wir betrachten zunächst die Qualität der Programme und anschließend die Effizienz der Programmerstellung und Wartung.

1.2.1 Qualität von Programmen

Bei der Qualität eines Programms unterscheiden wir zwischen

- der Brauchbarkeit (usability) des Programms,
- der Zuverlässigkeit des Programms
- und der Wartbarkeit des Programms.

Die Brauchbarkeit durch die AnwenderInnen steht natürlich an erster Stelle. Nur wenn die AnwenderInnen ihre tatsächlichen Aufgaben mit dem Programm zufriedenstellend lösen können, hat es für die AnwenderInnen einen Wert. Für SoftwareentwicklerInnen ist ein Softwareprojekt in der

Regel nur dann (sehr) erfolgreich, wenn der Wert des entwickelten Programms aus Sicht der Benutzer die Entwicklungskosten (stark) übersteigt.

Folgende Faktoren beeinflussen die Brauchbarkeit:

Zweckerfüllung: Die AnwenderInnen möchten mit einem Programm eine gegebene Klasse von Aufgaben lösen. Das Programm erfüllt seinen Zweck nur dann, wenn es genau die Aufgaben lösen kann, für die es eingesetzt wird. Features – das sind Eigenschaften – eines Programms, die AnwenderInnen nicht brauchen, haben keinen Einfluss auf die Zweckerfüllung. Allerdings können nicht benötigte Features die Brauchbarkeit durch größeren Ressourcenbedarf und schlechtere Bedienbarkeit negativ beeinflussen.

Bedienbarkeit: Die Bedienbarkeit besagt, wie effizient Aufgaben mit Hilfe des Programms lösbar sind und wie hoch der Einlernaufwand ist. Die Bedienbarkeit ist gut, wenn vor allem für häufig zu lösende Aufgaben möglichst wenige Arbeitsschritte nötig sind, keine unerwartet langen Wartezeiten entstehen und zur Bedienung keine besonderen Schulungen notwendig sind. Oft hängt die Bedienbarkeit aber auch von den Gewohnheiten und Erfahrungen der AnwenderInnen ab.

Effizienz des Programms: Jeder Programmablauf benötigt Ressourcen wie Rechenzeit, Hauptspeicher, Plattenspeicher und Netzwerkbandbreite. Ein Programm, das sparsamer mit solchen Ressourcen umgeht, hat eine höhere Qualität als ein weniger sparsames. Das gilt auch dann, wenn Computer in der Regel über ausreichend Ressourcen verfügen, denn wenn das Programm zusammen mit anderen Anwendungen läuft, können die Ressourcen trotzdem knapp werden. Das sparsamere Programm ist unter Umständen auch gleichzeitig mit anderen ressourcenverbrauchenden Anwendungen nutzbar.

Neben der Brauchbarkeit ist die *Zuverlässigkeit* sehr wichtig. Das Programm soll nicht nur manchmal brauchbar sein, sondern AnwenderInnen sollen sich darauf verlassen können. Fehlerhafte Ergebnisse und Programmabstürze sollen nicht vorkommen. Natürlich ist die geforderte Zuverlässigkeit von der Art der Anwendung abhängig. Für Software im Sicherheitssystem eines Kernkraftwerks wird ein weitaus höherer Grad an Zuverlässigkeit gefordert als für ein Textverarbeitungssystem. Absolute Zuverlässigkeit kann aber nie garantiert werden. Da die Zuverlässigkeit ein bedeutender Kostenfaktor ist, gibt man sich bei nicht sicherheitskritischen Anwendungen mit geringerer Zuverlässigkeit zufrieden, als erreichbar ist.

Oft lebt ein Programm nur so lange es weiterentwickelt wird. Sobald die Entwicklung und Weiterentwicklung – einschließlich laufender Fehlerkorrekturen und Anpassungen an sich ändernde Bedingungen, das heißt *Wartung* (maintenance) – abgeschlossen ist, kann ein Programm kaum mehr verkauft werden, und AnwenderInnen steigen auf andere Programme um. Daraus erkennt man, dass gerade bei erfolgreicher Software, die über einen langen Zeitraum verwendet wird – also einen langen *Lebenszyklus* hat –, die Wartungskosten einen erheblichen Teil der Gesamtkosten ausmachen. Man schätzt, dass die Wartungskosten bis zu 70 % der Gesamtkosten ausmachen, bei sehr erfolgreicher Software sogar weit mehr.

Faustregel: Gute *Wartbarkeit* kann die Gesamtkosten erheblich reduzieren.

Es gibt große Unterschiede in der Wartbarkeit von Programmen. Sie beziehen sich darauf, wie leicht Programme geändert werden können. Folgende Faktoren spielen eine Rolle:

Einfachheit: Ein einfaches Programm ist leichter verständlich und daher auch leichter änderbar als ein kompliziertes. Deswegen soll die Komplexität des Programms immer so klein wie möglich bleiben.

Lesbarkeit: Die Lesbarkeit ist gut, wenn es für ProgrammiererInnen einfach ist, durch Lesen des Programms die Logik im Programm zu verstehen und eventuell vorkommende Fehler oder andere zu ändernde Stellen zu entdecken. Die Lesbarkeit hängt zu einem guten Teil vom Programmierstil ab, aber auch von der Programmiersprache.

Lokalität: Der Effekt jeder Programmänderung soll auf einen kleinen Programmteil beschränkt bleiben. Dadurch wird vermieden, dass eine Änderung Programmteile beeinflusst, die auf den ersten Blick gar nichts mit der Änderung zu tun haben. Nicht-lokale beziehungsweise globale Effekte der Änderung – z. B. ein eingefügter Prozeduraufruf überschreibt den Wert einer globalen Variable – werden von ProgrammiererInnen oft nicht gleich erkannt und führen zu Fehlern.

Faktorisierung: Zusammengehörige Eigenschaften und Aspekte des Programms sollen zu Einheiten zusammengefasst werden. In Analogie zur Zerlegung eines Polynoms in seine Koeffizienten nennt man die Zerlegung eines Programms in Einheiten mit zusammengehörigen Eigenschaften *Faktorisierung* (factoring). Wenn zum Beispiel mehrere

Stellen in einem Programm aus denselben Sequenzen von Befehlen bestehen, soll man diese Stellen durch Aufrufe einer Routine ersetzen, die genau diese Befehle ausführt. Gute Faktorisierung führt dazu, dass zur Änderung aller dieser Stellen auf die gleiche Art und Weise eine einzige Änderung der Routine ausreicht. Bei schlechter Faktorisierung hätten alle Programmstellen gefunden und einzeln geändert werden müssen, um denselben Effekt zu erreichen. Gute Faktorisierung verbessert auch die Lesbarkeit des Programms, beispielsweise dadurch, dass die Routine einen Namen bekommt, der ihre Bedeutung widerspiegelt.

Objekte dienen durch Kapselung zusammengehöriger Eigenschaften in erster Linie der Faktorisierung des Programms. Durch Zusammenfügen von Daten mit Routinen haben ProgrammiererInnen mehr Freiheiten zur Faktorisierung als in der prozeduralen Programmierung, bei der Daten prinzipiell von Routinen getrennt sind (siehe Abschnitt 1.4).

Faustregel: Gute Faktorisierung kann die Wartbarkeit eines Programms wesentlich erhöhen.

Zur Klarstellung: Die objektorientierte Programmierung bietet mehr Möglichkeiten zur Faktorisierung als andere Paradigmen und erleichtert damit ProgrammiererInnen, eine für das Problem geeignete Zerlegung in einzelne Objekte, Module und Komponenten zu finden. Aber die Faktorisierung eines Programms erfolgt auf keinen Fall automatisch so, dass alle Zerlegungen in Objekte gut sind. Es ist die Aufgabe der ProgrammiererInnen, gute Zerlegungen von schlechten zu unterscheiden.

Die Lesbarkeit eines objektorientierten Programms kann man erhöhen, indem man es so in Objekte zerlegt, wie es der Erfahrung in der *realen Welt* entspricht. Das heißt, Software-Objekte sollen die reale Welt simulieren, soweit dies zur Erfüllung der Aufgaben sinnvoll erscheint. Vor allem Namen für Software-Objekte sollen den üblichen Bezeichnungen realer Objekte entsprechen. Dadurch ist das Programm einfacher lesbar, da stets die Analogie zur realen Welt besteht, vorausgesetzt alle EntwicklerInnen haben annähernd dieselben Vorstellungen über die reale Welt. Man darf die Simulation aber nicht zu weit treiben. Vor allem soll man keine Eigenschaften der realen Welt simulieren, die für die entwickelte Software bedeutungslos sind. Die Einfachheit ist wichtiger.

Faustregel: Man soll die reale Welt simulieren, aber nur so weit, dass die Komplexität dadurch nicht erhöht wird.

1.2.2 Effizienz der Programmerstellung und Wartung

Die große Zahl der Faktoren, die die Qualität eines Programms bestimmen, machen es ProgrammiererInnen schwer, qualitativ hochwertige Programme zu schreiben. Dazu kommt das Problem, dass viele Einflussgrößen zu Beginn der Entwicklung noch nicht bekannt sind. Einige davon sind von SoftwareentwicklerInnen nicht kontrollierbar. Zum Beispiel wissen AnwenderInnen oft nicht genau, welche Eigenschaften des Programms sie zur Lösung ihrer Aufgaben tatsächlich brauchen. Erfahrungen mit dem Programm können sie ja erst sammeln, wenn das Programm existiert.

Ein typischer Softwareentwicklungsprozess umfasst folgende Schritte:

Analyse (analysis): Die Aufgabe, die durch die zu entwickelnde Software gelöst werden soll, wird analysiert. Das Ergebnis, das ist die *Anforderungsdokumentation*, beschreibt die Anforderungen an die Software – was die Software tun soll.

Entwurf (design): Ausgehend von dieser Anforderungsdokumentation wird in der Entwurfsphase das Programm entworfen. Die *Entwurfsdokumentation* beschreibt, wie Anforderungen erfüllt werden sollen.

Implementierung (implementation): Der Entwurf wird in ein Programm umgesetzt. In diesem Schritt erzeugte Programmstücke werden Implementierungen (entsprechender Konzepte im Entwurf, die Beschreibungen des Verhaltens darstellen) genannt.

Verifikation (verification) und Validierung (validation): Die Verifikation ist die Überprüfung, ob das Programm die in der Anforderungsdokumentation beschriebenen Anforderungen erfüllt. Validierung ist die Überprüfung, wie gut das Programm die Aufgaben der AnwenderInnen tatsächlich löst und ob die Qualität des Programms dessen Weiterentwicklung rechtfertigt.

Im traditionellen *Wasserfallmodell* werden diese Schritte in der gegebenen Reihenfolge durchgeführt, gefolgt von einem Schritt für die Wartung. Solche Softwareentwicklungsprozesse haben den Nachteil, dass die Validierung erst sehr spät erfolgt. Es können also bereits recht hohe Kosten

angefallen sein, bevor festgestellt werden kann, ob die entwickelte Software für die AnwenderInnen überhaupt brauchbar ist. Das Risiko ist groß. Bei kleinen Projekten und in Fällen, in denen die Anforderungen sehr klar sind, kann das Wasserfallmodell aber durchaus vorteilhaft sein.

> **Faustregel:** Das Wasserfallmodell eignet sich für kleinere Projekte mit sehr klaren Anforderungen.

Heute verwendet man eher *zyklische Softwareentwicklungsprozesse*. Dabei werden die oben genannten Schritte in einem Zyklus wiederholt ausgeführt. Zuerst wird nur ein kleiner, aber wesentlicher Teil der durchzuführenden Aufgabe analysiert und ein entsprechendes Programm entworfen, implementiert, verifiziert und validiert. Im nächsten Zyklus wird das Programm erweitert, wobei die Erfahrungen mit dem ersten Programm in die Analyse und den Entwurf einfließen. Diese Zyklen werden fortgesetzt, solange das Programm lebt, also auch zur Wartung. In der Praxis werden die Zyklen und die einzelnen Schritte in den Zyklen jedoch meist nicht streng in der beschriebenen Reihenfolge durchgeführt, sondern häufig überlappend. Das heißt, es wird gleichzeitig analysiert, entworfen, implementiert und überprüft. Wenn man mit einem kleinen Teil des Programms beginnt und das Programm schrittweise ausweitet, spricht man auch von *schrittweiser Verfeinerung*. Die Vorteile solcher Entwicklungsprozesse liegen auf der Hand: Man kann bereits recht früh auf Erfahrungen mit dem Programm zurückgreifen, und die Gefahr, dass unter hohem Aufwand im Endeffekt nicht gebrauchte Eigenschaften in das Programm eingebaut werden, ist kleiner. Aber der Fortschritt eines Softwareprojekts ist nur schwer planbar. Daher kann es leichter passieren, dass sich die Qualität eines Programms zwar ständig verbessert, das Programm aber nie zum praktischen Einsatz gelangt, da die Mittel vorher erschöpft sind oder der Bedarf nicht mehr existiert.

> **Faustregel:** Zyklische Prozesse verkraften Anforderungsänderungen besser, aber Zeit und Kosten sind schwer planbar.

In der Praxis eingesetzte Entwicklungsprozesse unterscheiden sich stark voneinander. Jedes Unternehmen hat eigene Standards. Alles vom Wasserfallmodell bis zu sehr dynamischen zyklischen Prozessen kommt vor.

Qualitätsunterschiede zwischen einzelnen Softwareentwicklungsprozessen sind kaum greifbar, da viele Faktoren mitspielen und es nur wenige

vergleichbare Daten gibt. Zum Beispiel hängt die Qualität eines bestimmten Prozesses von der Art der Softwareprojekte ebenso ab wie von der internen Unternehmenskultur – Organisationsstruktur, Fähigkeiten der Mitarbeiter, etc. – und der Art der Zusammenarbeit mit Kunden und Partnern.

Jedes Unternehmen ist bestrebt, die eigenen Entwicklungsprozesse zu verbessern. Sobald irgendwo ein Problem auftaucht, wird es gelöst. Gerade solche oft durchgeführten kleinen Anpassungen führen schließlich zu einem konkurrenzfähigen Softwareentwicklungsprozess. Generell gilt, dass nur ein gut an die tatsächlichen Gegebenheiten angepasster Prozess von hoher Qualität ist. In der Regel funktioniert es nicht, wenn ein Unternehmen einen Softwareentwicklungsprozess von einem anderen Unternehmen übernimmt, ohne ihn an die eigenen Gegebenheiten anzupassen.

1.3 Rezept für gute Programme

Der Titel dieses Abschnitts ist ironisch zu verstehen. Niemand kann ein allgemeingültiges Rezept dafür angeben, wie man gute Programme schreibt. Dafür ist die Softwareentwicklung in ihrer Gesamtheit viel zu komplex. Nach wie vor ist die Programmierung eine Kunst – vor allem die Kunst, trotz unvollständigen Wissens über künftige Anforderungen, trotz vieler widersprüchlicher Zielsetzungen und oft unter großem Zeitdruck Lösungen zu entwickeln, die über einen längeren Zeitraum brauchbar sind. Das ist keine leichte Aufgabe. Ein einfaches Rezept, das immer zu guten Ergebnissen führt, sofern man alle vorgeschriebenen Schritte korrekt durchführt, wird es vermutlich nie geben.

Trotzdem hat sich in den vergangenen Jahrzehnten auch in der Programmierung ein umfangreicher Erfahrungsschatz darüber entwickelt, mit welchen Problemen man in Zukunft rechnen muss, wenn man eine Aufgabenstellung auf eine bestimmte Art und Weise löst. Gute ProgrammiererInnen werden diese Erfahrungen gezielt einsetzen. Eine Garantie für den Erfolg eines Softwareprojekts gibt es natürlich trotzdem nicht. Aber die Wahrscheinlichkeit, dass EntwicklerInnen die Komplexität des Projekts meistern können, steigt. Damit können noch komplexere Aufgabenstellungen mit vertretbaren Erfolgsaussichten in Angriff genommen werden.

Gerade in der objektorientierten Programmierung ist es wichtig, dass EntwicklerInnen Erfahrungen gezielt einsetzen. Objektorientierte Sprachen bieten viele unterschiedliche Möglichkeiten zur Lösung von Aufgaben.

Jede Lösungsmöglichkeit hat andere charakteristische Merkmale. Erfahrene EntwicklerInnen werden jene Möglichkeit wählen, deren Merkmale in späterer Folge am ehesten hilfreich sind. Weniger erfahrene EntwicklerInnen wählen einfach nur die Lösungsmöglichkeit, die sie zuerst entdecken. Damit verzichten sie auf einen wichtigen Vorteil der objektorientierten Programmierung gegenüber einigen anderen Paradigmen. Generell kann man sagen, dass die objektorientierte Programmierung durch erfahrene EntwicklerInnen derzeit wahrscheinlich das erfolgversprechendste Paradigma der Programmierung überhaupt darstellt, andererseits aber GelegenheitsprogrammiererInnen und noch unerfahrene SoftwareentwicklerInnen oft überfordert.

1.3.1 Zusammenhalt und Kopplung

Ein gutes Programm erfüllt die Kriterien, die wir in Abschnitt 1.2.1 beschrieben haben. Leider sind einige wichtige Kriterien in der Entwurfsphase und während der Implementierung noch nicht bewertbar. Sie stellen sich erst später heraus. SoftwareentwicklerInnen müssen aber in jeder Phase wissen, wie sie vorgehen müssen, um möglichst hochwertige Software zu produzieren. Vor allem eine gute Faktorisierung des Programms ist ein entscheidendes Kriterium. Daher gibt es Faustregeln, die EntwicklerInnen dabei unterstützen. Wir wollen hier zwei wichtige, eng miteinander verknüpfte Faustregeln betrachten, die in vielen Fällen den richtigen Weg zu guter Faktorisierung weisen. Zuvor führen wir einige Begriffe ein [3]:

Verantwortlichkeiten (responsibilities): Wir können die Verantwortlichkeiten einer Klasse durch drei w-Ausdrücke beschreiben:

- „was ich weiß“ – Beschreibung des Zustands der Instanzen
- „was ich mache“ – Verhalten der Instanzen
- „wen ich kenne“ – sichtbare Objekte, Klassen, etc.

Das Ich steht dabei jeweils für die Klasse. Die Klasse muss die Verantwortung für diese Verantwortlichkeiten übernehmen. Wenn etwas geändert werden soll, das in den Verantwortlichkeiten einer Klasse liegt, dann sind dafür die EntwicklerInnen der Klasse zuständig.

Klassen-Zusammenhalt (class coherence): Der Zusammenhalt einer Klasse ist der *Grad der Beziehungen zwischen den Verantwortlichkeiten der Klasse.* Dieser Grad der Beziehungen ist zwar nicht einfach

messbar, oft aber intuitiv einfach fassbar. Der Zusammenhalt ist offensichtlich hoch, wenn alle Variablen und Methoden der Klasse eng zusammenarbeiten und durch den Namen der Klasse gut beschrieben sind. Das heißt, einer Klasse mit hohem Zusammenhalt fehlt etwas Wichtiges, wenn man beliebige Variablen oder Methoden entfernt. Außerdem wird der Zusammenhalt niedriger, wenn man die Klasse sinnändernd umbenennt.

Objekt-Kopplung (object coupling): Unter der Objekt-Kopplung versteht man die *Abhängigkeit der Objekte voneinander.* Die Objekt-Kopplung ist stark, wenn

- die Anzahl der nach außen sichtbaren Methoden und Variablen groß ist,
- im laufenden System Nachrichten (beziehungsweise Methodenaufrufe) und Variablenzugriffe zwischen unterschiedlichen Objekten häufig auftreten
- und die Anzahl der Parameter dieser Methoden groß ist.

Das sind die Faustregeln:

Faustregel: Der Klassen-Zusammenhalt soll hoch sein.

Ein hoher Klassen-Zusammenhalt deutet auf eine gute Zerlegung des Programms in einzelne Klassen beziehungsweise Objekte hin – gute Faktorisierung. Bei guter Faktorisierung ist die Wahrscheinlichkeit, dass bei Programmänderungen auch die Zerlegung in Klassen und Objekte geändert werden muss (*Refaktorisierung*, refactoring), kleiner. Natürlich ist es bei hohem Zusammenhalt schwierig, bei Refaktorisierungen den Zusammenhalt beizubehalten oder noch weiter zu erhöhen.

Faustregel: Die Objekt-Kopplung soll schwach sein.

Schwache Objekt-Kopplung deutet auf gute Kapselung hin, bei der Objekte voneinander so unabhängig wie möglich sind. Dadurch beeinflussen Programmänderungen wahrscheinlich weniger Objekte unnötig. Beeinflussungen durch unvermeidbare Abhängigkeiten zwischen Objekten sind unumgänglich.

Klassen-Zusammenhalt und Objekt-Kopplung stehen in einer engen Beziehung zueinander. Wenn der Klassen-Zusammenhalt hoch ist, dann

ist oft die Objekt-Kopplung schwach und umgekehrt. Da Menschen auch dann sehr gut im Assoziieren zusammengehöriger Dinge sind, wenn sie Details noch gar nicht kennen, ist es relativ leicht, bereits in einer frühen Phase der Softwareentwicklung zu erkennen, auf welche Art und Weise ein hoher Klassen-Zusammenhalt und eine schwache Objekt-Kopplung erreichbar sein wird. Die Simulation der realen Welt hilft dabei vor allem zu Beginn der Softwareentwicklung.

Wenn EntwicklerInnen sich zwischen mehreren Alternativen zu entscheiden haben, können Klassen-Zusammenhalt und Objekt-Kopplung der einzelnen Alternativen einen wichtigen Beitrag zur Entscheidungsfindung liefern. Der erwartete Klassen-Zusammenhalt sowie die erwartete Objekt-Kopplung jeder Alternative lässt sich im direkten Vergleich einigermaßen sicher prognostizieren. Klassen-Zusammenhalt und Objekt-Kopplung sind Faktoren in der Bewertung von Alternativen. In manchen Fällen können jedoch andere Faktoren ausschlaggebend sein.

Auch noch so erfahrene EntwicklerInnen werden es kaum schaffen, auf Anhieb einen optimalen Entwurf für ein Programm zu liefern, in dem die Zerlegung in Objekte später nicht mehr geändert zu werden braucht. Normalerweise muss die Zerlegung einige Male geändert werden; man spricht von Refaktorisierung. Eine Refaktorisierung ändert die Struktur eines Programms, lässt aber dessen Funktionalität unverändert. Es wird dabei also nichts hinzugefügt oder weggelassen, und es werden auch keine inhaltlichen Änderungen vorgenommen. Solche Refaktorisierungen sind vor allem in einer frühen Projektphase ohne größere Probleme und Kosten möglich und werden durch eine Reihe von Werkzeugen unterstützt. Glücklicherweise ist es oft so, dass einige wenige gezielt durchgeführte Refaktorisierungen sehr rasch zu einer stabilen Zerlegung der davon betroffenen Programmteile in Objekte führen und später diese stabilen Teile kaum noch refaktorisiert zu werden brauchen. Es geht also gar nicht darum, von Anfang an einen optimalen Entwurf zu haben, sondern eher darum, ständig alle nötigen Refaktorisierungen durchzuführen bevor sich Probleme, die durch die Refaktorisierungen beseitigt werden, über das ganze Programm ausbreiten. Natürlich dürfen Refaktorisierungen auch nicht so häufig durchgeführt werden, dass bei der inhaltlichen Programmentwicklung überhaupt kein Fortschritt mehr erkennbar ist.

Faustregel: Ein vernünftiges Maß rechtzeitiger Refaktorisierungen führt häufig zu gut faktorisierten Programmen.

1.3.2 Wiederverwendung

Ein wichtiger Begriff im Zusammenhang mit effizienter Softwareentwicklung ist die *Wiederverwendung* (reuse). Es ist sinnvoll, bewährte Software so oft wie möglich wiederzuverwenden. Das spart Entwicklungsaufwand. Wir müssen aber zwischen zahlreichen Arten der Wiederverwendung unterscheiden. Hier sind einige Arten von Software, die wiederverwendet werden können:

Programme: Die meisten Programme werden im Hinblick darauf entwickelt, dass sie häufig (wieder)verwendet werden. Dadurch zahlt es sich erst aus, einen großen Aufwand zu betreiben, um die Programme handlich und effizient zu machen. Es gibt aber auch Programme, die nur für die einmalige Verwendung bestimmt sind.

Daten: Auch Daten in Datenbanken und Dateien werden in vielen Fällen häufig wiederverwendet. Nicht selten haben Daten eine längere Lebensdauer als die Programme, die sie benötigen oder manipulieren.

Erfahrungen: Häufig unterschätzt wird die Wiederverwendung von Konzepten und Ideen in Form von Erfahrungen. Diese Erfahrungen werden oft zwischen sehr unterschiedlichen Projekten ausgetauscht.

Code: Wenn man von Wiederverwendung spricht, meint man oft automatisch die Wiederverwendung von Programmcode. Viele Konzepte von Programmiersprachen, wie zum Beispiel enthaltender Polymorphismus, Vererbung und Generizität, wurden insbesondere im Hinblick auf die Wiederverwendung von Code entwickelt. Man kann mehrere Arten der Codewiederverwendung mit verschiedenen Wiederverwendungshäufigkeiten unterscheiden:

Globale Bibliotheken: Einige Klassen in allgemein verwendbaren Klassenbibliotheken – zum Beispiel als Standardbibliotheken zusammen mit Programmierwerkzeugen oder separat erhältlich – werden sehr häufig (wieder)verwendet. Allerdings kommen nur wenige, relativ einfache Klassen für die Aufnahme in solche Bibliotheken in Frage. Die meisten etwas komplexeren Klassen sind nur in bestimmten Bereichen sinnvoll einsetzbar und daher für die Allgemeinheit nicht brauchbar.

Fachspezifische Bibliotheken: Komplexere Klassen und *Komponenten* – größere Einheiten bzw. Objekte, meist aus mehreren

Klassen zusammengesetzt – lassen sich in fach- oder auch firmenspezifischen Bibliotheken unterbringen. Ein Beispiel dafür sind Bibliotheken für grafische Benutzeroberflächen. Auch mit solchen Bibliotheken lässt sich manchmal ein hoher Grad an Wiederverwendung erreichen, aber wieder sind die am häufigsten wiederverwendeten Klassen und Komponenten eher einfacher Natur.

Projektinterne Wiederverwendung: Zu einem hohen Grad spezialisierte Klassen und Komponenten lassen sich oft nur innerhalb eines Projekts, zum Beispiel in unterschiedlichen Versionen eines Programms, wiederverwenden. Obwohl der damit erzielbare Grad der Wiederverwendung nicht sehr hoch ist, ist diese Art der Wiederverwendung bedeutend: Wegen der höheren Komplexität der wiederverwendeten Software erspart bereits eine einzige Wiederverwendung viel Arbeit.

Programminterne Wiederverwendung: Ein und derselbe Programmcode kann in einem Programm sehr oft wiederholt ausgeführt werden, auch zu unterschiedlichen Zwecken. Durch die Verwendung eines Programmteils für mehrere Aufgaben wird das Programm einfacher, kleiner und leichter wartbar.

Gute SoftwareentwicklerInnen werden nicht nur darauf schauen, dass sie so viel Software wie möglich wiederverwenden, sondern auch darauf, dass neu entwickelte Software einfach wiederverwendbar wird. Die Erfahrung zeigt, dass durch objektorientierte Programmierung tatsächlich Code-Wiederverwendung erzielbar ist. Kosteneinsparungen ergeben sich dadurch aber normalerweise nur, wenn

- SoftwareentwicklerInnen ausreichend erfahren sind, um die Möglichkeiten der objektorientierten Programmierung optimal zu nutzen
- und Zeit in die Wiederverwendbarkeit investiert wird.

Weniger erfahrene EntwicklerInnen investieren oft zu wenig oder zu viel oder an falscher Stelle in die Wiederverwendbarkeit von Klassen und Komponenten. Solche Fehlentscheidungen können sich später rächen und durch lange Entwicklungszeiten sogar zum Scheitern eines Projekts führen. Im Zweifelsfall soll man anfangs eher weniger in die Wiederverwendbarkeit investieren, diese Investitionen zum Beispiel durch Refaktorisierung aber nachholen, sobald sich ein Bedarf dafür ergibt.

Faustregel: Code-Wiederverwendung erfordert beträchtliche Investitionen in die Wiederverwendbarkeit. Man soll diese tätigen, wenn ein tatsächlicher Bedarf dafür absehbar ist.

1.3.3 Entwurfsmuster

Erfahrung ist eine wertvolle Ressource zur effizienten Erstellung und Wartung von Software. Am effizientesten ist es, gewonnene Erfahrungen in Programmcode auszudrücken und diesen Code direkt wiederzuverwenden. Aber in vielen Fällen funktioniert Code-Wiederverwendung nicht. In diesen Fällen muss man zwar den Code neu schreiben, kann dabei aber auf bestehende Erfahrungen zurückgreifen.

In erster Linie betrifft die Wiederverwendung von Erfahrung die persönlichen Erfahrungen der SoftwareentwicklerInnen. Aber auch kollektive Erfahrung ist von großer Bedeutung. Gerade für den Austausch kollektiver Erfahrung können Hilfsmittel nützlich sein.

In den letzten Jahren sind sogenannte *Entwurfsmuster* (design patterns) als ein solches Hilfsmittel populär geworden. Entwurfsmuster geben im Softwareentwurf immer wieder auftauchenden Problemstellungen und deren Lösungen Namen, damit die EntwicklerInnen einfacher darüber sprechen können. Außerdem beschreiben Entwurfsmuster, welche Eigenschaften man sich von den Lösungen erwarten kann. EntwicklerInnen, die einen ganzen Katalog möglicher Lösungen für ihre Aufgaben entweder in schriftlicher Form oder nur abstrakt vor Augen haben, können gezielt jene Lösungen auswählen, deren Eigenschaften den erwünschten Eigenschaften der zu entwickelnden Software am ehesten entsprechen. Kaum eine Lösung wird nur gute Eigenschaften haben. Häufig wählt man daher jene Lösung, deren Nachteile man am ehesten für akzeptabel hält.

Jedes Entwurfsmuster besteht im Wesentlichen aus folgenden vier Elementen:

Name: Der Name ist wichtig, damit man in einem einzigen Begriff ein Problem und dessen Lösung sowie Konsequenzen daraus ausdrücken kann. Damit kann man den Softwareentwurf auf eine höhere Ebene verlagern; man braucht nicht mehr jedes Detail einzeln anzusprechen. Der Name ist auch nützlich, wenn man über ein Entwurfsmuster diskutiert. Es ist gar nicht leicht, solche Namen für Entwurfsmuster zu finden, die jeder mit dem Entwurfsmuster assoziiert. Wir verwenden

hier *Factory Method* (siehe Abschnitt 4.1.1) als Beispiel für ein einfaches Entwurfsmuster.

Problemstellung: Das ist die Beschreibung des Problems zusammen mit dessen Umfeld. Daraus geht hervor, unter welchen Bedingungen das Entwurfsmuster überhaupt anwendbar ist. Bevor man ein Entwurfsmuster in Betracht zieht, muss man sich überlegen, ob die zu lösende Aufgabe mit dieser Beschreibung übereinstimmt. Für Factory Method lautet die Beschreibung folgendermaßen: „Eine Factory Method definiert eine Schnittstelle für die Objekterzeugung, wobei Unterklassen entscheiden, von welcher Klasse die erzeugten Objekte sein sollen; die tatsächliche Objekterzeugung wird in Unterklassen verschoben.“ Falls wir beispielsweise Unterklassen vermeiden wollen, müssen wir auf ein anderes Entwurfsmuster wie etwa „Prototype“ ausweichen.

Lösung: Das ist die Beschreibung einer bestimmten Lösung der Problemstellung. Diese Beschreibung ist allgemein gehalten, damit sie leicht an unterschiedliche Situationen angepasst werden kann. Sie soll jene Einzelheiten enthalten, die zu den beschriebenen Konsequenzen führen, aber nicht mehr. Im Beispiel der Factory Method enthält die Beschreibung Erklärungen dafür, wie die Klassenstrukturen aussehen, welche Abhängigkeiten zwischen den Klassen bestehen, und welche Arten von Methoden geeignet sind.

Konsequenzen: Das ist eine Liste von Eigenschaften der Lösung. Man kann sie als eine Liste von Vor- und Nachteilen der Lösung betrachten, muss dabei aber aufpassen, da ein und dieselbe Eigenschaft in manchen Situationen einen Vorteil darstellt, in anderen einen Nachteil und in wieder anderen irrelevant ist. Eine Eigenschaft von Factory Method ist die höhere Flexibilität bei der Objekterzeugung, eine andere das Entstehen paralleler Klassenhierarchien mit einer oft großen Anzahl an Klassen.

Entwurfsmuster scheinen die Lösung vieler Probleme zu sein, da man nur mehr aus einem Katalog von Mustern zu wählen braucht, um eine ideale Lösung für ein Problem zu finden. Tatsächlich lassen sich Entwurfsmuster häufig so miteinander kombinieren, dass man alle gewünschten Eigenschaften erhält. Leider führt der exzessive Einsatz von Entwurfsmustern oft zu einem unerwünschten Effekt: Das entstehende Programm ist sehr komplex und undurchsichtig. Damit ist die Programmerstellung langwierig und die Wartung schwierig, obwohl die über den Einsatz der

Entwurfsmuster erzielten Eigenschaften anderes versprechen. SoftwareentwicklerInnen sollen also genau abwägen, ob es sich im Einzelfall auszahlt, eine bestimmte Eigenschaft auf Kosten der Programmkomplexität zu erzielen. Die Softwareentwicklung bleibt also auch dann eher eine Kunst als ein Handwerk, wenn Entwurfsmuster eingesetzt werden.

> **Faustregel:** Entwurfsmuster sollen zur Abschätzung der Konsequenzen von Designentscheidungen eingesetzt werden und können (in begrenztem Ausmaß) als Bausteine zur Erzielung bestimmter Eigenschaften dienen.

1.4 Paradigmen der Programmierung

Unter einem *Paradigma* der Programmierung versteht man im Wesentlichen einen Stil, in dem Programme geschrieben werden. Die meisten Programmiersprachen unterstützen einen bestimmten Stil besonders gut und weisen dafür charakteristische Merkmale auf. Am effektivsten wird man die Programmiersprache nutzen, wenn man Programme unter diesem Paradigma schreibt, also den durch die Programmiersprache zumindest zum Teil vorgegebenen Stil einhält. Dieses Paradigma soll natürlich mit der verwendeten Softwareentwicklungsmethode kompatibel sein [10].

Eine der wichtigsten Unterteilungen zwischen Paradigmen ist die zwischen *imperativer* und *deklarativer* Programmierung.

1.4.1 Imperative Programmierung

Die *Rechnerarchitektur* hinter der imperativen Programmierung beruht auf einem hardwarenahen Berechnungsmodell wie beispielsweise der „von Neumann-Architektur“: Eine CPU (central processing unit) ist über einen Bus mit einem Speichermodul verbunden. Die CPU führt zyklisch folgende Schritte aus: Ein Maschinenbefehl wird aus dem Speicher geladen und ausgeführt, und anschließend werden die Ergebnisse in den Speicher geschrieben. Praktisch alle derzeit verwendeten Computer beruhen auf einer ähnlichen Architektur.

Imperative Programmierung wird dadurch charakterisiert, dass Programme aus *Anweisungen* – das sind Befehle – aufgebaut sind. Diese werden in einer festgelegten Reihenfolge ausgeführt, in parallelen imperativen Programmen teilweise auch gleichzeitig beziehungsweise überlap-

pend. Grundlegende Sprachelemente sind Variablen, Konstanten und Routinen. Der wichtigste Befehl ist die *destruktive Zuweisung*: Eine Variable bekommt einen neuen Wert, unabhängig vom Wert, den sie vorher hatte. Die Menge der Werte in allen Variablen im Programm sowie ein Zeiger auf den nächsten auszuführenden Befehl beschreiben den Programmzustand, der sich mit der Ausführung jeder Anweisung ändert.

Im Laufe der Zeit entwickelte sich eine ganze Reihe von Paradigmen aufbauend auf der imperativen Programmierung. Unterschiede zwischen diesen Paradigmen beziehen sich hauptsächlich auf die Strukturierung von Programmen. Die wichtigsten imperativen Paradigmen sind die prozedurale und objektorientierte Programmierung:

Prozedurale Programmierung: Das ist der konventionelle Programmierstil. Der wichtigste Abstraktionsmechanismus in prozeduralen Sprachen wie z. B. Algol, Fortran, Cobol, C, Pascal und Modula-2 ist die Prozedur. Programme werden, den verwendeten Algorithmen entsprechend, in sich gegenseitig aufrufende, den Programmzustand verändernde Prozeduren zerlegt. Programmzustände werden im Wesentlichen als global angesehen, das heißt, Daten können an beliebigen Stellen im Programm verändert werden. Saubere prozedurale Programme schreibt man mittels *strukturierter Programmierung.*

Objektorientierte Programmierung: Die objektorientierte Programmierung ist eine Weiterentwicklung der strukturierten prozeduralen Programmierung, die den Begriff des Objekts in den Mittelpunkt stellt. Der wesentliche Unterschied zur prozeduralen Programmierung ist der, dass zusammengehörende Routinen und Daten zu Objekten zusammengefasst werden. In vielen Fällen ist es möglich, die Programmausführung anhand der Zustandsänderungen in den einzelnen Objekten zu beschreiben, ohne globale Änderungen der Programmzustände betrachten zu müssen. Das ist vor allem bei der Wartung vorteilhaft. Eine Konsequenz aus der Aufteilung von Routinen auf Objekte ist jedoch, dass ein Algorithmus manchmal nicht mehr an nur einer Stelle im Programm steht, sondern auf mehrere Objekte beziehungsweise Klassen aufgeteilt ist.

1.4.2 Deklarative Programmierung

Deklarative Programme beschreiben Beziehungen zwischen Ausdrücken in einem System. Es gibt keine zustandsändernden Anweisungen. Statt zeit-

lich aufeinanderfolgender Zustände gibt es ein sich nicht mit der Zeit änderndes Geflecht von Beziehungen zwischen Ausdrücken. Deklarative Sprachen entstammen mathematischen Modellen und stehen meist auf einem höheren Abstraktionsniveau als imperative Sprachen. Grundlegende Sprachelemente sind Symbole, die sich manchmal in mehrere Gruppen wie Variablensymbole, Funktionssymbole und Prädikate einteilen lassen. Daher spricht man auch von *symbolischer Programmierung.*

Die wichtigsten Paradigmen in der deklarativen Programmierung sind die funktionale und logikorientierte Programmierung:

Funktionale Programmierung: Eines der für die Informatik bedeutendsten theoretischen Modelle ist der *Lambda-Kalkül*, der den mathematischen Begriff *Funktion* formal definiert. Programmiersprachen, die auf diesem Kalkül beruhen, heißen funktionale Sprachen. Beispiele sind Lisp, ML und Haskell. Alle Ausdrücke in diesen Sprachen werden als Funktionen aufgefasst, und der wesentliche Berechnungsschritt besteht in der Anwendung einer Funktion auf einen Ausdruck. Der Lambda-Kalkül hat auch die historische Entwicklung der imperativen Sprachen beeinflusst. Manchmal werden funktionale Sprachen als saubere Varianten prozeduraler Sprachen angesehen, die ohne unsaubere destruktive Zuweisung auskommen. Durch das Fehlen der destruktiven Zuweisung und anderer Seiteneffekte haben funktionale Programme eine wichtige Eigenschaft, die als *referentielle Transparenz* bezeichnet wird: Ein Ausdruck in einem Programm bedeutet immer dasselbe, egal wann und wie der Ausdruck ausgewertet wird. Im Gegensatz zu anderen Paradigmen brauchen ProgrammiererInnen nicht zwischen einem Objekt und der Kopie des Objekts unterscheiden; solche Unterschiede werden nirgends sichtbar.

Logikorientierte Programmierung: Sprachen für die logikorientierte Programmierung beruhen auf einer (mächtigen) Teilmenge der Prädikatenlogik erster Stufe. Die Menge aller wahren Aussagen in einem Modell wird mittels Fakten und Regeln beschrieben. Um einen Berechnungsvorgang zu starten, wird eine Anfrage gestellt. Das Ergebnis der Berechnung besagt, ob und unter welchen Bedingungen die in der Anfrage enthaltene Aussage wahr ist. Der wichtigste Vertreter dieser Sprachen, Prolog, hat eine prozedurale Interpretation. Das heißt, Fakten und Regeln können als Prozeduren aufgefasst und wie in prozeduralen Sprachen ausgeführt werden. Spezielle Varianten der logikorientierten Programmierung spielen bei Datenbankabfragespra-

chen eine bedeutende Rolle.

Zusammenfassend kann man sagen, dass sich die oben beschriebenen Paradigmen vor allem im Umgang mit Programmzuständen voneinander unterscheiden. Im prozeduralen Paradigma sind Programmzustände nur global bewertbar, im objektorientierten Paradigma lokal gekapselt und in deklarativen Paradigmen auf verschiedene Arten abstrakt gehalten. Algorithmen stehen vor allem in den prozeduralen und funktionalen Paradigmen, durch die prozedurale Interpretation aber auch in der logikorientierten Programmierung zentral im Mittelpunkt. Diese Paradigmen eignen sich daher besonders gut für Aufgaben, die durch komplexe Algorithmen dominiert werden. Lokalität steht im Mittelpunkt der objektorientierten Programmierung, die deshalb eher für größere Aufgaben interessant ist, bei denen die Komplexität des Gesamtsystems jene der einzelnen Algorithmen deutlich übersteigt.

C++ unterstützt mehrere Paradigmen. Der imperative und prozedurale Teil wurde von C übernommen. Der objektorientierte Teil wurde in Anlehnung an Simula entwickelt. Die generische Programmierung ist später mit Hilfe der Templates hinzugefügt worden. Durch diese, welche eine eigene turing-vollständige Subsprache in C++ bilden, können auch funktionale Konzepte wie Funktoren, Lambda (anonyme Funktionen) und partielle Anwendung von Funktionen realisiert werden. Es hilft somit, C++ als eine Sammlung verschiedener Sprachen zu sehen[Mey05].

Die STL (Standard Template Library), wie aus ihrem Namen bereits zu entnehmen ist, setzt generische Programmierung intensiv ein. Die Container und Iteratoren erfüllen zwar die Anforderung der Datenkapselung, die besondere Stärke liegt aber in den Algorithmen, welche als generische globale Funktionen implementiert sind. Diese Algorithmen setzen keine Untertypbeziehungen im objektorientierten Sinne voraus[KS05].

Faustregel: Objektorientierte Programmierung eignet sich zur Entwicklung von Systemen, deren Gesamtkomplexität jene der einzelnen Algorithmen deutlich übersteigt. Sonst sind andere Paradigmen besser geeignet.

1.4.3 Paradigmen für Modularisierungseinheiten

Programmierparadigmen beziehen sich nicht nur auf das zu Grunde liegende Rechenmodell, sondern auch auf die Art und Weise, wie größere

Programme und Programmteile in kleinere Einheiten zerlegt werden, also auf die Faktorisierung. Folgende Paradigmen können im Prinzip sowohl mit imperativen als auch deklarativen Paradigmen kombiniert werden:

Programmierung mit abstrakten Datentypen: Ein abstrakter Datentyp, kurz ADT, versteckt die interne Darstellung seiner Instanzen. Nach außen hin wird eine Instanz als abstraktes Objekt ohne innere Struktur, beispielsweise als Adresse repräsentiert. Auf diesen Objekten sind nur die vom ADT exportierten Operationen anwendbar. Diese Zeiger und einige darauf anwendbare Routinen werden exportiert. In vielen objektorientierten Sprachen ist ein ADT ein Verbund (record bzw. struct), der neben Daten auch Routinen enthält. Einige Komponenten werden durch ein Typsystem vor Zugriffen von außen geschützt. Im Großen und Ganzen sind Klassen abstrakte Datentypen. Die Programmierung mit abstrakten Datentypen entspricht der objektorientierten Programmierung, abgesehen davon, dass auf die Verwendung von enthaltendem Polymorphismus und Vererbung (und damit auch auf dynamisches Binden) verzichtet wird.

Programmierung mit Modulen: Dieses Paradigma legt großen Wert auf Modularisierungseinheiten, das sind Gruppierungen von Variablen, Routinen, Typen, Klassen, etc. Ein Programm besteht aus einer Menge solcher Module. In einem Modul kann man angeben, welche Dienste das Modul nach außen hin anbietet und welche Dienste eines anderen Moduls im Modul verwendet werden. Dadurch ergibt sich eine natürliche Trennung eines Programms in voneinander weitgehend unabhängige Namensräume. Modula-2 und Ada sind für die Programmierung mit Modulen bekannt. Module sind unverzichtbar, wenn größere Programme in kleinere Einheiten zerlegt werden sollen. Module entsprechen Objekten, die direkt (ohne Verwendung von Klassen) vom Compiler erzeugt werden; die Erzeugung zur Laufzeit ist in der Regel nicht vorgesehen. Anders als in der objektorientierten Programmierung ist es bei der Programmierung mit Modulen weder nötig noch möglich, Module entsprechend ihres Verhaltens in Klassen einzuteilen. Da Module getrennte Übersetzungseinheiten darstellen ist die Verwendung von Modulen notwendigerweise zyklenfrei, das heißt, wenn ein Modul B ein anderes Modul A verwendet, dann kann A weder direkt noch indirekt B verwenden; andernfalls wäre es nicht möglich, A vor B zu compilieren.

Komponentenprogrammierung: Wie die Programmierung mit Modulen möchte die Komponentenprogrammierung ein großes Programm in möglichst unabhängige Programmteile zerlegen. Die Zielsetzung geht aber weiter: Es soll möglich sein, ein und dieselbe Komponente in unterschiedlichen Programmen einzusetzen und Komponenten gegen andere auszutauschen. Statt namentlicher Verweise werden nur Schnittstellen benötigter anderer Komponenten (required interfaces) angegeben, und das System wird meist erst zur Laufzeit aus vorhandenen Komponenten zusammengesetzt. Dazu brauchen wir normierte Schnittstellen und Funktionalität zum Zusammenfügen von Komponenten. Komponenten sind also Objekte mit klar spezifizierten Schnittstellen, die bestimmte Eigenschaften erfüllen (z.B. bestimmte Methoden implementieren) um einer Norm zu entsprechen. Die einzige sprachunabhängige Spezifikation eines Komponentenmodelles ist CCM (CORBA Component Model). Hier können neben C++ auch in Ada, C, Lisp, Ruby, Smalltalk, Java, COBOL, PL/I, Python und vielen anderen Sprachen (mittels nicht standardisierter Mappings), Komponenten geschrieben und verwendet werden. Zur Spezifikation von Schnittstellen dient die IDL (interface definition language). Im Gegensatz zu Modulen schränkt die getrennte Übersetzung Abhängigkeiten zwischen Komponenten nicht ein, da diese sich nicht namentlich aufeinander beziehen. Es kann zyklische Abhängigkeiten geben. Andererseits verursacht die von Komponentenmodellen verlangte zusätzliche Funktionalität (im Vergleich zu Modulen) deutlich längere Entwicklungszeiten und mehr Laufzeit-Ressourcen.

Generische Programmierung: Dieses Paradigma unterstützt die Entwicklung *generischer* Abstraktionen als modulare Programmeinheiten auf einer sehr hohen Ebene. Generische Einheiten werden zur Compilations- oder Laufzeit zu konkreten Datenstrukturen, Klassen, Typen, Funktionen, Prozeduren, etc. instanziert, die im Programm benötigt werden. Die generische Programmierung wird vor allem mit der objektorientierten und funktionalen Programmierung kombiniert. Zum Beispiel ist `std::list<T>` eine generische Klasse mit dem generischen Typparameter `T`, für den beliebige Typen eingesetzt werden können. Die konkreten Klassen `std::list<int>`, `std::list<float>` und `std::list<CPerson>` werden durch Instanzierung der generischen Klasse erzeugt, wobei `int`, `float` und `CPerson` den Typparameter ersetzen. Wir haben die Klassen der Listen von ganzen Zahlen,

Fließkommazahlen und Personen aus einer einzigen Klasse erzeugt.

Manchmal wird auch die objektorientierte Programmierung zu den Paradigmen gezählt, die beliebig mit imperativen und deklarativen Paradigmen kombinierbar sind. Tatsächlich gibt es funktionale und logikorientierte Sprachen, die auch die objektorientierte Programmierung unterstützen, wie beispielsweise Objective Caml und LIFE. Derzeit ist die deklarative objektorientierte Programmierung mit vielen Problemen behaftet oder beruht auf einer in eine deklarative Programmiersprache eingebetteten imperativen Teilsprache. In der Praxis wird nur die imperative objektorientierte Programmierung verwendet.

1.5 Wiederholungsfragen

Folgende Fragen können beim Erarbeiten des Stoffes helfen. Sie stellen aber auch eine vollständige Aufzählung möglicher mündlicher Prüfungsfragen dar.

1. Erklären Sie folgende Begriffe:
   - Objekt, Klasse, Vererbung
   - Identität, Zustand, Verhalten, Schnittstelle
   - Instanz einer Klasse, einer Schnittstelle und eines Typs
   - Deklaration und Definition
   - deklarierter, statischer und dynamischer Typ
   - Nachricht, Methode, Konstruktor
   - Faktorisierung, Refaktorisierung
   - Verantwortlichkeiten, Klassen-Zusammenhalt, Objekt-Kopplung
   - Softwareentwurfsmuster
2. Welche Arten von Polymorphismus unterscheidet man? Welche davon sind in der objektorientierten Programmierung wichtig? Warum?
3. Wann sind zwei gleiche Objekte identisch, und wann sind zwei identische Objekte gleich?
4. Sind Datenabstraktion, Datenkapselung und data hiding einander entsprechende Begriffe? Wenn Nein, worin unterscheiden sie sich?

5. Was besagt das Ersetzbarkeitsprinzip? (Hinweis: Sehr häufige Prüfungsfrage!)
6. Warum ist gute Wartbarkeit so wichtig?
7. Wie lauten die wichtigsten Faustregeln im Zusammenhang mit Klassenzusammenhalt und Objektkopplung? Welche Vorteile kann man sich davon erwarten, dass diese Faustregeln erfüllt sind?
8. Welche Arten von Software kann man wiederverwenden, und welche Rolle spielt jede davon in der Softwareentwicklung?
9. Welche Rolle spielen Refaktorisierungen in der Wiederverwendung?
10. Nennen Sie die wichtigsten Paradigmen der Programmierung und ihre essentiellen Eigenschaften.
11. Wofür ist die objektorientierte Programmierung gut geeignet, und wofür ist sie nicht gut geeignet?
12. Erkläre den Ablauf der Laufzeit eines Objektes. Welche Rolle spielen hier Konstruktoren und Destruktoren?
13. Wofür kann ein Namensbereich eingesetzt werden? Wie verwendet man einen?
14. Was ist Mehrfachvererbung? Wofür wird sie benötigt? Wie können Interfaces in C++ realisiert werden? Gibt es einen Unterschied in der Mehrfachvererbung von Interfaces und sonstigen Klassen?

Kapitel 2

Enthaltender Polymorphismus und Vererbung

Vererbung und enthaltender Polymorphismus sind auf Grund ihrer Definitionen zwei sehr unterschiedliche Konzepte, die aber häufig in einem einzigen Sprachkonstrukt zusammengefasst sind: In C++ ist `public` Vererbung ist auf solche Weise eingeschränkt, dass sie auch die wichtigsten Anforderungen des enthaltenden Polymorphismus erfüllen kann.

In diesem Kapitel werden wir zunächst in Abschnitt 2.1 die Grundlagen des enthaltenden Polymorphismus untersuchen. In Abschnitt 2.2 gehen wir auf einige wichtige Aspekte des Objektverhaltens ein, die ProgrammiererInnen bei der Verwendung von enthaltendem Polymorphismus beachten müssen. Danach betrachten wir in Abschnitt 2.3 einige Aspekte der Vererbung, vor allem im Zusammenhang mit Codewiederverwendung. Schließlich behandeln wir in Abschnitt 2.4 Klassen, Vererbung und das Konzept der Interfaces in C++ .

2.1 Das Ersetzbarkeitsprinzip

Die wichtigste Grundlage des enthaltenden Polymorphismus ist das Ersetzbarkeitsprinzip:

Definition: Ein Typ U ist ein Untertyp eines Typs T, wenn eine Instanz von U überall verwendbar ist, wo eine Instanz von T erwartet wird.

Über dieses Ersetzbarkeitsprinzip definieren wir enthaltenden Polymorphismus. Daher ist, per Definition, eine Instanz eines Untertyps überall verwendbar, wo eine Instanz eines Obertyps erwartet wird. Insbesondere benötigt man das Ersetzbarkeitsprinzip für

- den Aufruf einer Routine mit einem Argument, dessen Typ ein Untertyp des Typs des entsprechenden formalen Parameters ist
- und für die Zuweisung eines Objekts an eine Variable, wobei der Typ des Objekts ein Untertyp des deklarierten Typs der Variable ist.

Beide Fälle kommen in der objektorientierten Programmierung häufig vor.

2.1.1 Untertypen und Schnittstellen

Wann ist das Ersetzbarkeitsprinzip erfüllt? Diese Frage wird in der Fachliteratur intensiv behandelt [1]. Wir wollen die Frage hier nur so weit beantworten, als es in der Praxis relevant ist. Als Beispiel für eine praktisch verwendete Sprache verwenden wir hier stets C++ , obwohl fast alles, was hier über C++ gesagt wird, auch für Sprachen wie C#, Java und Eiffel gilt. Wir gehen davon aus, dass Typen Schnittstellen von Objekten sind, die in Klassen spezifiziert wurden. Es gibt in C++ auch Typen wie `int`, die keiner Klasse entsprechen. Aber für solche Typen gibt es in C++ keine Untertypen. Deshalb werden wir sie hier nicht näher betrachten.

Eine Voraussetzung für das Bestehen einer Untertypbeziehung in C++ ist, dass auf den entsprechenden Klassen eine Vererbungsbeziehung besteht. Die dem Untertyp entsprechende Klasse muss also von der dem Obertyp entsprechenden Klasse direkt oder indirekt `public` abgeleitet sein. Solche Voraussetzungen sind praktisch sinnvoll, wie wir später sehen werden. Man kann Untertypbeziehungen aber auch ohne eine solche Voraussetzung definieren. Objective-C und Smalltalk [11] sind Beispiele für Sprachen, in denen man Vererbung nicht als Voraussetzung für das Bestehen einer Untertypbeziehung ansieht.

Nun wollen wir einige allgemeingültige (nicht auf eine konkrete Programmiersprache bezogene) Bedingungen für das Bestehen einer Untertypbeziehung betrachten. Alle Untertypbeziehungen sind

- reflexiv – jeder Typ ist Untertyp von sich selbst,
- transitiv – ist ein Typ U Untertyp eines Typs S und ist S Untertyp eines Typs T, dann ist U auch Untertyp von T,

- und antisymmetrisch – ist ein Typ U Untertyp eines Typs T und ist T außerdem Untertyp von U, dann sind U und T gleich.

Generell ist U ein Untertyp von T (wobei U und T beliebige Typen beziehungsweise Schnittstellen sind), wenn folgende Bedingungen erfüllt sind:

- Für jede Konstante (das ist eine Variable, die nach der Initialisierung nur lesende Zugriffe erlaubt) in T gibt es eine entsprechende Konstante in U, wobei der deklarierte Typ B der Konstante in U ein Untertyp des deklarierten Typs A der Konstante in T ist.

  Begründung: Auf eine Konstante kann von außerhalb des Objekts nur lesend zugegriffen werden. Wenn man die Konstante in einem Objekt vom Typ T liest, erwartet man sich, dass man ein Ergebnis vom Typ A erhält. Diese Erwartung soll auch erfüllt sein, wenn das Objekt vom Typ U ist, wenn also eine Instanz von U verwendet wird, wo eine Instanz von T erwartet wird. Auf Grund der Bedingung gibt es im Objekt vom Typ U eine entsprechende Konstante vom Typ B. Da B ein Untertyp von A sein muss, ist die Erwartung immer erfüllt.

- Für jede Variable in T gibt es eine entsprechende Variable in U, wobei die deklarierten Typen der Variablen gleich sind.

  Begründung: Auf eine Variable kann lesend und schreibend zugegriffen werden. Ein lesender Zugriff entspricht der oben beschriebenen Situation bei Konstanten; der deklarierte Typ B der Variable in U muss ein Untertyp des deklarierten Typs A der Variable in T sein. Wenn man eine Variable eines Objekts vom Typ T von außerhalb des Objekts schreibt, erwartet man sich, dass man jede Instanz vom Typ A der Variablen zuweisen darf. Diese Erwartung soll auch erfüllt sein, wenn das Objekt vom Typ U und die Variable vom Typ B ist. Die Erwartung ist nur erfüllt, wenn A ein Untertyp von B ist. Wenn man lesende und schreibende Zugriffe gemeinsam betrachtet, muss B ein Untertyp von A und A ein Untertyp von B sein. Da Untertypbeziehungen antisymmetrisch sind, müssen A und B gleich sein.

- Für jede Methode in T gibt es eine entsprechende Methode in U, wobei der deklarierte Ergebnistyp der Methode in U ein Untertyp des Ergebnistyps der Methode in T ist, die Anzahl der formalen Parameter der beiden Methoden gleich ist und der deklarierte Typ jeden formalen Parameters in U ein Obertyp des deklarierten Typs des entsprechenden formalen Parameters in T ist.

Begründung: Für die Ergebnistypen der Methoden gilt dasselbe wie für Typen von Konstanten beziehungsweise lesende Zugriffe auf Variablen: Der Aufrufer einer Methode möchte ein Ergebnis des in T versprochenen Ergebnistyps bekommen, auch wenn tatsächlich die entsprechende Methode in U ausgeführt wird. Für die Typen der formalen Parameter gilt dasselbe wie für schreibende Zugriffe auf Variablen: Der Aufrufer möchte alle Argumente der Typen an die Methode übergeben können, die in T deklariert sind, auch wenn tatsächlich die entsprechende Methode in U ausgeführt wird. Daher dürfen die Parametertypen in U nur Obertypen der Parametertypen in T sein.

Diese Beziehung für Parametertypen gilt nur für Argumente die nur vom Aufrufer an die aufgerufene Methode übergeben werden (Eingangsparameter). In Ada können aber auch Objekte an den Aufrufer zurückgegeben werden (Ausgangsparameter). Für die Typen solcher Parameter gelten dieselben Bedingungen wie für Ergebnistypen. Mit Referenzen und Zeigern ist es auch möglich, dass über ein und denselben Parameter ein Argument an die Methode übergeben und von dieser ein (anderes) Argument an den Aufrufer zurückgegeben wird (Durchgangsparameter). Die deklarierten Typen solcher Parameter müssen in U und T gleich sein. In C++ kann, durch Setzen von entsprechenden `const` Modifier, dieses Verhalten allerdings unterbunden werden. Dann kann davon ausgegangen werden, dass die formalen Parameter nicht modifiziert werden (Eingangsparameter).

Diese Bedingungen hängen nur von den Strukturen der Typen ab und berücksichtigen das Verhalten in keiner Weise. Außerdem verlangt keine dieser Bedingungen, dass ein Untertyp explizit aus einem Obertyp abgeleitet werden muss. Untertypbeziehungen können auch implizit zwischen Typen gegeben sein, die zufällig zusammenpassende Strukturen haben.

Ein Untertyp kann nicht nur einen Obertyp um neue Elemente erweitern, sondern auch deklarierte Typen der einzelnen Elemente gegenüber dem Obertyp ändern; das heißt, die deklarierten Typen der Elemente können variieren. Je nach dem, wie diese Typen variieren können, spricht man von Kovarianz, Kontravarianz und Invarianz:

Kovarianz: Der deklarierte Typ eines Elements im Untertyp ist ein Untertyp des deklarierten Typs des entsprechenden Elements im Obertyp. Zum Beispiel sind deklarierte Typen von Konstanten und von Ergebnissen der Methoden (so wie von Ausgangsparametern) kovariant. Typen und die betrachteten darin enthaltenen Elementtypen

variieren in dieselbe Richtung.

Kontravarianz: Der deklarierte Typ eines Elements im Untertyp ist ein Obertyp des deklarierten Typs des Elements im Obertyp. Zum Beispiel sind deklarierte Typen von formalen Eingangsparametern kontravariant. Typen und die betrachteten darin enthaltenen Elementtypen variieren in entgegengesetzte Richtungen.

Invarianz: Der deklarierte Typ eines Elements im Untertyp ist gleich dem deklarierten Typ des entsprechenden Elements im Obertyp. Zum Beispiel sind deklarierte Typen von Variablen und Durchgangsparametern invariant. Die betrachteten in den Typen enthaltenen Elementtypen variieren nicht.

Betrachten wir ein Beispiel in C++:

Listing 2.1: Kontravariante Parameter

```
class Derived;

class Base
{
public:
    virtual Base& f(Derived const&) { return *this; }
    virtual ~Base() {}
};

class Derived: public Base
{
public:
    virtual Derived& f(Base const&) { return *this; }
};
```

Entsprechend den oben angeführten Bedingungen ist `Derived` ein Untertyp von `Base`. Obwohl Ersetzbarkeit gegeben wäre, unterstützt C++ allerdings keine Kontravarianz für formale Parameter, in diesem Beispiel haben Instanzen von `Derived` zwei überladene Methoden. Die Methode wird nicht überschrieben.

Obige Bedingungen für Untertypbeziehungen sind notwendig und in gewisser Weise (solange Verhalten ausgeklammert bleibt) auch vollständig. Man kann keine weglassen oder aufweichen, ohne mit dem Ersetzbarkeitsprinzip in Konflikt zu kommen. Die meisten dieser Bedingungen stellen keine praktische Einschränkung dar. ProgrammiererInnen kommen kaum in Versuchung sie zu brechen. Nur eine Bedingung, nämlich die geforderte Kontravarianz von formalen Parametertypen, möchte man manchmal gerne umgehen. Sehen wir uns dazu ein Beispiel an:

Listing 2.2: Kovariante Parameter

```
class Point2D
{
protected:
    int x,y;
public:
    Point2D (int x1, int y1): x(x1), y(y1) {}
    virtual bool operator == (Point2D const& p) const
    {
        return (x==p.x && y==p.y);
    }
    virtual ~Point2D() {}
};

class Point3D: public Point2D
{
protected:
    int z;
public:
    Point3D (int x1, int y1, int z1): Point2D(x1,y1), z(z1) {}
    virtual bool operator == (Point3D const& p) const
    {
        return (x==p.x && y==p.y && z==p.z);
    }
};
```

Der hier verwendete Operator `operator==` verhält sich wie eine normale Methode. Es wird eine Art konventionelle Syntax unterstützt:

```
void equal1(Point2D const& p1, Point2D const& p2)
{
    if (p1.operator==(p2)) cout << "equal" << endl;
}
```

Allerdings kann der Aufruf dann auch einfacher und verständlicher gestaltet werden:

```
void equal2(Point2D const& p1, Point2D const& p2)
{
    if (p1==p2) cout << "equal" << endl;
}
```

Diese Funktionalität ist aber viel mehr als syntaktischer Zucker, wie wir in 2.4.3 sehen werden.

In diesem Programmstück erfüllt der Operator `operator==` nicht die Kriterien für Untertypbeziehungen, da der Parametertyp kovariant und nicht, wie gefordert, kontravariant ist. Der Operator `operator==` in `Point3D` kann jene in `Point2D` daher nicht überschreiben. Eine Methode, bei der ein formaler Parametertyp stets gleich der Klasse ist, in der die Methode definiert ist, heißt *binäre Methode*. Die Eigenschaft *binär* bezieht sich darauf, dass der Name der Klasse in der Methode mindestens zwei mal vorkommt – einmal als Typ von `this` und mindestens einmal als Typ eines expliziten

Parameters. Binäre Methoden sind über den einfachen enthaltenden Polymorphismus, wie wir ihn hier verwenden, prinzipiell nicht realisierbar. In C++ wird dazu, wie in diesem Beispiel, der Operator `operator==` überladen, nicht überschrieben; die Klasse `Point3D` hat somit zwei Operatoren `operator==`.

> **Faustregel:** Kovariante Eingangsparametertypen und binäre Methoden widersprechen dem Ersetzbarkeitsprinzip. Es ist sinnlos, in solchen Fällen Ersetzbarkeit anzustreben.

Untertypbeziehungen sind in C++ stärker eingeschränkt, als es durch obige Bedingungen notwendig wäre: Parametertypen sind invariant. Da überladene Methoden durch die Typen der formalen Parameter unterschieden werden, wäre es schwierig, überladene Methoden von Methoden mit kontravariant veränderten Typen auseinander zu halten.

Ergebnistypen hingegen sind kovariant:

Listing 2.3: kovariante Ergebnistypen

```
class Base
{
public:
    virtual Base* clone() {return new Base;}
    virtual ~Base() {}
};

class Derived: public Base
{
public:
    virtual Derived* clone() {return new Derived;}
};
```

Wie erwartet wird `clone` in `Derived` überschrieben. Damit kann auch der deklarierte Typ Zeiger oder Referenz auf `Base` das Objekt `Derived` zurückgeben.

Untertypbeziehungen in C++ setzen entsprechende Vererbungsbeziehungen voraus. `public` Vererbung ist in C++ ist so eingeschränkt, dass diese Bedingungen bei Verwendung von `virtual` für Untertypbeziehungen erfüllt sind. Die Bedingungen werden bei der Übersetzung eines C++ -Programms überprüft.

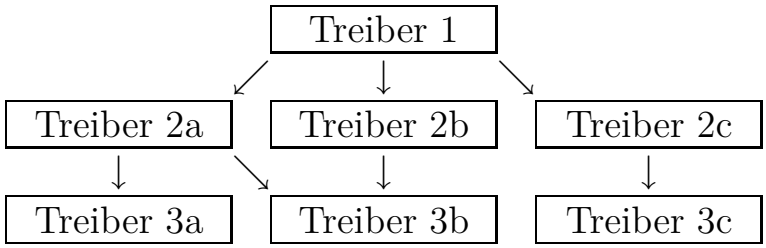
2.1.2 Untertypen und Codewiederverwendung

Die wichtigste Entscheidungsgrundlage für den Einsatz des enthaltenden Polymorphismus ist sicherlich die erzielbare Wiederverwendung. Der rich-

tige Einsatz von enthaltendem Polymorphismus eröffnet durch das Ersetzbarkeitsprinzip einige Möglichkeiten, die auf den ersten Blick aber gar nicht so leicht zu erkennen sind.

Nehmen wir als Beispiel die Treiber-Software für eine Grafikkarte. Anfangs genügt ein einfacher Treiber für einfache Ansprüche. Wir entwickeln eine Klasse, die den Code für den Treiber enthält und nach außen eine Schnittstelle anbietet, über die wir die Funktionalität des Treibers verwenden können. Letzteres ist der Typ des Treibers. Wir schreiben einige Anwendungen, die die Treiberklasse verwenden. Daneben werden vielleicht auch von anderen EntwicklerInnen, die wir nicht kennen, Anwendungen erstellt, die unsere Treiberklasse verwenden. Alle Anwendungen greifen über dessen Schnittstelle beziehungsweise Typ auf den Treiber zu.

Mit der Zeit wird unser einfacher Treiber zu primitiv. Wir entwickeln einen neuen, effizienteren Treiber, der auch Eigenschaften neuerer Grafikkarten verwenden kann. Wir erben von der alten Klasse und lassen die Schnittstelle unverändert, abgesehen davon, dass wir neue Methoden hinzufügen. Nach obiger Definition ist der Typ der neuen Klasse ein Untertyp des alten Typs. Neue Treiber – das sind Instanzen des Treibertyps – können überall verwendet werden, wo alte Treiber erwartet werden. Daher können wir in den vielen Anwendungen, die den Treiber bereits verwenden, den alten Treiber ganz einfach gegen den neuen austauschen, ohne die Anwendungen sonst irgendwie zu ändern. In diesem Fall haben wir Wiederverwendung in großem Umfang erzielt: Viele Anwendungen sind sehr einfach auf einen neuen Treiber umgestellt worden. Darunter sind auch Anwendungen, die wir nicht einmal kennen. Das Beispiel können wir beliebig fortsetzen, indem wir immer wieder neue Varianten von Treibern schreiben und neue Anwendungen entwickeln oder bestehende Anwendungen anpassen, die die jeweils neuesten Eigenschaften der Treiber nützen. Dabei kann es natürlich auch passieren, dass aus einer Treiberversion mehrere weitere Treiberversionen entwickelt werden, die nicht zueinander kompatibel sind. Folgendes Bild zeigt, wie die Treiberversionen nach drei Generationen aussehen könnten:



An diesem Bild fällt die Version 3b auf: Sie vereinigt die zwei inkompati-

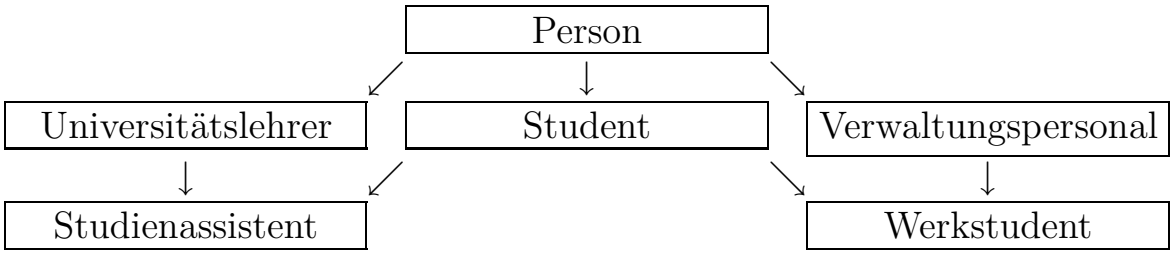
blen Vorgängerversionen 2a und 2b. Ein Untertyp kann mehrere Obertypen haben, die zueinander in keiner Untertypbeziehung stehen.

> **Faustregel:** Man soll auf Ersetzbarkeit achten, um Codewiederverwendung zwischen Versionen zu erreichen.

Die Wiederverwendung zwischen verschiedenen Versionen funktioniert nur dann gut, wenn die Schnittstellen bzw. Typen zwischen den Versionen stabil bleiben. Das heißt, eine neue Version darf die Schnittstellen nicht beliebig ändern, sondern nur so, dass die in Abschnitt 2.1.1 beschriebenen Bedingungen erfüllt sind. Im Wesentlichen kann die Schnittstelle also nur erweitert werden. Wenn die Aufteilung eines Programms in einzelne Objekte gut ist, bleiben Schnittstellen normalerweise recht stabil.

> **Faustregel:** Schnittstellen sollen stabil bleiben. Gute Faktorisierung hilft dabei.

Das, was in obigem Beispiel für verschiedene Versionen einer Klasse funktioniert, kann man genauso gut innerhalb eines einzigen Programms nutzen, wie wir an einem modifizierten Beispiel sehen. Wir wollen ein Programm zur Verwaltung der Personen an einer Universität entwickeln. Die dafür verwendete Klassenstruktur könnte so aussehen:



Entsprechend diesen Strukturen sind StudienassistentInnen sowohl UniversitätslehrerInnen als auch StudentInnen, und WerkstudentInnen an der Universität gehören zum Verwaltungspersonal und sind StudentInnen. Wir benötigen im Programm eine Komponente, die Serienbriefe – Einladungen zu Veranstaltungen, etc. – an alle Personen adressiert. Für das Erstellen einer Anschrift benötigt man nur Informationen aus der Klasse `Person`. Die entsprechende Methode braucht nicht zwischen verschiedenen Arten von Personen unterscheiden, sondern funktioniert für jede Instanz des (deklarierten) Typs `Person`, auch wenn es tatsächlich eine Instanz des (dynamischen) Typs `Studienassistent` ist. Diese Methode wird also für

alle Arten von Personen (wieder)verwendet. Ebenso funktioniert eine Methode zum Ausstellen eines Zeugnisses für alle Instanzen von `Student`, auch wenn es StudienassistentInnen oder WerkstudentInnen sind.

> **Faustregel:** Man soll auf Ersetzbarkeit achten, um interne Codewiederverwendung im Programm zu erzielen.

Solche Klassenstrukturen können helfen, Auswirkungen nötiger Programmänderungen möglichst lokal zu halten. Wenn man eine Klasse, zum Beispiel `Student`, ändert, bleiben andere Klassen, die nicht von `Student` erben, unberührt. Anhand der Klassenstruktur ist leicht erkennbar, welche Klassen von der Änderung betroffen sein können. Unter „betroffen“ verstehen wir dabei, dass als Folge der Änderung möglicherweise weitere Änderungen in den betroffenen Programmteilen nötig sind. Die Änderung kann nicht nur diese Klassen selbst betreffen, sondern auch alle Programmstellen, die auf Instanzen der Typen `Student`, `Studienassistent` oder `Werkstudent` zugreifen. Aber Programmteile, die auf Instanzen von `Person` zugreifen, sollten von der Änderung auch dann nicht betroffen sein, wenn die Instanzen tatsächlich vom dynamischen Typ `Student` sind. Diese Programmteile haben keine Zugriffsmöglichkeit auf geänderte Eigenschaften der Instanzen.

> **Faustregel:** Man soll auf Ersetzbarkeit achten, um Programmänderungen lokal zu halten.

Falls bei der nötigen Programmänderung alle Schnittstellen der Klasse unverändert bleiben, betrifft die Änderung keine Programmstellen, an denen `Student` und dessen Unterklassen verwendet werden. Lediglich diese Klassen selbst sind betroffen. Auch daran kann man sehen, wie wichtig es ist, dass Schnittstellen und Typen möglichst stabil sind. Eine Programmänderung führt möglicherweise zu vielen weiteren nötigen Änderungen, wenn dabei eine Schnittstelle geändert wird. Die Anzahl wahrscheinlich nötiger Änderungen hängt auch davon ab, wo in der Klassenstruktur die geänderte Schnittstelle steht. Eine Änderung ganz oben in der Struktur hat wesentlich größere Auswirkungen als eine Änderung ganz unten. Eine Schlussfolgerung aus diesen Überlegungen ist, dass man möglichst nur von solchen Klassen erben soll, deren Schnittstellen bereits – oft nach mehreren Refaktorisierungsschritten – recht stabil sind.

Faustregel: Die Stabilität von Schnittstellen an der Wurzel der Typhierarchie ist wichtiger als an den Blättern. Man soll nur Untertypen von stabilen Obertypen bilden.

Aus obigen Überlegungen folgt auch, dass man die Typen von formalen Parametern möglichst allgemein halten soll. Wenn in einer Methode von einem Parameter nur die Eigenschaften von `Person` benötigt werden, sollte der Parametertyp `Person` sein und nicht `Werkstudent`, auch wenn die Methode voraussichtlich nur mit Argumenten vom Typ `Werkstudent` aufgerufen wird. Wenn aber die Wahrscheinlichkeit hoch ist, dass nach einer späteren Programmänderung in der Methode vom Parameter auch Eigenschaften von `Werkstudent` benötigt werden, sollte man gleich von Anfang an `Werkstudent` als Parametertyp verwenden, da nachträgliche Änderungen von Schnittstellen sehr teuer werden können.

Faustregel: Man soll Parametertypen vorausschauend und möglichst allgemein wählen.

Trotz der Wichtigkeit stabiler Schnittstellen darf man nicht den Fehler machen, bereits in einer frühen Phase der Softwareentwicklung zu viel Zeit in den detaillierten Entwurf der Schnittstellen zu investieren. Zu diesem Zeitpunkt hat man häufig noch nicht genug Information, um stabile Schnittstellen zu erhalten. Schnittstellen werden trotz guter Planung oft erst nach einigen Refakorisierungen stabil.

2.1.3 Dynamisches Binden

Bei Verwendung von enthaltendem Polymorphismus kann der dynamische Typ einer Variablen oder eines Parameters ein Untertyp des statischen beziehungsweise deklarierten Typs sein. Eine Variable vom Typ Referenz auf `Person` kann zum Beispiel eine Instanz von `Werkstudent` enthalten. Oft ist zur Übersetzungszeit des Programms der dynamische Typ nicht bekannt; das heißt, der dynamische Typ kann sich vom statischen Typ unterscheiden. Dann können Aufrufe einer Methode im Objekt, das in der Variable steht, erst zur Laufzeit an die auszuführende Methode gebunden werden. In C++ wird bei Verwendung von `virtual` bei Referenzen und Zeigern, unabhängig vom statischen Typ, immer die Methode ausgeführt, die in der Klasse des Objekts definiert ist. Dabei gilt der Grundsatz, dass bei allen Unterklassen die Methoden, bei Übereinstimmung der Signatur,

überschrieben werden – also implizit auch `virtual` sind. Die Schnittstelle dieser Klasse entspricht dem spezifischsten dynamischen Typ der Variablen.

Wir demonstrieren die Funktionsweise dynamischen Bindens an folgendem kleinen Beispiel:

Listing 2.4: Dynamic Binding Test

```
#include <string>
#include <iostream>

class A
{
public:
    virtual std::string foo1() const { return "foo1A"; }
    virtual std::string foo2() const { return fooX(); }
    virtual std::string fooX() const { return "foo2A"; }
    virtual ~A() {}
};

class B: public A
{
public:
    virtual std::string foo1() const { return "foo1B"; }
    virtual std::string fooX() const { return "foo2B"; }
};

void test (A const& x)
{
    std::cout << x.foo1() << std::endl;
    std::cout << x.foo2() << std::endl;
}

int main ()
{
    test(A());
    test(B());
}
```

Wenn wir das Programm compilieren und ausführen, erhalten wir am Bildschirm folgende Ausgabe:

```
foo1A
foo2A
foo1B
foo2B
```

Die ersten Zeilen sind einfach erklärbar: Nach dem Programmaufruf wird die Methode `main` ausgeführt, die `test` mit einer neuen Instanz von `A` als Argument aufruft. Diese Methode ruft zuerst `foo1` und dann `foo2` auf und gibt die Ergebnisse in den ersten beiden Zeilen aus. Dabei ent-

spricht der deklarierte Typ `A` des formalen Parameters `x` dem statischen und dynamischen Typ. Es werden daher `foo1` und `foo2` in `A` ausgeführt.

Der zweite Aufruf von `test` übergibt eine Instanz von `B` als Argument. Dabei ist `A` der deklarierte Typ von `x`, aber der dynamische Typ ist `B`. Wegen dynamischen Bindens werden diesmal `foo1` und `foo2` in `B` ausgeführt. Die dritte Zeile der Ausgabe enthält das Ergebnis des Aufrufs von `foo1` in einer Instanz von `B`.

Die letzte Zeile der Ausgabe lässt sich folgendermaßen erklären: Da die Klasse `B` die Methode `foo2` nicht überschreibt, wird `foo2` von `A` geerbt. Der Aufruf von `foo2` in `B` ruft `fooX` in der aktuellen Umgebung, das ist eine Instanz von `B`, auf. Die Methode `fooX` liefert als Ergebnis die Zeichenkette `"foo2B"`, die in der letzten Zeile ausgegeben wird.

Bei dieser Erklärung muss man vorsichtig sein: Man macht leicht den Fehler anzunehmen, dass `foo2` in `A` aufgerufen wird, da `foo2` ja nicht explizit in `B` steht, und daher `fooX` in `A` aufruft. Tatsächlich wird aber `fooX` in `B` aufgerufen, da `B` der spezifischste Typ der Umgebung ist.

Dynamisches Binden ist mit `switch`-Anweisungen und geschachtelten `if`-Anweisungen verwandt. Wir betrachten als Beispiel eine Funktion, die eine Anrede in einem Brief, deren Art auf konventionelle Weise über eine ganze Zahl bestimmt ist, auf die Standardausgabe schreibt:

```
void gibAnredeAus (int anredeArt, std::string name)
{
    switch (anredeArt)
    {
        case 1: std::cout << "S.g.␣Frau␣" << name << std::endl;
            break;
        case 2: std::cout << "S.g.␣Herr␣" << name << std::endl;
            break;
        default:std::cout << name << std::endl;
    }
}
```

In der objektorientierten Programmierung wird man die Art der Anrede eher durch die Klassenstruktur zusammen mit dem Namen beschreiben:

```
class Adressat
{
protected:
    std::string m_name;
public:
    virtual public void gibAnredeAus()
    {
        std::cout << m_name << std::endl;
    }
    virtual ~Adressat() {}
    ... // Konstruktoren und weitere Methoden
};
```

```
class WeiblicherAdressat : public Adressat
{
public:
    virtual public void gibAnredeAus()
    {
        std::cout << "S.g. Frau " << m_name << std::endl;
    }
};
class MaennlicherAdressat : public Adressat
{
public:
    virtual public void gibAnredeAus()
    {
        std::cout << "S.g. Herr " << m_name << std::endl;
    }
};
```

Durch dynamisches Binden wird automatisch die gewünschte Version von `gibAnredeAus()` aufgerufen. Statt einer `switch`-Anweisung wird in der objektorientierten Variante also dynamisches Binden verwendet. Ein Vorteil der objektorientierten Variante ist die bessere Lesbarkeit. Man weiß anhand der Namen, wofür bestimmte Unterklassen von `Adressat` stehen. Die Zahlen 1 oder 2 bieten diese Information nicht. Außerdem ist die Anredeart mit dem auszugebenden Namen verknüpft, wodurch man im Programm stets nur eine Instanz von `Adressat` anstatt einer ganzen Zahl und einem String verwalten muss. Ein anderer Vorteil der objektorientierten Variante ist besonders wichtig: Wenn sich herausstellt, dass neben „Frau“ und „Herr“ noch weitere Arten von Anreden, etwa „Firma“, benötigt werden, kann man diese leicht durch Hinzufügen einer weiteren Klasse einführen. Es sind keine zusätzlichen Änderungen nötig. Insbesondere bleiben die Methodenaufrufe unverändert.

Auf den ersten Blick mag es scheinen, als ob die konventionelle Variante mit `switch`-Anweisung kürzer und auch einfach durch Hinzufügen einer Zeile änderbar wäre. Am Beginn der Programmentwicklung trifft das oft auch zu. Leider haben solche `switch`-Anweisungen die Eigenschaft, dass sie sich sehr rasch über das ganze Programm ausbreiten. Beispielsweise gibt es bald auch spezielle Methoden zur Ausgabe der Anrede in generierten e-Mails, abgekürzt in Berichten, oder über Telefon als gesprochener Text, jede Methode mit zumindest einer eigenen `switch`-Anweisung. Dann ist es schwierig, zum Einfügen der neuen Anredeart alle solchen `switch`-Anweisungen zu finden und noch schwieriger, diese Programmteile über einen längeren Zeitraum konsistent zu halten. Die objektorientierte Lösung hat dieses Problem nicht, da alles auf die Klasse `Adressat` und ihre Unterklassen konzentriert ist. Es bleibt auch dann alles konzentriert, wenn

zu `gibAnredeAus()` weitere Methoden hinzukommen.

> **Faustregel:** Dynamisches Binden ist `switch`-Anweisungen und geschachtelten `if`-Anweisungen stets vorzuziehen.

2.2 Ersetzbarkeit und Objektverhalten

In Abschnitt 2.1 haben wir einige Bedingungen kennen gelernt, die erfüllt sein müssen, damit ein Typ Untertyp eines anderen Typs sein kann. Die Erfüllung dieser Bedingungen wird vom Compiler überprüft. Die Bedingungen sind aber nicht in jedem Fall ausreichend, um die uneingeschränkte Ersetzbarkeit einer Instanz eines Obertyps durch eine Instanz eines Untertyps zu garantieren. Dazu müssen weitere Bedingungen hinsichtlich des Objektverhaltens erfüllt sein, die von einem Compiler nicht überprüft werden können. SoftwareentwicklerInnen müssen ohne Compilerunterstützung sicherstellen, dass diese Bedingungen erfüllt sind.

2.2.1 Client-Server-Beziehungen

Für die Beschreibung des Objektverhaltens ist es hilfreich, das Objekt aus der Sicht anderer Objekte, die auf das Objekt zugreifen, zu betrachten. Man spricht von *Client-Server-Beziehungen* zwischen Objekten. Einerseits sieht man ein Objekt als einen *Server*, der anderen Objekten seine Dienste zur Verfügung stellt. Andererseits ist ein Objekt ein *Client*, der Dienste anderer Objekte in Anspruch nimmt. Die meisten Objekte spielen gleichzeitig die Rollen von Server und Client.

Für die Ersetzbarkeit von Objekten sind Client-Server-Beziehungen bedeutend. Man kann ein Objekt gegen ein anderes austauschen, wenn das neue Objekt als Server allen Clients zumindest dieselben Dienste anbietet wie das ersetzte Objekt. Um das gewährleisten zu können, brauchen wir eine Beschreibung der Dienste, also das Verhalten der Objekte.

Das *Objektverhalten* beschreibt, wie sich das Objekt beim Empfang einer Nachricht verhält, das heißt, was das Objekt beim Aufruf einer Methode macht. Diese Definition von Objektverhalten lässt etwas offen: Es ist unklar, wie exakt die Beschreibung dessen, was das Objekt tut, sein soll. Einerseits beschreibt die Schnittstelle eines Objekts das Objekt nur sehr unvollständig. Eine genauere Beschreibung wäre wünschenswert. Andererseits enthält die Implementierung des Objekts, also der Programmcode in

der Klasse, oft zu viele Implementierungsdetails, die bei der Betrachtung des Verhaltens hinderlich sind. Im Programmcode gibt es meist keine Beschreibung, deren Detailiertheitsgrad zwischen dem der Objektschnittstelle und dem der Implementierung liegt. Wir haben es beim Objektverhalten also mit einem abstrakten Begriff zu tun. Er wird vom Programmcode nicht notwendigerweise widergespiegelt.

Es hat sich bewährt, das Verhalten eines Objekts als einen Vertrag zwischen dem Objekt als Server und seinen Clients zu sehen (*Design by Contract*). Der Server muss diesen Vertrag ebenso einhalten wie jeder Client. Generell sieht der Vertrag folgendermaßen aus [20]:

> Jeder Client kann einen Dienst des Servers in Anspruch nehmen, wenn alle festgeschriebenen Bedingungen dafür erfüllt sind. Im Falle einer Inanspruchnahme setzt der Server alle festgeschriebenen Maßnahmen und liefert dem Client ein Ergebnis, das die festgeschriebenen Bedingungen erfüllt.

Im einzelnen regelt der Vertrag für jeden vom Server angebotenen Dienst, also für jede aufrufbare Methode, folgende Details:

Vorbedingungen (preconditions): Das sind Bedingungen, für deren Erfüllung vor Ausführung der Methode der Client verantwortlich ist. Vorbedingungen beschreiben hauptsächlich, welche Eigenschaften die Argumente, mit denen die Methode aufgerufen wird, erfüllen müssen. Zum Beispiel muss ein bestimmtes Argument ein Array von aufsteigend sortierten ganzen Zahlen im Wertebereich von 0 bis 99 sein. Vorbedingungen können auch den Zustand des Servers einbeziehen, soweit Clients diesen kennen. Zum Beispiel ist eine Methode nur aufrufbar, wenn eine Variable des Servers einen Wert größer 0 hat.

Nachbedingungen (postconditions): Für die Erfüllung dieser Bedingungen nach Ausführung der Methode ist der Server verantwortlich. Nachbedingungen beschreiben Eigenschaften des Methodenergebnisses und Änderungen beziehungsweise Eigenschaften des Objektzustandes. Als Beispiel betrachten wir eine Methode zum Einfügen eines Elements in eine Menge: Das Boolesche Ergebnis der Methode besagt, ob das Argument vor dem Aufruf bereits in der Menge enthalten war; am Ende muss das Argument auf jeden Fall in der Menge sein. Diese Beschreibung kann man als Nachbedingung auffassen.

Invarianten (invariants): Für die Erfüllung dieser Bedingungen sowohl vor als auch nach Ausführung jeder Methode ist grundsätzlich der

Server zuständig. Direkte Schreibzugriffe von Clients auf Variablen des Servers kann der Server aber nicht kontrollieren; dafür ist der Client verantwortlich. Zum Beispiel darf das Guthaben auf einem Sparbuch nie kleiner 0 sein, egal welche Operationen auf dem Sparbuch durchgeführt werden. Die Gültigkeit einer Invariante kann auch von Bedingungen abhängen. Zum Beispiel bleibt ein Objekt immer in einer Menge enthalten, sobald es eingefügt wurde. Jede Invariante impliziert eine Nachbedingung auf jeder Methode des Servers.

Vorbedingungen, Nachbedingungen und Invarianten sind verschiedene Arten von *Zusicherungen* (assertions).

Zum Teil sind Vorbedingungen und Nachbedingungen bereits in der Objektschnittstelle in Form von Parameter- und Ergebnistypen von Methoden beschrieben. Typkompatibilität wird vom Compiler überprüft. In der Programmiersprache Eiffel gibt es Sprachkonstrukte, mit denen man komplexere Zusicherungen schreiben kann [19]. Diese werden zur Laufzeit überprüft. Sprachen wie C++ unterstützen überhaupt keine Zusicherungen – abgesehen von trivialen `assert`-Anweisungen (siehe Ende Abschnitt 3.1.2), die sich aber kaum zur Beschreibung von Verträgen eignen. Sogar in Eiffel sind viele sinnvolle Zusicherungen nicht direkt ausdrückbar. In diesen Fällen kann und soll man Zusicherungen als Kommentare in den Programmcode schreiben und händisch überprüfen.

(für Interessierte)

Anmerkungen wie diese geben zusätzliche Informationen für interessierte Leser. Ihr Inhalt gehört nicht zum Prüfungsstoff.

Ein Beispiel in Eiffel soll veranschaulichen, wie Zusicherungen in Programmiersprachen integrierbar sind. Zu jeder Methode kann man vor der eigentlichen Implementierung (`do`-Klausel) eine Vorbedingung (`require`-Klausel) und nach der Implementierung eine Nachbedingung (`ensure`-Klausel) angeben. Invarianten stehen am Ende der Klasse. In jeder Zusicherung steht eine Liste Boolescher Ausdrücke, die durch Strichpunkt getrennt sind. Der Strichpunkt steht für eine Konjunktion (Und-Verknüpfung). Die Zusicherungen werden zur Laufzeit zu Ja oder Nein ausgewertet. Wird eine Zusicherung zu Nein ausgewertet, erfolgt eine Ausnahmebehandlung oder Fehlermeldung. In Nachbedingungen ist die Bezugnahme auf Variablen- und Parameterwerte zum Zeitpunkt des Methodenaufrufs erlaubt. Zum Beispiel bezeichnet `old guthaben` den Wert der Variable `guthaben` zum Zeitpunkt des Methodenaufrufs.

```
class KONTO feature {ANY}
    guthaben: Integer;
    ueberziehungsrahmen: Integer;
    einzahlen (summe: Integer) is
        require summe >= 0
```

```
            do guthaben := guthaben + summe
            ensure guthaben = old guthaben + summe
        end; -{}- einzahlen
        abheben (summe: Integer) is
            require summe >= 0;
                    guthaben + ueberziehungsrahmen >= summe
            do guthaben := guthaben - summe
            ensure guthaben = old guthaben - summe
        end; -{}- abheben
        invariant guthaben >= -ueberziehungsrahmen
    end -{}- class KONTO
```

Diese Klasse sollte bis auf einige syntaktische Details selbsterklärend sein. Die Klausel `feature {ANY}` besagt, dass die danach folgenden Variablendeklarationen und Methodendefinitionen überall im Programm sichtbar sind. Nach dem Schlüsselwort `end` und einem (in unserem Fall leeren) Kommentar kann zur besseren Lesbarkeit der Name der Methode oder der Klasse folgen.

Hier ist ein C++ -Beispiel für Kommentare als Zusicherungen:

```
class Konto
{
public:
    // public damit Zusicherungen von Client ueberprueft werden koennen
    long m_guthaben;
    long m_ueberziehungsrahmen;
    // guthaben >= -ueberziehungsrahmen
    // einzahlen addiert summe zu m_guthaben; summe >= 0
    void einzahlen (long summe)
    {
        m_guthaben = m_guthaben + summe;
    }
    // abheben zieht summe von m_guthaben ab;
    // summe >= 0; m_guthaben+m_ueberziehungsrahmen >= summe
    void abheben (long summe)
    {
        m_guthaben = m_guthaben - summe;
    }
};
```

Beachten Sie, dass Kommentare in der Praxis (so wie in diesem Beispiel) keine expliziten Aussagen darüber enthalten, ob und wenn Ja, um welche Arten von Zusicherungen es sich dabei handelt. Solche Informationen kann man aus dem Kontext herauslesen. Die erste Kommentarzeile kann nur eine Invariante darstellen, da allgemein gültige (das heißt, nicht auf einzelne Methoden eingeschränkte) Beziehungen zwischen Variablen hergestellt werden. Die zweite Kommentarzeile enthält gleich zwei verschiedene Arten von Zusicherungen: Die Aussage „Einzahlen addiert Summe

zu Guthaben“ bezieht sich darauf, wie die Ausführung einer bestimmten Methode den Objektzustand verändert. Das kann nur eine Nachbedingung sein. Nachbedingungen lesen sich häufig wie Beschreibungen dessen, was eine Methode tut. Aber die Aussage „summe ≥ 0“ bezieht sich auf eine erwartete Eigenschaft eines Parameters und ist daher eine Vorbedingung auf `einzahlen`. Mit derselben Begründung ist „Abheben zieht Summe von Guthaben ab“ eine Nachbedingung und sind „summe ≥ 0“ und „guthaben+ueberziehungsrahmen $\geq$ summe“ Vorbedingungen auf `abheben`.

Nebenbei bemerkt sollen Geldbeträge wegen möglicher Rundungsfehler niemals durch Fließkommazahlen (`float` oder `double`) dargestellt werden. Verwenden Sie statt dessen wie in obigem Beispiel ausreichend große ganzzahlige Typen oder noch besser spezielle Typen für Geldbeträge.

Bisher haben wir die Begriffe Typ und Schnittstelle als im Wesentlichen gleichbedeutend angesehen. Ab jetzt betrachten wir Zusicherungen, unabhängig davon, ob sie durch eigene Sprachkonstrukte oder in Kommentaren beschrieben sind, als zum Typ eines Objekts gehörend. Ein Typ besteht demnach aus

- dem Namen einer Klasse, eines Interfaces oder eines einfachen Typs,
- der entsprechenden Schnittstelle
- und den dazugehörenden Zusicherungen.

Der Name sollte eine kurze Beschreibung des Zwecks der Instanzen des Typs geben. Die Schnittstelle enthält alle vom Compiler überprüfbaren Bestandteile des Vertrags zwischen Clients und Server. Zusicherungen enthalten schließlich alle Vertragsbestandteile, die nicht vom Compiler überprüft werden.

In Abschnitt 2.1 haben wir gesehen, dass Typen wegen der besseren Wartbarkeit stabil sein sollen. Solange eine Programmänderung den Typ der Klasse unverändert lässt, oder nur auf unbedenkliche Art und Weise erweitert (siehe Abschnitt 2.2.2), hat die Änderung keine Auswirkungen auf andere Programmteile. Das betrifft auch Zusicherungen. Eine Programmänderung kann sich sehr wohl auf andere Programmteile auswirken, wenn dabei eine Zusicherung geändert wird.

Faustregel: Zusicherungen sollen stabil bleiben. Das ist für Zusicherungen in Typen an der Wurzel der Typhierarchie ganz besonders wichtig.

ProgrammiererInnen können die Genauigkeit der Zusicherungen selbst bestimmen. Dabei sind Auswirkungen der Zusicherungen zu beachten: Clients dürfen sich nur auf das verlassen, was in der Schnittstelle und in den Zusicherungen vom Server zugesagt wird, und der Server auf das, was von den Clients zugesagt wird. Sind die Zusicherungen sehr genau, können sich die Clients auf viele Details des Servers verlassen, und auch der Server kann von den Clients viel verlangen. Aber Programmänderungen werden mit größerer Wahrscheinlichkeit dazu führen, dass Zusicherungen geändert werden müssen, wovon alle Clients betroffen sind. Steht hingegen in den Zusicherungen nur das Nötigste, sind Clients und Server relativ unabhängig voneinander. Der Typ ist bei Programmänderungen eher stabil. Aber vor allem die Clients dürfen sich nur auf Weniges verlassen. Wenn keine Zusicherungen gemacht werden, dürfen sich Clients auf nichts verlassen, was nicht in der Objektschnittstelle steht.

Faustregel: Zur Verbesserung der Wartbarkeit sollen Zusicherungen keine unnötigen Details festlegen.

Zusicherungen bieten umfangreiche Möglichkeiten zur Gestaltung der Client-Server-Beziehungen. Aus Gründen der Wartbarkeit soll man Zusicherungen aber nur dort einsetzen, wo tatsächlich Informationen benötigt werden, die über jene in der Objektschnittstelle hinausgehen. Insbesondere soll man Zusicherungen so einsetzen, dass der Klassenzusammenhalt maximiert und die Objektkopplung minimiert wird. In obigem Konto-Beispiel wäre es wahrscheinlich besser, die Vorbedingung, dass `abheben` den Überziehungsrahmen nicht überschreiten darf, wegzulassen und dafür die Einhaltung der Bedingung direkt in der Implementierung von `abheben` durch eine `if`-Anweisung zu überprüfen. Dann ist nicht mehr der Client für die Einhaltung der Bedingung verantwortlich, sondern der Server.

Faustregel: Alle benötigten Zusicherungen sollen (explizit als Kommentare oder zumindest durch sprechende Namen impliziert) im Programm stehen.

Die Vermeidung unnötiger Zusicherungen zielt darauf ab, dass Client und Server als relativ unabhängig voneinander angesehen werden können. Die Wartbarkeit wird dadurch natürlich nur dann verbessert, wenn diese Unabhängigkeit tatsächlich gegeben ist. Einen äußerst unerwünschten Effekt erzielt man, wenn man Zusicherungen einfach aus Bequemlichkeit

nicht in den Programmcode schreibt, der Client aber trotzdem bestimmte Eigenschaften vom Server erwartet (oder umgekehrt), also beispielsweise implizit voraussetzt, dass eine Einzahlung den Kontostand erhöht. In diesem Fall hat man die Abhängigkeiten zwischen Client und Server nur versteckt. Wegen der Abhängigkeiten können Programmänderungen zu unerwarteten Fehlern führen, die man nur schwer findet, da die Abhängigkeiten nicht offensichtlich sind. Es sollen daher alle Zusicherungen explizit im Programmcode stehen. Andererseits sollen Client und Server aber so unabhängig wie möglich bleiben.

Sprechende Namen sagen viel darüber aus, wofür Typen und Methoden gedacht sind. Namen implizieren damit die wichtigsten Zusicherungen. Beispielsweise wird eine Methode `insert` in einer Instanz von `Set` ein Element zu einer Menge hinzufügen. Darauf werden sich Clients verlassen, auch wenn dieses Verhalten nicht durch explizite Kommentare spezifiziert ist. Trotzdem ist es gut, wenn das Verhalten zusätzlich als Kommentar beschrieben ist, da Kommentare den Detailiertheitsgrad viel besser angeben können als aus den Namen hervorgeht. Kommentare und Namen müssen in Einklang zueinander stehen.

2.2.2 Untertypen und Verhalten

Zusicherungen, die zu Typen gehören, müssen auch bei der Verwendung von enthaltendem Polymorphismus beachtet werden. Auch für Zusicherungen gilt das Ersetzbarkeitsprinzip bei der Feststellung, ob ein Typ Untertyp eines anderen Typs ist. Neben den Bedingungen, die wir in Abschnitt 2.1 kennen gelernt haben, müssen folgende Bedingungen gelten, damit ein Typ U Untertyp eines Typs T ist [17]:

- Jede Vorbedingung auf einer Methode in T muss eine Vorbedingung auf der entsprechenden Methode in U implizieren. Das heißt, Vorbedingungen in Untertypen können schwächer, dürfen aber nicht stärker sein als entsprechende Vorbedingungen in Obertypen. Der Grund liegt darin, dass ein Aufrufer der Methode, der nur T kennt, nur die Erfüllung der Vorbedingungen in T sicherstellen kann, auch wenn die Methode tatsächlich in U statt T aufgerufen wird. Daher muss die Vorbedingung in U automatisch erfüllt sein, wenn sie in T erfüllt ist. Wenn Vorbedingungen in U aus T übernommen werden, können sie mittels Oder-Verknüpfungen schwächer werden. Ist die Vorbedingung in T zum Beispiel „$x > 0$“, kann die Vorbedingung in U auch „$x > 0$ oder $x = 0$“, also abgekürzt „$x \geq 0$“ lauten.

- Jede Nachbedingung auf einer Methode in U muss eine Nachbedingung auf der entsprechenden Methode in T implizieren. Das heißt, Nachbedingungen in Untertypen können stärker, dürfen aber nicht schwächer sein als entsprechende Nachbedingungen in Obertypen. Der Grund liegt darin, dass ein Aufrufer der Methode, der nur T kennt, sich auf die Erfüllung der Nachbedingungen in T verlassen kann, auch wenn die Methode tatsächlich in U statt T aufgerufen wird. Daher muss eine Nachbedingung in T automatisch erfüllt sein, wenn ihre Entsprechung in U erfüllt ist. Wenn Nachbedingungen in U aus T übernommen werden, können sie mittels Und-Verknüpfungen stärker werden. Lautet die Nachbedingung in T beispielsweise „`result` $>$ 0“, kann sie in U auch „`result` > 0 und `result` > 2“, also „`result` $>$ 2“ sein.

- Jede Invariante in U muss eine Invariante in T implizieren. Das heißt, Invarianten in Untertypen können stärker, dürfen aber nicht schwächer sein als Invarianten in Obertypen. Der Grund liegt darin, dass ein Client, der nur T kennt, sich auf die Erfüllung der Invarianten in T verlassen kann, auch wenn tatsächlich eine Instanz von U statt einer von T verwendet wird. Der Server kennt seinen eigenen spezifischsten Typ, weshalb das Ersetzbarkeitsprinzip aus der Sicht des Servers nicht erfüllt zu sein braucht. Die Invariante in T muss automatisch erfüllt sein, wenn sie in U erfüllt ist. Wenn Invarianten in U aus T übernommen werden, können sie, wie Nachbedingungen, mittels Und-Verknüpfungen stärker werden. Dieser Zusammenhang mit Nachbedingungen ist notwendig, da Invarianten entsprechende Nachbedingungen auf allen Methoden des Typs implizieren.

  Diese Erklärung geht davon aus, dass Instanzvariablen nicht durch andere Objekte verändert werden. Ist dies doch der Fall, so müssen Invarianten, die sich auf global änderbare Variablen beziehen, in U und T übereinstimmen. Beim Schreiben einer solchen Variablen muss die Invariante vom Client überprüft werden, was dem generellen Konzept widerspricht. Außerdem kann ein Client die Invariante gar nicht überprüfen, wenn in der Bedingung vorkommende Variablen und Methoden nicht öffentlich zugänglich sind. Daher sollen Instanzvariablen möglichst selten oder nie durch andere Objekte verändert werden.

Im Prinzip lassen sich obige Bedingungen auch formal überprüfen. In Programmiersprachen wie Eiffel, in denen Zusicherungen formal definiert

sind, wird das tatsächlich gemacht. Aber bei Verwendung anderer Programmiersprachen sind Zusicherungen meist nicht formal, sondern nur umgangssprachlich als Kommentare gegeben. Unter diesen Umständen ist natürlich keine formale Überprüfung möglich. Daher müssen die ProgrammiererInnen alle nötigen Überprüfungen per Hand durchführen. Im Einzelnen muss sichergestellt werden, dass

- obige Bedingungen für Untertypbeziehungen eingehalten werden,
- die Implementierungen der Server die Nachbedingungen und Invarianten erfüllen und nichts voraussetzen, was nicht durch Vorbedingungen oder Invarianten festgelegt ist
- und Clients die Vorbedingungen der Aufrufe erfüllen und nichts voraussetzen, was nicht in Nachbedingungen und Invarianten vom Server zugesichert wird.

Es kann sehr aufwändig sein, alle solchen Überprüfungen vorzunehmen. Einfacher geht es, wenn ProgrammiererInnen während der Codeerstellung und bei Änderungen stets an die einzuhaltenden Bedingungen denken, die Überprüfungen also nebenbei erfolgen. Wichtig ist dabei darauf zu achten, dass die Zusicherungen unmissverständlich formuliert sind. Nach Änderung einer Zusicherung ist die Überprüfung besonders schwierig, und die Änderung einer Zusicherung ohne gleichzeitige Änderung *aller* betroffenen Programmteile ist eine häufige Fehlerursachen in Programmen.

> **Faustregel:** Zusicherungen sollen unmissverständlich formuliert sein und während der Programmentwicklung und Wartung ständig bedacht werden.

Betrachten wir ein Beispiel für einen Typ beziehungsweise eine Klasse mit Zusicherungen in Form von Kommentaren:

```
class Set
{
public:
    // inserts x into set iff not already there;
    // x is in set immediately after invocation
    virtual void insert (int x)
    {
        ...;
    }
    // returns true if x is in set, otherwise false
    virtual bool inSet (int x)
    {
        ...;
```

```
    }
    virtual ~Set() {}
};
```

Die Methode `insert` fügt eine ganze Zahl genau dann („iff“ ist eine übliche Abkürzung für „if and only if“, also „genau dann wenn“) in eine Instanz von `Set` ein, wenn sie noch nicht in dieser Menge ist. Unmittelbar nach Aufruf der Methode ist die Zahl in jedem Fall in der Menge. Die Methode `inSet` stellt fest, ob eine Zahl in der Menge ist oder nicht. Dieses Verhalten der Instanzen von `Set` ist durch die Zusicherungen in den Kommentaren festgelegt. Wenn man den Inhalt dieser Beschreibungen von Methoden genauer betrachtet, sieht man, dass es sich dabei um Nachbedingungen handelt. Da Nachbedingungen beschreiben, was sich ein Client vom Aufruf einer Methode erwartet, sind Nachbedingungen oft tatsächlich nur Beschreibungen von Methoden.

Folgende Klasse unterscheidet sich von `Set` nur durch eine zusätzliche Invariante:

```
// elements in the set always remain in the set
class SetWithoutDelete : public Set
{
};
```

Die Invariante besagt, dass eine Zahl, die einmal in der Menge war, stets in der Menge bleibt. Offensichtlich ist `SetWithoutDelete` ein Untertyp von `Set`, da nur eine Invariante dazugefügt wurde, die Invarianten insgesamt also strenger wurden. Wie kann ein Client eine solche Invariante nutzen? Sehen wir uns dazu eine kurze Codesequenz für einen Client an:

```
void foo (Set &s)
{
    s.insert(41);
    doSomething(s);
    if (s.inSet(41)) { doSomeOtherThing(s); }
    else { doSomethingElse(); }
}
```

Während der Ausführung von `doSomething` könnte `s` verändert werden. Es ist nicht ausgeschlossen, dass 41 dabei aus der Menge gelöscht wird, da die Nachbedingung von `insert` in `Set` ja nur zusichert, dass 41 unmittelbar nach dem Aufruf von `insert` in der Menge ist. Bevor wir die Methode `doSomeOtherThing` aufrufen (von der wir annehmen, dass sie ihren Zweck nur erfüllt, wenn 41 in der Menge ist), stellen wir sicher, dass 41 tatsächlich in der Menge ist. Dies geschieht durch Aufruf von `inSet`.

Verwenden wir eine Instanz von `SetWithoutDelete` anstatt einer von `Set`, ersparen wir uns den Aufruf von `inSet`. Wegen der stärkeren Zusi-

cherung ist 41 sicher in der Menge:

```
void bar (SetWithoutDelete &s)
{
    s.insert(41);
    doSomething(s);
    doSomeOtherThing(s); // s.inSet(41) returns true
}
```

Von diesem kleinen Vorteil von `SetWithoutDelete` für Clients darf man sich nicht dazu verleiten lassen, generell starke Nachbedingungen und Invarianten zu verwenden. Solche umfangreichen Zusicherungen können die Wartung erschweren (siehe Abschnitt 2.2.1). Zum Beispiel können wir `Set` problemlos um eine Methode `del` (zum Löschen einer Zahl aus der Menge) erweitern:

```
class SetWithDelete : public Set
{
public:
    // deletes x from the set if it is there
    void del (int x)
    {
        ...;
    }
};
```

Aber `SetWithoutDelete` können wir, wie der Klassenname schon sagt, nicht um eine solche Methode erweitern. Zwar wäre eine derart erweiterte Klasse mit obigen Bedingungen für Zusicherungen bei Untertypbeziehungen vereinbar, aber die Nachbedingung von `del` steht in Konflikt zur Invariante. Es wäre also unmöglich, `del` so zu implementieren, dass sowohl die Nachbedingung als auch die Invariante erfüllt ist. Man darf nicht zu früh festlegen, dass es kein `del` gibt, nur weil man es gerade nicht braucht. Invarianten wie in `SetWithoutDelete` soll man nur verwenden, wenn man sie wirklich braucht. Andernfalls verbaut man sich Wiederverwendungsmöglichkeiten.

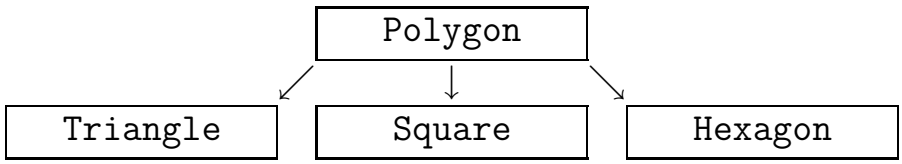
In Sprachen wie Modula-3 [21] gibt es Typen, bei denen Untertypbeziehungen angenommen werden, sobald die statisch prüfbaren Bedingungen aus Abschnitt 2.1.1 erfüllt sind. Dafür sind keine Untertypdeklarationen oder Vererbungsbeziehungen nötig. Kommentare als Zusicherungen setzen jedoch voraus, dass Untertypbeziehungen explizit deklariert werden. Damit bringt man den Compiler dazu, beliebige weitere Bedingungen für eine Untertypbeziehung vorauszusetzen. Beispielsweise muss man explizit angeben, dass `SetWithoutDelete` ein Untertyp von `Set` ist, da sich diese Klassen für einen Compiler nur im Namen und in Kommentaren unterscheiden, deren Bedeutung der Compiler nicht kennt. Andern-

falls könnte eine Instanz von `Set` auch verwendet werden, wo eine von `SetWithoutDelete` erwartet wird. Es soll auch keine Untertypbeziehung zwischen `SetWithoutDelete` und `SetWithDelete` bestehen, obwohl dafür alle Bedingungen aus Abschnitt 2.1.1 erfüllt sind. Sonst wäre eine Instanz von `SetWithDelete` verwendbar, wo eine Instanz von `SetWithoutDelete` erwartet wird. Daher sind in vielen objektorientierten Sprachen enthaltender Polymorphismus und Vererbung zu einem Konstrukt vereint: Explizite Vererbungsbeziehungen schließen zufällige Untertypbeziehungen aus, und wo eine Untertypbeziehung besteht ist oft auch Codevererbung sinnvoll.

2.2.3 Abstrakte Klassen

Klassen, die wir bis jetzt betrachtet haben, dienen der Beschreibung der Struktur ihrer Instanzen, der Erzeugung und Initialisierung neuer Instanzen und der Festlegung des spezifischsten Typs der Instanzen. Im Zusammenhang mit enthaltendem Polymorphismus benötigt man oft nur eine der Aufgaben, nämlich die Festlegung des Typs. Das ist dann der Fall, wenn im Programm keine Instanzen der Klasse selbst erzeugt werden sollen, sondern nur Instanzen von Unterklassen. Aus diesem Grund unterstützen viele objektorientierte Sprachen *abstrakte Klassen*, von denen keine Instanzen erzeugt werden können.

Nehmen wir als Beispiel folgende Klassenstruktur:



Jede Unterklasse von `Polygon` beschreibt ein z. B. am Bildschirm darstellbares Vieleck mit einer bestimmten Anzahl von Ecken. `Polygon` selbst beschreibt keine bestimmte Anzahl von Ecken, sondern fasst nur die Menge aller möglichen Vielecke zusammen. Wenn man eine Liste unterschiedlicher Vielecke benötigt, wird man den Typ der Vielecke in der Liste mit `Polygon` festlegen, obwohl in der Liste tatsächlich nur Instanzen von `Triangle`, `Square` und `Hexagon` vorkommen. Es werden keine Instanzen der Klasse `Polygon` selbst benötigt, sondern nur Instanzen der Unterklassen. `Polygon` ist ein typischer Fall einer abstrakten Klasse.

In C++ sieht eine abstrakte Klasse beispielsweise so aus:

```
class Polygon
{
    // ...
public:
```

```
    // draw a polygon on the screen
    virtual void draw() = 0;
    virtual ~Polygon() = 0;
    // ...
};
```

Ist eine abstrakte Klasse erwünscht, welche keine abstrakte Methode anbietet, so kann der Destruktor rein virtuell deklariert werden. Es ist aber darauf zu achten, dass auch eine Definition zu dem rein virtuellen Destruktor existiert:

```
Polygon::~Polygon() {}
```

Zur Wiederholung: Grundsätzlich ist immer darauf zu achten, dass ein virtueller Destruktor existiert, wenn irgendeine Methode virtuell ist und Polymorphismus erwünscht ist. [Mey05]

Da obige Klasse `Polygon` abstrakt ist, ist es nicht möglich, diese Klasse zu instanzieren. Der Compiler gibt eine entsprechende Fehlermeldung aus. Alle drei folgenden Zeilen werden somit nicht akzeptiert:

```
void wont_compile()
{
    Polygon p1;
    Polygon &p1 = Polygon();
    Polygon *p1 = new Polygon();
}
```

Aber es ist problemlos möglich, einen Zeiger oder Referenz auf ein `Polygon` zu verwenden:

```
void is_ok1(Polygon& p) { }
void is_ok2(Polygon* p) { }
```

Und man kann auch Unterklassen von `Polygon` ableiten, und diesen Funktionen dann eine Instanz von einem Untertyp übergeben. Jede Unterklasse muss eine Methode `draw` enthalten, da diese Methode in `Polygon` deklariert ist. Genaugenommen ist `draw` als abstrakte Methode deklariert; das heißt, es ist keine Implementierung von `draw` angegeben, sondern nur dessen Schnittstelle mit einer kurzen Beschreibung – einer Zusicherung als Kommentar. In abstrakten Klassen brauchen wir keine Implementierungen für Methoden angeben, da die Methoden ohnehin nicht ausgeführt werden (mit Ausnahme des Destruktors) ; es gibt ja keine Instanzen. Nichtabstrakte Unterklassen – das sind *konkrete Klassen* – müssen Implementierungen für abstrakte Methoden bereitstellen, diese also überschreiben. Wird eine rein virtuelle Methode nicht überschrieben, so sind auch die Unterklassen abstrakt. Neben abstrakten Methoden dürfen abstrakte Klassen auch konkrete (also implementierte) Methoden enthalten, die wie üblich vererbt werden.

Die konkrete Klasse `Triangle` könnte so aussehen:

```
class Triangle : public Polygon
{
public:
    // draw a triangle on the screen
    void draw();
};
```

Wie bereits erwähnt ist hier kein `virtual` notwendig, da die Oberklasse bereits ein virtuelles `draw` ohne Parameter deklariert hat. Es können aber – aus Gründen der Lesbarkeit – trotzdem die überschriebenen Methoden `virtual` deklariert werden. Die Implementierung erfolgt wie üblich in einer Source Datei:

```
void Triangle::draw()
{
    // ...
}
```

Auch `Square` und `Hexagon` müssen die Methode `draw` implementieren.

So wie in diesem Beispiel kommt es vor allem in gut faktorisierten Programmen häufig vor, dass der Großteil der Implementierungen von Methoden in Klassen steht, die keine Unterklassen haben. Abstrakte Klassen, die keine Implementierungen enthalten, sind eher stabil als andere Klassen. Zur Verbesserung der Wartbarkeit soll man vor allem von stabilen Klassen erben. Außerdem soll man möglichst stabile Typen für formale Parameter und Variablen verwenden. Da es oft leichter ist, abstrakte Klassen ohne Implementierungen stabil zu halten, ist man gut beraten, hauptsächlich solche Klassen für Parameter- und Variablentypen zu verwenden.

> **Faustregel:** Es ist empfehlenswert, als Obertypen und Parametertypen hauptsächlich Interfaces zu verwenden.

Vor allem Parametertypen sollen keine Bedingungen an Argumente stellen, die nicht benötigt werden. Konkrete Klassen legen aber oft zahlreiche Bedingungen in Form von Zusicherungen und Methoden in der Schnittstelle fest. Diesen Konflikt kann man leicht lösen, indem man für die Typen der Parameter nur abstrakte Klassen verwendet. Es ist ja leicht, zu jeder konkreten Klasse eine oder mehrere abstrakte Klassen als Oberklassen zu schreiben, die die benötigten Bedingungen möglichst genau angeben. Damit werden unnötige Abhängigkeiten vermieden.

Wenn eine abstrakte Klasse ausschließlich rein virtuelle Methoden hat, nennt man sie auch *Interface*. Diese Konvention wird hier deshalb verwen-

det, weil diese Klasse ausschließlich aus einer Schnittstelle, ohne Implementationen, besteht.

2.3 Vererbung versus Ersetzbarkeit

Vererbung ist im Grunde sehr einfach: Von einer Oberklasse wird scheinbar, aber meist nicht wirklich, eine Kopie angelegt, die entsprechend den Wünschen der ProgrammiererInnen durch Erweitern und Überschreiben abgeändert wird. Die resultierende Klasse ist die Unterklasse. Wenn man nur Vererbung betrachtet und Einschränkungen durch enthaltenden Polymorphismus ignoriert, haben ProgrammiererInnen vollkommene Freiheit in der Abänderung der Oberklasse. Vererbung ist zur direkten Wiederverwendung von Code einsetzbar und damit auch unabhängig vom Ersetzbarkeitsprinzip sinnvoll. Wir wollen zunächst einige Arten von Beziehungen zwischen Klassen unterscheiden lernen und dann die Bedeutungen dieser Beziehungen für die Codewiederverwendung untersuchen.

2.3.1 Reale Welt versus Vererbung versus Ersetzbarkeit

In der objektorientierten Softwareentwicklung begegnen wir zumindest drei verschiedenen Arten von Beziehungen zwischen Klassen [15]:

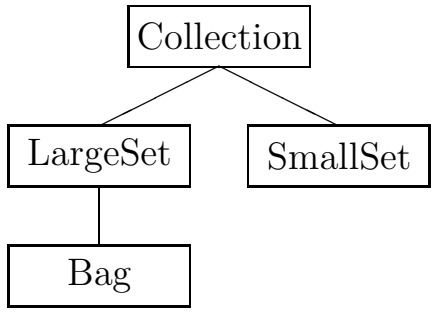
Untertypbeziehung: Diese Beziehung, die auf dem Ersetzbarkeitsprinzip beruht, haben wir bereits untersucht.

Vererbungsbeziehung: Dabei handelt es sich um eine Beziehung zwischen Klassen, bei der eine Klasse durch Abänderung einer anderen Klasse entsteht. Es ist nicht nötig, aber wünschenswert, dass dabei Code aus der Oberklasse in der Unterklasse direkt wiederverwendet wird. Für eine reine Vererbungsbeziehung ist das Ersetzbarkeitsprinzip irrelevant. In C++ gibt es auch ein Sprachmittel, um direkt auszudrücken, dass Ersetzbarkeit nicht angestrebt wird: `private` und `protected` Vererbung. Dann stellt der Compiler sicher, dass nicht versucht wird ein polymorphes Verhalten zu bekommen, indem keine abgeleitete Klassen, die dann keine Untertypen sind, statt einer Oberklasse verwendet werden dürfen. Diese Beziehung nennt man auch *ist-implementiert-mit*. In Situationen wo kein Zugriff auf geschützte Elemente benötigt wird, kann stattdessen auch Komposition verwendet werden.

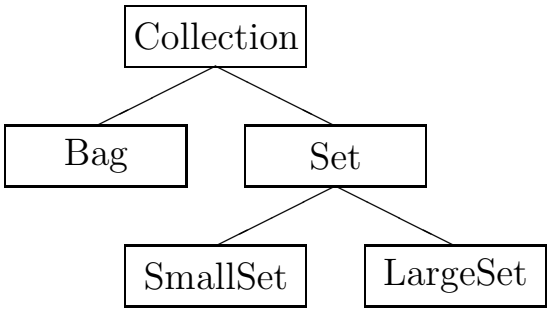
Reale-Welt-Beziehung: In der Analysephase und zu Beginn der Entwurfsphase haben sich oft schon abstrakte Einheiten herauskristallisiert, die in späteren Phasen zu Klassen weiterentwickelt werden. Auch Beziehungen zwischen diesen Einheiten existieren bereits sehr früh. Sie spiegeln angenommene *ist-ein-Beziehungen* („is a“) in der realen Welt wider. Zum Beispiel haben wir die Beziehung „ein Studierender ist eine Person“, wobei „Studierender“ und „Person“ abstrakte Einheiten sind, die später voraussichtlich zu Klassen weiterentwickelt werden. Durch die Simulation der realen Welt sind solche Beziehungen bereits sehr früh intuitiv klar, obwohl die genauen Eigenschaften der Einheiten noch gar nicht feststehen. Normalerweise entwickeln sich diese Beziehungen während des Entwurfs zu (vor allem) Untertyp- und (gelegentlich) Vererbungsbeziehungen zwischen Klassen weiter. Es kann sich aber auch herausstellen, dass Details der Klassen dem Ersetzbarkeitsprinzip widersprechen und nur `private` Vererbung einsetzbar ist. In solchen Fällen wird es zu Refaktorisierungen kommen, die in dieser Phase einfach durchführbar sind.

Beziehungen in der realen Welt verlieren stark an Bedeutung, sobald genug Details bekannt sind, um sie zu Untertyp- und Vererbungsbeziehungen weiterzuentwickeln. Deshalb konzentrieren wir uns hier nur auf die Unterscheidung zwischen Untertyp- und Vererbungsbeziehungen. Genaugenommen setzen Untertypbeziehungen, zumindest in C++ und ähnlichen objektorientierten Sprachen, Vererbungsbeziehungen voraus und sind derart eingeschränkt, dass die vom Compiler überprüfbaren Bedingungen für Untertypbeziehungen stets erfüllt sind. Das heißt, das wesentliche Unterscheidungskriterium ist das, ob die Zusicherungen zwischen Unter- und Oberklasse kompatibel sind. Diese Unterscheidung können nur SoftwareentwicklerInnen treffen, die Bedeutungen von Namen und Kommentaren verstehen. Wie wir sehen werden, gibt es in C++ auch Möglichkeiten mit Vererbungsbeziehungen andere Arten als Untertypbeziehungen darzustellen.

Man kann leicht erkennen, ob EntwicklerInnen reine Vererbungs- oder Untertypbeziehungen anstreben. Betrachten wir dazu ein Beispiel:



reine Vererbungsbeziehung



Untertypbeziehung

Es ist das Ziel der reinen Vererbung, so viele Teile der Oberklasse wie möglich direkt in der Unterklasse wiederzuverwenden. Angenommen, die Implementierungen von `LargeSet` und `Bag` zeigen so starke Ähnlichkeiten, dass sich die Wiederverwendung von Programmteilen lohnt. In diesem Fall erbt `Bag` große Teile der Implementierung von `LargeSet`. Für diese Entscheidung ist nur der pragmatische Gesichtspunkt, dass sich `Bag` einfacher aus `LargeSet` ableiten lässt als umgekehrt, ausschlaggebend. Für `SmallSet` wurde eine von `LargeSet` unabhängige Implementierung gewählt, die bei kleinen Mengen effizienter ist als `LargeSet`.

Wenn wir uns von Konzepten beziehungsweise Typen leiten lassen, schaut die Hierarchie anders aus. Wir führen eine zusätzliche (abstrakte) Klasse `Set` ein, da die Typen von `LargeSet` und `SmallSet` dieselbe Bedeutung haben sollen. Wir wollen im Programmcode nur selten zwischen `LargeSet` und `SmallSet` unterscheiden. `Bag` und `LargeSet` stehen in keinem Verhältnis zueinander, da die Methoden für das Hinzufügen von Elementen einander ausschließende Bedeutungen haben, obwohl `Set` und `Bag` dieselbe Schnittstelle haben können. Einander ausschließende Bedeutungen kommen daher, dass eine Instanz von `Set` höchstens ein Vorkommen eines Objekts enthalten kann, während in einer Instanz von `Bag` mehrere Vorkommen erlaubt sind. Entsprechend darf eine Methode nur dann ein Element zu einer Instanz von `Set` hinzufügen, wenn das Element noch nicht vorkommt, während die Methode zum Hinzufügen in eine Instanz von `Bag` jedes gewünschte Element akzeptieren muss.

Obiges Beispiel demonstriert unterschiedliche Argumentationen für die reine Vererbung im Vergleich zu Untertypbeziehungen. Die Unterschiede zwischen den Argumentationen sind wichtiger als jene zwischen den Hierarchien, da die Hierarchien selbst letztendlich von Details und beabsichtigten Verwendungen abhängen.

Tipp: Verwende nur dann `public` Vererbung, wenn eine Untertypbeziehung angestrebt wird.

2.3.2 Vererbung und Codewiederverwendung

Manchmal kann man durch reine Vererbungsbeziehungen, die Untertypbeziehungen unberücksichtigt lassen, einen höheren Grad an direkter Codewiederverwendung erreichen als wenn man bei der Softwareentwicklung Untertypbeziehungen anstrebt. Natürlich möchten wir einen möglichst hohen Grad an Codewiederverwendung erzielen. Ist es daher günstig, Untertypbeziehungen unberücksichtigt zu lassen? Diese Frage muss man ganz klar mit Nein beantworten. Durch die Nichtbeachtung des Ersetzbarkeitsprinzips – das heißt, Untertypbeziehungen sind nicht gegeben – ist es nicht mehr möglich, eine Instanz eines Untertyps zu verwenden, wo eine Instanz eines Obertyps erwartet wird. Wenn man trotzdem eine Instanz einer Unterklasse statt der einer Oberklasse verwendet, kann ein Programmfehler auftreten. Genau das wird durch `private` Vererbung verhindert. Verzichtet man auf Ersetzbarkeit, wird die Wartung erschwert, da sich fast jede noch so kleine Programmänderung auf das ganze Programm auswirken kann. Viele Vorteile der objektorientierten Programmierung gehen damit verloren. Unter Umständen gewinnt man zwar durch die reine Vererbung bessere direkte Codewiederverwendung in kleinem Umfang, tauscht diese aber gegen viele Möglichkeiten für die indirekte Codewiederverwendung in großem Umfang, die nur durch die Ersetzbarkeit gegeben sind.

Faustregel: Wiederverwendung durch das Ersetzbarkeitsprinzip ist wesentlich wichtiger als direkte Wiederverwendung durch Vererbung.

Der allgemeine Ratschlag ist daher ganz klar: Ein wichtiges Ziel ist die Entwicklung geeigneter Untertypbeziehungen. Vererbungsbeziehungen sind nur Mittel zum Zweck; das heißt, sie sollen sich den Untertypbeziehungen unterordnen. Im Allgemeinen soll es im Programm keine Vererbungsbeziehung geben, die nicht auch eine Untertypbeziehung ist, bei der also alle Zusicherungen kompatibel sind.

Wie die Erfahrung zeigt, vergessen Anfänger in der objektorientierten Programmierung allzu leicht das Ersetzbarkeitsprinzip und konzentrieren sich ganz und gar auf direkte Codewiederverwendung durch Vererbung.

Daher soll noch einmal klar gesagt werden, dass die Menge des aus einer Oberklasse ererbten Codes für die Codewiederverwendung nur sehr geringe Bedeutung hat. Viel wichtiger für die Wiederverwendung ist das Bestehen von Untertypbeziehungen.

Man soll aber nicht gleich von vornherein auf direkte Codewiederverwendung durch Vererbung verzichten. In vielen Fällen lässt sich auch dann ein hoher Grad an direkter Codewiederverwendung erzielen, wenn das Hauptaugenmerk auf Untertypbeziehungen liegt. In obigem Beispiel gibt es vielleicht Programmcode, der sowohl in der Klasse `SmallSet` als auch in `LargeSet` vorkommt. Entsprechende Methoden kann man bereits in der abstrakten Klasse `Set` implementieren, von der `SmallSet` und `LargeSet` erben. Vielleicht gibt es sogar Methoden, die in `Set` und `Bag` gleich sind und in `Collection` implementiert werden können.

Direkte Codewiederverwendung durch Vererbung erspart ProgrammiererInnen nicht nur das wiederholte Schreiben desselben Codes, sondern hat auch Auswirkungen auf die Wartbarkeit. Wenn ein Programmteil nur einmal statt mehrmals implementiert ist, brauchen Änderungen nur an einer einzigen Stelle vorgenommen werden, wirken sich aber auf alle Programmteile aus, in denen der veränderte Code verwendet wird. Nicht selten muss man alle gleichen oder ähnlichen Programmteile gleichzeitig ändern, wenn sich die Anforderungen ändern. Gerade dabei kann Vererbung sehr hilfreich sein.

> **Faustregel:** Auch reine Vererbung kann sich positiv auf die Wartbarkeit auswirken.

Es kommt vor, dass nicht alle solchen Programmteile geändert werden sollen, sondern nur einer oder einige wenige. Dann ist es nicht möglich, eine Methode unverändert zu erben. Glücklicherweise ist es in diesem Fall sehr einfach, eine geerbte Methode durch eine neue Methode zu überschreiben. In Sprachen wie C++ ist es sogar möglich, die Methode zu überschreiben und trotzdem noch auf die überschriebene Methode in der Oberklasse zuzugreifen. Ein Beispiel soll das demonstrieren:

```
class A
{
public:
    virtual void foo() { ... }
};
class B : public A
{
    bool m_b;
public:
```

```
    void foo()
    {
        if (m_b) { ... }
        else { A::foo(); }
    }
    virtual ~A() {}
};
```

Der Programmcode in `A` ist trotz Überschreibens auch in `B` verwendbar. Diese Art des Zugriffs auf Oberklassen funktioniert über mehrere Vererbungsebenen hinweg.

In komplizierten Situationen ist geschickte Faktorisierung notwendig, um direkte Codewiederverwendung zu erreichen:

```
class A
{
public:
    virtual void foo()
    {
        if (...) { ... }
        else { ...; x = 1; ...}
    }
    virtual ~A() {}
};
class B : public A
{
public:
    void foo()
    {
        if (...) { ... }
        else { ...; x = 2; ...}
    }
};
```

Die Methode `foo` muss gänzlich neu geschrieben werden, obwohl der Unterschied minimal ist. Eine Aufspaltung von `foo` kann helfen:

```
class A
{
    virtual void fooX() { ...; x = 1; ... }
public:
    void foo()
    {
        if (...) { ... }
        else { fooX(); }
    }
    virtual ~A() {}
};
class B : public A
{
    void fooX() { ...; x = 2; ... }
};
```

Das ist eine Anwendung der Template Method (siehe Abschnitt 4.3). Man braucht nur mehr einen Teil der Methode zu überschreiben. Solche Techni-

ken setzen aber voraus, dass man bereits beim Schreiben der Klasse `A` sehr klare Vorstellungen davon hat, welche Teile später überschrieben werden müssen. Direkte Code-Wiederverwendung ergibt sich also nicht automatisch oder zufällig, sondern in der Regel nur dort, wo Vererbung gezielt eingeplant wurde.

Unterschiede zwischen Unter- und Oberklasse kann man auch durch zusätzliche Parameter beschreiben und nach außen sichtbare Methoden nur zum Setzen der Parameter verwenden:

```
class A
{
protected:
    void fooY (int y)
    {
        if (...) { ... }
        else { ...; x=y; ... }
    }
public:
    virtual void foo() { fooY(1); }
    virtual ~A() {}
};
class B : public A
{
public:
    void foo() { fooY(2); }
};
```

Der Code von `fooY` wird von `B` zur Gänze geerbt. Die überschriebene Methode `foo` braucht nur ein Argument an `fooY` zu übergeben.

C++ bietet mehr Flexibilität bei der Vererbung als reine Untertypenbeziehungen abzubilden. Obwohl diese Form die wichtigste ist, kann reine Codewiederverwendung wie oben gezeigt nützlich sein. In C++ ist das Konzept, um direkte Codewiederverwendung darzustellen die *private Vererbung*. Hier ist klar, dass keine Ersetzbarkeit gegeben ist und der Compiler verbietet, dass eine Instanz einer Unterklasse, wo eine Instanz einer Oberklasse erwartet wird, verwendet wird. Diese Möglichkeit kann direkte Wiederverwendung von Code genauso verbessern wie die indirekte Wiederverwendbarkeit. Hier ein Beispiel wie die weiter oben skizzierte reine Vererbungsbeziehung aussehen würde:

```
class Collection { /* Common subroutines */ };
class LargeSet :private Collection { };
class SmallSet :private Collection { };
class Bag :private LargeSet { };
```

Technisch gesehen setzt `private` Vererbung alle vererbten Methoden auf `private`. Damit ist es in `Bag` nicht möglich auf Routinen von `Collection` zuzugreifen. Ist dies erwünscht, so kann stattdessen `protected` verwendet

werden. Hier werden alle `public` und `protected` Elemente auf `protected` gesetzt (siehe zweite Tabelle im Abschnitt 2.4.4), und sind somit auch noch in der späteren Vererbungshierarchie nutzbar. Zusätzlich anzumerken ist, dass mit der Art der Vererbung die Rechte immer nur reduziert werden können, d.h. auch mit `public` Vererbung bleiben private Elemente privat. Das Rechte nur von einer Klasse aus vergeben werden können, und sich niemals genommen werden können, ist ein wichtiges Merkmal der Objektorientierten Programmierung in C++.

(für Interessierte)

In der Sprache Sather (siehe `http://www.icsi.berkeley.edu/~sather/`) gibt es zwei komplett voneinander getrennte Hierarchien auf Klassen: die Vererbungshierarchie (für direkte Codewiederverwendung) und die Typhierarchie (für indirekte Codewiederverwendung). Da die Vererbungshierarchie nicht den Einschränkungen des Ersetzbarkeitsprinzips unterliegt, gibt es zahlreiche Möglichkeiten der Codeveränderung bei der Vererbung, beispielsweise die Umbenennung ererbter Routinen und Variablen:

```
class A is          -- Definition einer Klasse A
    ...;            -- Routinen und Variablen von A
end;
class B is          -- Definition einer Klasse B
    include A       -- B erbt von A
      a->b,         -- wobei a aus A in B b heisst
      c->,          -- und c aus A in B nicht verwendbar ist
      d->private d; -- und d aus A in B private ist
    ...;            -- Routinen und Variablen von B
end;
```

Neben den konkreten Klassen gibt es in Sather (wie in C++) auch abstrakte Klassen. Deren Namen müssen mit `$` beginnen:

```
abstract class $X is ...; end;
```

Abstrakte Klassen spielen in Sather eine ganz besondere Rolle, da nur sie als Obertypen in Untertypdeklarationen verwendbar sind:

```
abstract class $Y < $X is ...; end;  -- $Y ist Untertyp von $X
class C < $Y, $Z is ...; end;  -- C ist Untertyp von $Y und $Z
```

Damit sind Instanzen von `C` überall verwendbar, wo Instanzen von `$X`, `$Y` oder `$Z` erwartet werden. Anders als `extends` in C++ bedeutet `<` in Sather jedoch nicht, dass die Unterklasse von der Oberklasse erbt, sondern nur, dass der Compiler die statisch überprüfbaren Bedingungen für eine Untertyprelation prüft und dynamisches Binden ermöglicht. Für Vererbung ist eine separate `include`-Klausel notwendig.

2.4 Exkurs: Klassen und Vererbung in C++

In den vorhergehenden Abschnitten haben wir einige wichtige Konzepte in objektorientierten Sprachen betrachtet. In diesem Abschnitt geben wir einen Überblick über die konkrete Umsetzung in die Programmiersprache C++ . Dieser Abschnitt dient dazu, häufige Unklarheiten und Missverständnisse bezüglich C++ bei Anfängern und Umsteigern zu beseitigen und auf empfohlene Verwendungen einiger C++ -spezifischer Sprachkonstrukte hinzuweisen. Erfahrene C++ -Programmierer mögen verzeihen, dass es zur Erreichung dieses Ziels in einigen Bereichen notwendig ist, scheinbar ganz triviale Sprachkonstrukte zu erklären. Anfänger seien darauf hingewiesen, dass dieser Abschnitt nicht ausreicht, um C++ von Grund auf zu erlernen; es geht nur um die Klärung von Missverständnissen.

2.4.1 Speicherplatzverwaltung

Es gibt drei fundamentale Arten, Speicherplatz in C++ zu verwenden [Str00]:

- *Statischer Speicherplatz*, welcher für die Dauer des Programmes angelegt wird. Das betrifft alle globalen, im Namensbereichen gebundenen Variablen sowie statische Variablen. Dieser Speicherbereich wird bereits vor dem eigentlichen Eintreten in die `main()` Funktion angelegt und mit 0 initialisiert.

- *Automatischer Speicherplatz*, welcher während der Ausführung eines Blockes am Stack angelegt wird. Das betrifft lokale Variablen und Funktionsargumente. Jeder Einstieg in ein Block bekommt seine eigene Kopie dieser Daten. Diese Art von Speicher wird automatisch erzeugt und wieder zerstört. Daraus folgt direkt der Begriff Gültigkeitsbereich (Scope):

  ```
  {
      int i = 1;
      { // i wird hier verdeckt
          int i = 2;
          cout << "innen␣" << i << endl;
      } // inneres i wird freigegeben
      cout << "auesseres␣i␣:␣" << i << endl;
  } // auesseres i wird freigegeben
  ```

  Dies gilt gleichermaßen für eigene Typen. Bei der Initialisierung wird dabei der Konstruktor aufgerufen, bei der Freigabe der Destruktor:

```
{
    A a; // A::A() wird hier aufgerufen
    a.foo();
} // A::~A() wird hier aufgerufen und a wird freigegeben
```

- *Freispeicher*, welcher explizit mit den Operator `new` angefordert werden kann und dann wenn er nicht mehr gebraucht wird, mit dem Operator `delete` freigegeben werden muss. Wird mehr Freispeicher (auch dynamischer Speicher oder Heap genannt) benötigt, so fordert `new` diesen von dem Betriebsystem an. Für Freispeicher sollte immer die *RAII* Technik (Siehe auch 1.1.2) verwendet werden:

```
class A
{
    int* m_p;
public:
    A() : m_p(new int) {}
    ~A() {delete m_p;}
};
{
    A a;
} // m_p wird hier mit delete freigegeben
```

RAII ist aber nur eine Möglichkeit um die Frage der Besitzsemantik zu klären. Um Speicherlecks oder Zugriff auf freigegebene Ressourcen zu vermeiden muss immer klar definiert sein, wem eine Ressource gehört. Nur der Besitzer darf und muß seine Ressourcen freigeben. Zudem obliegt es in seiner Verantwortung sich darum zu kümmern, dass keine weiteren Zugriffe nach der Freigabe erfolgen können. RAII setzt all diese Konzepte um. Da sie so nützlich ist, gibt es das Klassen-Template `auto_ptr` (siehe Kapitel 3.1.3), wodurch die wiederholte Implementation von Klassen wie `A` in den meisten Fällen vermieden werden kann.

Automatischer Speicherplatz ist weniger fehleranfällig, da leider oftmals Code geschrieben wird, der die Frage der Besitzsemantik nicht eindeutig klärt oder zu kompliziert macht. Er ist aber auch wesentlich performanter, da statt der komplexen Heapverwaltung nur ein Stackpointer inkrementiert und dekrementiert werden muss. Allerdings ist es damit nicht möglich, dynamische Datenstrukturen zu implementieren. Für diese ist es aber ratsam, immer wenn möglich die STL zu verwenden.

2.4.2 Klassen in C++

Den Aufbau einer Klasse in C++ , eingeleitet durch das Schlüsselwort `class`, haben wir bereits in einigen Beispielen gesehen:

```
class Klassenname { ... };
```

Eine Klasse kann aber auch mit `struct` eingeleitet werden:

```
struct Klassenname { ... };
```

Die Unterscheidung ist, dass die Zugriffskontrolle und Vererbung dann per default `public` ist. Wenn eine Klasse (oder in diesem Fall Struktur) hauptsächlich einige Variablen kapselt, aber direkten Zugriff darauf zulässt, so ist `struct` ansonsten `class` zu empfehlen. Es ist aber problemlos möglich dass in einer Vererbungshierarchie `struct` und `class` wechselt.

In einer flexiblen und mächtigen Sprache sind *Konventionen* besonders wichtig. In der Übung werden Klassen mit großen Anfangsbuchstaben geschrieben – bevorzugt beginnend mit einem C um explizit zu machen, dass es sich um eine konkrete Klasse handelt. Interfaces hingegen müssen mit einem I beginnen. Namen von Konstanten werden nur mit Großbuchstaben und Membervariablen beginnend mit `m_` geschrieben. Zu beachten ist, dass in C++ es nicht erlaubt ist, globale Namen mit `_` beginnen zu lassen, da diese reserviert sind. In C++ wird streng zwischen Groß- und Kleinschreibung unterschieden. Die Namen `A` und `a` sind daher verschieden.

Der Inhalt der Klasse steht innerhalb geschwungener Klammern. Nach dem Inhalt können gleich Variablen mit dem Typ der Klasse definiert werden. Die Klassendefinition wird mit `;` abgeschlossen:

```
class A {} a1, a2;
```

Eine Klasse kann mehrere *Konstruktoren* enthalten:

```
class Circle
{
    int m_radius;
public:
    Circle(int radius) : m_radius(radius) {}          // 1
    Circle(Circle const& c): m_radius(c.m_radius) {} // 2
    Circle() :m_radius(1) {}                          // 3
};
```

Die Klasse `Circle` hat drei verschiedene Konstruktoren, die sich in der Anzahl oder in den Typen der formalen Parameter unterscheiden. Das ist ein typischer Fall von Überladen. Der erste Konstruktor initialisiert die Instanz mit dem übergebenen `int` Wert. Der zweite Konstruktor ist ein sogenannter *Kopierkonstruktor*. Er initialisiert die neue Instanz aus einer anderen Instanz eines `Circle`. Der dritte Konstruktor nimmt keinen Wert entgegen und initialisiert `m_radius` mit 1. Dieser Konstruktor hätte, mit Hilfe eines Default-Wertes, einfach eingespart werden können:

```
class Circle
```

```
{
    // ...
    // ohne Konstruktor 1, 3
    Circle(int radius = 1) : m_radius(radius) {} // 1a
};
```

Beim Erzeugen einer neuen Instanz werden dem Konstruktor Argumente übergeben. Anhand der Anzahl und den Typen der Argumente wird der geeignete Konstruktor gewählt:

```
Circle a(2); // 1
Circle b(a); // 2
Circle c;    // 3 oder 1a
```

In der dritten Zeile darf keine Klammer verwendet werden.

```
Circle c(type);
```

Dieser Ausdruck deklariert die Funktion `c`, mit dem Ergebnistyp `Circle` und dem Parameter `type`. Bei leerer Klammer ist `type` automatisch `void`, da `()` in C++ bei Methoden und Funktionen `(void)` entspricht.

Die Konstruktoren können gleichermaßen für den Freispeicher verwendet werden:

```
Circle *a = new Circle(2);  // 1
Circle *b = new Circle(*a); // 2
Circle *c = new Circle();   // 3 oder 1a
delete c;
delete b;
delete a;
```

In der dritten Zeile wird eine Klammer verwendet, da sie veranlasst, dass auch dann Typen initialisiert werden, wenn sie keinen Konstruktor anbieten:

```
int *i = new int();
// i ist auf 0 initialisiert
delete i;
```

Falls eine Klasse ohne Konstruktoren oder Destruktoren definiert ist, bekommt sie einen default Konstruktor, Kopierkonstruktor, Zuweisungsoperator und Destruktor:

- Der *default Konstruktor* initialisiert alle Instanzvariablen des Typs, mit einem Konstruktor ohne Parameter. Ist von einem Typ der Instanzvariablen dieser nicht vorhanden, so gibt der Compiler eine Fehlermeldung aus. Wie aber bereits angesprochen, wird dieser Konstruktor bei eingebauten Datentypen (oder Strukturen die nur eingebaute Datentypen haben) nur aufgerufen bei:

```
struct pod { int i; double d; };

void foo() {
    pod i1 = pod();
    pod *i2 = new pod();
    delete i2;
}
```

Dieses Verhalten ist vor allem bei Generizität interessant, da es dort keine Möglichkeit gibt zwischen eingebauten und eigenen Klassen zu unterscheiden.

Wenn hingegen in einer Klasse ein Konstruktor angegeben ist, dann wird dieser verwendet oder der Compiler gibt eine Fehlermeldung aus, dass die Argumente nicht übereinstimmen.

- Der *default Kopierkonstruktor* kopiert alle Instanzvariablen in das neu erzeugte Objekt:

```
Circle c1 = Circle(); // default initialisierung fuer pod
Circle c2 (c1); // Kopierkonstruktor
Circle c3 = c1; // andere Schreibweise
```

  Es ist darauf zu achten, dass nur eine flache Kopie durchgeführt wird. Das Verhalten bei Zeigern ist deshalb meistens unerwünscht, da dann mehrere Instanzen auf die gleichen Objekte zeigen. In diesem Fall muss der Kopierkonstruktor dann überschrieben werden:

```
class Circle
{
    // ...
    Circle (Circle const& other) { ... }
};
```

- Der *default Zuweisungsoperator* verhält sich gleich wie der Kopierkonstruktor, nur wird er bei Zuweisungen verwendet. Das sind all jene Stellen an denen keine Variablen definiert werden, aber eine Zuweisung stattfindet:

```
Circle c1 = Circle();
Circle c2 = Circle();
c1 = c2; // Zuweisung
```

  Um den Zuweisungsoperator für einen eigenen Typ selbst zu definieren, verwendet man den Operator `operator=`:

```
class Circle
{
    // ...
    Circle& operator= (Circle const& other) { ... }
};
```

- Der *default Destruktor* führt nichts durch. Es ist allerdings darauf zu achten, dass er nicht `virtual` ist.

static

Manchmal benötigt man Variablen, die nicht zu einer bestimmten Instanz einer Klasse gehören, sondern zur Klasse selbst. Solche *Klassenvariablen* kann man in C++ einfach durch Voranstellen des Schlüsselwortes `static` deklarieren. Hier ist ein Beispiel für eine Klassenvariable:

```
class Circle
{
    // ...
    static int maxRadius;
};
```

wobei nicht konstante statische Variablen definiert werden müssen:

```
int Circle::maxRadius = 1023;
```

Solche Variablen stehen nicht in den Instanzen der Klasse, sondern in der Klasse selbst. Der Zugriff erfolgt über den Namen der Klasse – z. B. `Circle::maxRadius`.

Statische *Konstanten* stellen einen häufig verwendeten Spezialfall von Klassenvariablen dar. Sie werden durch `static const` gekennzeichnet:

```
class Circle
{
    // ...
    static const int MAX_SIZE = 1024;
};
```

Eine zusätzliche Definition ist nicht erforderlich. Der Wert solcher Variablen kann nach der Initialisierung nicht mehr geändert werden.

Es gibt auch statische Methoden (siehe Abschnitt 4.1.3), welche nur auf Klassenvariablen zugreifen sollten.

Auch wenn es verlockend ist, sollte man es vermeiden, Klassenvariablen als Variablen zu sehen, die allen Instanzen einer Klasse gemeinsam gehören, da diese Sichtweise längerfristig zu unklaren Verantwortlichkeiten und damit zu Konflikten führt. Von einer nicht-statischen Methode aus sollte man auf eine Klassenvariable nur mit derselben Vorsicht zugreifen, mit der man auf Variablen eines anderen Objekts zugreift – am besten nicht direkt, sondern nur über statische Zugriffsmethoden.

this

Das Schlüsselwort `this` kann in nicht-statischen Methoden von Klassen verwendet werden. `this` bezeichnet immer die aktuelle Instanz einer Klasse. In Konstruktoren ist das die Instanz, die gerade erzeugt wird, in Destruktoren die Instanz die gerade freigegeben wird. Sollten formale Parameter (oder lokale Variablen) Variablen in der aktuellen Instanz der Klasse verdecken, so kann `this` verwendet werden um trotzdem auf die Instanzvariablen zugreifen zu können.

Abstrakte Klassen

Wie wir in Abschnitt 2.2 gesehen haben, können Klassen und Methoden in C++ abstrakt sein. Eine Klasse ist abstrakt, wenn sie zumindest eine abstrakte Methode enthält. Solche Klassen, wie auch abstrakte Methoden, für die keine Implementierungen angegeben sind, müssen mit `=0` gekennzeichnet sein.

Geschachtelte Klassen

Zudem gibt es *geschachtelte Klassen* (nested classes), die innerhalb anderer Klassen definiert sind. Geschachtelte Klassen können überall definiert sein, wo Variablen definiert werden dürfen, wie am folgenden Beispiel verdeutlicht wird:

```
class EnclosingClass
{
    class InnerClass
    {
    } m_inner;
    void bar()
    {
        class LocalClass { } local;
    }
};
```

Geschachtelte Klassen geben den inneren Klassen nicht automatisch Zugriffsrechte [Str00] (siehe Kapitel 2.4.4). Um das zu bewerkstelligen, können Klassen als `friend` einer Klasse deklariert werden. Diese erhalten den Zugriff auf private Instanzvariablen, Methoden und Typdefinitionen. Es ist aber genauso möglich Funktionen als `friend` zu deklarieren. Die Erlaubnis gilt aber nur in einer Richtung. Es ist einleuchtend, daß `friend`-Klassen nur verwendet werden sollten, um stark gekoppelte Konzepte zu beschreiben[Str00].

2.4.3 Benutzerdefinierte Typen

C++ hat alle Typen von C übernommen, aber zugleich eine geniale Möglichkeit eingebaut, die es erlaubt eigene Typen so zu schreiben, dass sie nicht von eingebauten Typen unterscheidbar sind.

Das Basiskonzept, um das zu bewerkstelligen ist die Möglichkeit, alle Operatoren, die direkt auf eingebaute Typen wirken, überladen zu können. Die Möglichkeiten die sich damit, hauptsächlich für numerische Applikationen, eröffnen sind gewaltig. Zu einem können Typen von ganzen und reellen Zahlen geschrieben werden, die sich zwar komplett gleich verhalten, sich aber nur beschränkt oder gar nicht aufeinander zuweisen lassen. Das kann für Währungen oder Zahlen mit unterschiedlicher Einheiten sehr nützlich sein. Ein anderer Anwendungsfall ist es den Definitionsbereich oder Genauigkeit von Zahlen zu verkleinern (beispielsweise eine Zahl von 0 bis 100 für Prozent) oder zu vergrößern bis hin zu beliebig großen oder genauen Zahlen. Für spezielle Hardware können so auch Unterstützung für spezielle Typen bereitgestellt werden, z.B. `half` für GPUs, weobei hier die Implementation dann in Assembler erfolgen muss. Die Hauptanwendung liegt aber bei zahlenähnlichen Klassen, die die (meisten) algebraischen Gesetze befolgen wie Matrizen (bzw. Tensoren) oder komplexen Zahlen. Aber auch abseits von Numerik kann dieses Feature sehr eleganten syntaktischen Zucker bieten, beispielsweise für Zeit und Datum.

Allerdings sollte dieses Feature für komplett artfremde Klassen nur sehr bedacht verwendet werden. Um das schlimmste zu verhindern, sind zumindest die Prioritätsregeln und die Stelligkeiten der Operatoren fix vorgegeben und können nicht verändert werden.

> **Tipp:** Operatoren sollten immer dann überladen werden, wenn die Verwendung der Klasse für einen Experten der Domäne dadurch natürlicher wirkt.

Für das Überladen wird das Schlüsselwort `operator`, gefolgt von dem Zeichen für den Operator selbst, angegeben:

```
class complex
{
    double re, im;
public:
    //...
    complex& operator+= (complex const& c)
    {
        re += c.re;
```

```
        im += a.im;
        return *this;
    }
};
```

Neben der Verwendung der Operatoren wie bei `int` und `double` gewöhnt, wird auch eine ausgeschriebene Variante unterstützt:

```
void f(Zahl a, Zahl b, Zahl c)
{
    Zahl e1 = a+b*c;
    Zahl e2 = operator+(a, operator*(b,c));
}
```

Wie man hier im Vergleich sieht, kann die prägnante Schreibweise, die dadurch ermöglicht wird, kann kaum überbewertet werden. Sie ist, durch jahrelanges Training in der Schule, wesentlich besser lesbarer und verständlicher.

Andererseits ist es oftmals unabdingbar, Operatoren zu überladen, um eine gewisse Funktionalität zu erreichen. Soll ein eigener Typ auch mit den Eingabe/Ausgabe Streams von C++ zusammenarbeiten, so muss der Operator `operator>>` für Eingabe und der Operator `operator<<` für Ausgabe überladen werden:

Listing 2.5: hex.h

```
#include <iostream>
#include <iomanip>

class Hex
{
    int m_i;
public:
    Hex (int i) : m_i (i) {}
    friend std::ostream& operator<< (std::ostream& os, Hex const& h);
};

std::ostream& operator<< (std::ostream& os, Hex const& h)
{
    std::ios_base::fmtflags fl = os.flags();
    os << "0x" << std::setw(8) << std::setfill('0')
           << std::uppercase  << std::hex << h.m_i;
    os.flags (fl);

    return os;
}
```

In diesem Beispiel wird demonstriert, wie ein Typ, der ein `int` kapselt, eine benutzerdefinierte Ausgabe realisiert. Soll nun eine Zahl Hexadezimal mit Präfix und Großbuchstaben in einer fixierten Länge von acht Zeichen ausgegeben werden, so kann nun folgendes verwendet werden:

```
cout << Hex (i) << endl;
```

Operatoren können sowohl Methoden als auch Funktionen sein. Diese Unterscheidung, kann nach [Str00] so getroffen werden: Wird prinzipiell das erste Argument modifiziert, (wie z.B. bei `+=`) so ist die Implementation als Methode vorzuziehen. Wird ein neuer Wert erzeugt, aber kein Wert verändert, (wie z.B. bei `+` und `<<`) so ist eine Funktion vorzuziehen. Muss in diesem Fall auch auf `private` Variablen zugegriffen werden, so ist `friend` zu verwenden.

Um einen benutzerdefinierten Typ zu einem fundamentalen Typ zu konvertieren, werden *Konvertierungsoperatoren* verwendet:

```
class Tiny
{
    char v;
public:
    operator int() const {return v;}
};
```

Jedesmal, wenn eine Instanz von `Tiny` an einer Stelle steht, wo ein `int` benötigt wird, wird diese automatisch konvertiert. Diese Möglichkeit sollte aber sparsam verwendet werden, da sie sehr leicht zu unbeabsichtigten Konvertierungen führt.

Die Möglichkeiten Operatoren in C++ zu überladen gehen viel weiter, als sie hier behandelt werden. Im Kapitel 3.1.4 wird noch kurz auf `++`, `++(int)`, `*`, `->` und `==` eingegangen um eigene Iteratoren zu implementieren. Es ist aber auch möglich mit dem Operator `operator[]` Indexzugriffe zu erlauben. Ein sehr mächtiges Konzept, welches zu den fundamentalen Mitteln gehört um in C++ funktional zu programmieren, ist das Überladen von dem Operator `operator()`. Diese Objekte werden *Funktoren* genannt. Das sind Objekte die sich wie Funktionen verhalten.

Es sei hier abschießend noch angemerkt, dass `typedef` *keinen* neuen Typen erstellt.

2.4.4 Zugriffskontrolle in C++

Variablen und Methoden in Klassen können mit Hilfe von *Zugriffskontrolle* gegen versehentliches Verwenden geschützt werden. Dazu dienen die Schlüsselwörter `public`, `protected` und `private`. Diese trennen die Klasse in Bereiche und gelten jeweils für alle folgende Felder, (wie Variablen und Methoden) bis ein anderes Schlüsselwort die Zugriffskontrolle ändert oder die Klassendefinition endet. Felder in einem `private` Bereich sind am besten geschützt. Sie können nur von Objekte der eigenen Klasse und von allen als `friend` deklarierten Klassen verwendet werden. Felder im `public`

Bereich sind hingegen überhaupt nicht geschützt. Sie können überall verwendet werden, wo eine Instanz der Klasse bekannt ist. Neben diesen beiden Extremfällen gibt es noch eine weitere Möglichkeit der Zugriffskontrolle. Dieser Bereich heißt `protected` und erlaubt es neben den von `private` erlaubten Klassen auch allen Unterklassen auf die Felder zuzugreifen.

Folgende Tabelle gibt eine Übersicht über die Zugriffskontrolle:

	`public`	`protected`	`private`
von überall	ja	nein	nein
Objekte von Unterklassen	ja	ja	nein
Objekte von eigener Klasse und `friend`	ja	ja	ja

Folgende Tabelle gibt eine Übersicht über die Vererbung von Zugriffskontrolle:

	`public`	`protected`	`private`
`public` Vererbung	`public`	`protected`	`private`
`protected` Vererbung	`protected`	`protected`	`private`
`private` Vererbung	`private`	`private`	`private`

Für weniger geübte C++ -ProgrammiererInnen ist es gar nicht leicht, die Zugriffskontrolle stets richtig zu gestalten. Hier sind einige Ratschläge, die diese Wahl erleichtern sollen:

- Alle Methoden, Konstanten und in ganz seltenen Fällen auch Variablen, die man bei der Verwendung der Klasse oder von Instanzen der Klasse benötigt, sollen `public` sein.
- Man verwendet `private` am besten für alle Methoden und Variablen in einer Klasse, die nur innerhalb der Klasse verwendet werden sollen. Das betrifft meist Methoden, deren Funktionalität außerhalb der Klasse nicht verständlich ist, sowie Variablen, die diese Methoden benötigen.
- Wenn Variablen und Methoden für die Verwendung einer Klasse und ihrer Instanzen nicht nötig sind, diese Methoden und Variablen aber bei späteren Erweiterungen der Klasse voraussichtlich hilfreich sind, verwendet man am besten `protected`.

Faustregel: Variablen sollen nicht `public` sein.

Es ist oft schwierig, geeignete Zusicherungen für Zugriffe auf Variablen anzugeben. Das ist ein wichtiger Grund für die Empfehlung, Variablen generell nicht `public` zu machen. Statt einer solchen Variablen kann man in der nach außen sichtbaren Schnittstelle eines Objekts immer auch eine Methode zum Abfragen des aktuellen Wertes („getter“) und eine zum Setzen des Wertes („setter“) schreiben. Obwohl solche Methoden oft weniger problematisch sind als Variablen, ist es noch besser, wenn sie gar nicht benötigt werden. Solche Methoden deuten, wie nach außen sichtbare Variablen, auf starke Objekt-Kopplung und niedrigen Klassen-Zusammenhalt und damit auf eine schlechte Faktorisierung des Programms hin. Refaktorisierung ist angesagt.

> **Faustregel:** Methoden zum direkten Setzen bzw. Abfragen von Variablenwerten sind zu vermeiden.

Wenn unklar ist, welche Zugriffskontrolle am besten geeignet ist, verwendet man zu Beginn der Entwicklung die am stärksten eingeschränkte Variante. Erst wenn sich herausstellt, dass eine weniger restriktive Variante nötig ist, erlaubt man weitere Zugriffe. Diese Vorgehensweise ist empfehlenswert, da es um einiges einfacher ist, die Zugriffskontrolle zu lockern als umgekehrt.

Zugriffskontrolle kann nur auf Klassen eingeschränkt werden. Es gibt keine Möglichkeit der Einschränkung auf einzelne Objekte. Daher sind alle Variablen eines Objekts stets auch außerhalb des Objekts zugreifbar, zumindest von einem anderen Objekt derselben Klasse aus. Das bedeutet jedoch nicht, dass solche Zugriffe wünschenswert sind. Im Gegenteil: Direkte Zugriffe (vor allem Schreibzugriffe) auf Variablen eines anderes Objekts führen leicht zu inkonsistenten Zuständen und Verletzungen von Invarianten. Dieses Problem kann nur durch vorsichtige, disziplinierte Programmierung gelöst werden. Zugriffskontrolle kann aber helfen, den Bereich, in dem es zu direkten Variablenzugriffen von außen kommen kann, klein zu halten. Softwareentwickler sind ja stets für ganze Klassen aber nicht für einzelne Objekte verantwortlich. Insofern sind Klassen und als Grundeinheiten für die Zugriffskontrolle gut gewählt.

2.5 Wiederholungsfragen

1. In welcher Form kann man durch das Ersetzbarkeitsprinzip Wiederverwendung erzielen?

2. Unter welchen Bedingungen, die von einem Compiler überprüfbar sind, ist ein Typ im Allgemeinen Untertyp eines anderen Typs? Welche zusätzliche Bedingungen müssen in C++ gelten? (Hinweis: Sehr häufige Prüfungsfrage!)

3. Sind die in Punkt 2 angeschnittenen Bedingungen hinreichend, damit das Ersetzbarkeitsprinzip erfüllt ist? Wenn nicht, was muss noch beachtet werden?

4. Welche Rolle spielt dynamisches Binden für die Ersetzbarkeit und Wartbarkeit?

5. Welche Arten von Zusicherungen werden unterschieden, und wer ist für deren Einhaltung verantwortlich?

6. Wie müssen sich Zusicherungen in Unter- und Obertypen zueinander verhalten, damit das Ersetzbarkeitsprinzip erfüllt ist? Warum? (Hinweis: Häufige Prüfungsfrage!)

7. Warum sollen Schnittstellen und Typen stabil bleiben? Wo ist Stabilität besonders wichtig?

8. Was ist im Zusammenhang mit allgemein zugänglichen (`public`) Variablen und Invarianten zu beachten?

9. Wie genau sollen Zusicherungen spezifiziert sein?

10. Wozu dienen abstrakte Klassen und abstrakte Methoden? Wo und wie soll man abstrakte Klassen einsetzen?

11. Ist Vererbung dasselbe wie das Ersetzbarkeitsprinzip? Wenn Nein, wo liegen die Unterschiede?

12. Worauf kommt es zur Erzielung von Codewiederverwendung eher an – auf Vererbung oder Ersetzbarkeit? Warum?

13. Welche Arten von Speicherplatz gibt es in C++?

14. Was ist RAII? Was wird damit bezweckt?

15. Wie kann man den Konstruktor überladen? Was wird bei einer leeren Klassen angelegt? Was sind Default Werte bei Parameter?

16. Was bedeuten folgende Begriffe in C++?

    - Instanzvariable, Klassenvariable, Klassenkonstanten

- geschachtelte und innere Klasse
- Konstruktor, Kopierkonstruktor, Zuweisungsoperator, Destruktor

17. Was sind abstrakte Klassen? Was sind Interfaces in C++ ? Welcher Vorteil ergibt sich durch deren Verwendung?

18. Wie können eigene Typen die gleiche Syntax wie eingebaute Typen bekommen? Wie weit sollte man das einsetzen? Wie können eigene Typen in Streams eingesetzt werden?

19. Welche Möglichkeiten gibt es bei der Zugriffskontrolle in C++ , und wann soll man welche Möglichkeit wählen? Was bezweckt man mit dem Schlüsselwort `friend`?

20. Wie verändert sich der Zugriff nach einer Vererbung? Können Zugriffsrechte außerhalb der Klasse erweitert werden?

Kapitel 3

Generizität und Ad-hoc-Polymorphismus

In Kapitel 2 haben wir uns mit enthaltendem Polymorphismus beschäftigt. Nun werden wir alle weiteren Arten von Polymorphismus in objektorientierten Sprachen betrachten. Die Abschnitte 3.1 und 3.2 sind der Generizität und ihrer Verwendung gewidmet. Zum besseren Verständnis behandeln wir in Abschnitt 3.3 eine Alternative zur Generizität, die auf dynamischen Typvergleichen und Typumwandlungen beruht. In Abschnitt 3.4 werden wir uns Unterschiede zwischen Überladen und mehrfachem dynamischem Binden durch Multimethoden vor Augen führen. Dabei werden wir Möglichkeiten aufzeigen, mehrfaches dynamisches Binden in Sprachen zu verwenden, die nur einfaches dynamisches Binden bereitstellen. Wir werden uns in Abschnitt 3.5 mit Ausnahmebehandlungen in C++ beschäftigen, obwohl entsprechendes Konzept keine Ausformung des Polymorphismus ist.

3.1 Generizität

Generische Klassen, Typen und Routinen enthalten Typparameter, für die Typen eingesetzt werden. Damit ist Generizität eine weitere Form des universellen Polymorphismus, die Wiederverwendung unterstützen kann. Generizität ist im Wesentlichen ein statischer Mechanismus. Dynamisches Binden wie beim enthaltenden Polymorphismus ist nicht nötig. Das ist ein wichtiges Unterscheidungsmerkmal zwischen den Unterarten des universellen Polymorphismus. Templates sind das Sprachmittel um in C++

Generizität umzusetzen. Die sogenannte Meta-Programmierung mit Templates ermöglicht aber wesentlich mehr. Fast die gesamte STL ist generisch und Generizität ist bereits so wichtig, dass der kommende Standard C++0x sich fast ausschließlich diesem Thema widmet.

3.1.1 Wozu Generizität?

Bei der Programmierung mit Generizität werden an Stelle expliziter Typen im Programm Typparameter verwendet. Typparameter sind Namen, die später durch Typen ersetzt werden. Anhand eines Beispiels wollen wir zeigen, dass eine Verwendung von Typparametern anstelle von Typen und die spätere Ersetzung der Typparameter durch Typen sinnvoll sein kann:

Beispiel. Programmcode für Listen soll entwickelt werden. Alle Elemente in einer Liste sollen vom selben Typ, sagen wir `std::string` sein. Es ist einfach, entsprechenden Programmcode zu schreiben. Bald stellt sich jedoch heraus, dass wir auch eine Liste mit Elementen vom Typ `int` sowie eine mit Instanzen von `Student` brauchen. Da der existierende Programmcode nur mit Zeichenketten umgehen kann, müssen wir zwei neue Varianten schreiben. Untertypen und Vererbung sind dabei wegen der Unterschiedlichkeit der Typen nicht hilfreich. Aber Typparameter können helfen: Statt für `std::string` schreiben wir den Code für `Element`. Der Name `Element` ist dabei kein tatsächlich existierender Typ, sondern einfach nur ein Typparameter. Den Code für Listen mit Instanzen von `std::string`, `int` und `Student` kann man daraus erzeugen, indem man alle Vorkommen von `Element` im Programmcode durch diese Typnamen ersetzt.

Warum soll man den Code für Listen mit einem Typparameter `Element` schreiben? Diesen Effekt kann man anscheinend auch erzielen, wenn man alle Vorkommen von `std::string` im Code der Listen von Zeichenketten durch `int` beziehungsweise `Student` ersetzt. Leider gibt es dabei aber ein Problem: Der Name `std::string` kann auch für ganz andere Zwecke als für Elementtypen eingesetzt sein. Eine Ersetzung würde alle Vorkommen von `std::string` ersetzen, auch solche, die gar nichts mit Elementtypen zu tun haben. Aus diesem Grund wählt man einen neutralen Namen wie `Element`, der in keiner anderen Bedeutung vorkommt. Zudem gibt ein sprechender Name zusätzliche Informationen über welcher Typ erwartet wird. Oftmals wird aber fast jeder Typ akzeptiert und einfach `T` verwendet.

Natürlich kann man sich Schreibaufwand ersparen, wenn man eine Kopie eines Programmstücks anfertigt und darin alle Vorkommen eines Typparameters mit Hilfe eines Texteditors oder Makros durch einen Typ ersetzt. Aber dieser einfache Ansatz bereitet Probleme bei der Wartung: Nötige Änderungen des kopierten Programmstücks müssen in allen Kopien gemacht werden, was einen erheblichen Aufwand verursachen kann. Leichter geht es, wenn das Programmstück nur einmal existiert. Das ist einer der Gründe, warum viele moderne (nicht nur objektorientierte) Programmiersprachen Generizität unterstützen: ProgrammiererInnen schreiben ein Programmstück nur einmal und kennzeichnen Typparameter als solche. Statt einer Kopie verwendet man nur den Namen des Programmstücks zusammen mit den Typen, die an Stelle der Typparameter zu verwenden sind. Erst der Compiler erzeugt nötige Kopien. Änderungen sind nach dem nächsten Übersetzungsvorgang überall sichtbar, wo das Programmstück verwendet wird.

Tipp: Für Code, der sich nur in Typen oder Konstanten unterscheidet, sind statt Kopien Templates vorzuziehen.

Der Compiler erzeugt Kopien die genauso effizient sind wie handgeschriebener Code. Diese können sogar mittels inline direkt in den Aufrufercode eingebettet werden und sind somit auch in dieser Hinsicht vollständiger Ersatz für Makros. Obwohl immer für jeden Typ Kopien angefertigt werden, kann durch geschickten Einsatz nicht generischen Codes das Programm sogar kleiner machen. Generizität bedeutet bei intensiven Einsatz allerdings einen nicht unerheblichen Mehraufwand für den Compiler. Deshalb gibt es auch Möglichkeiten, dass explizit angegeben wird welche Kopien angefertigt werden sollen. Da dies aber nur die Verantwortlichkeit vom Compiler zum Programmierer verschiebt, und nur bei sehr großen Projekten und sicher nicht in der Laborübung notwendig ist, wollen wir auf diese Techniken hier nicht eingehen. Zur Laufzeit gibt es aber auf jeden Fall keinen zusätzlichen Aufwand. Generizität ist damit ein rein statischer Mechanismus.

3.1.2 Funktions-Templates

Generische Funktionen haben einen oder mehrere Typparameter, die bereits vor der Funktionsdefinition in spitzer Klammer geschrieben werden:

Listing 3.1: max.hpp

```
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a<b ? b : a;
}
```

Diese generische Funktion returniert den größeren Wert über den noch unbekannten Typ `T`.

Betrachten wir in einem Beispiel, wie die Funktion verwendet werden kann:

Listing 3.2: max.cpp

```
#include "max.hpp"

#include <iostream>
#include <string>

int main()
{
    using namespace std;

    int i = 43;
    cout << "max(7,i): " << ::max(7,i) << endl;

    double f1 = 3.4;
    double f2 = -7.4;
    cout << "max(f1,f2): " << ::max(f1,f2) << endl;

    string s1 = "C++";
    string s2 = "C";
    cout << "max(s1,s2): " << ::max(s1,s2) << endl;
}
```

Es fällt sogleich auf dass die gesamte Template-Funktion inkludiert wird. Das ist bei Templates in der Laborübung erlaubt, da der Compiler dann entscheidet, für welche Typen die Template-Funktion benötigt wird. Eine konkrete Funktion von einer Template-Funktion nennt man *Instanz* der Template-Funktion, z.b. `max(float,float)`. In diesem Beispiel werden also drei Instanzen gebildet und verwendet.

Es wird in dem Beispiel `::max` benutzt, um das in `"max.hpp"` global definierte Template `max()` zu verwenden. Damit wird verhindert, dass versehentlich das von der STL in `<algorithm>` definierte Template eingesetzt wird, da dies im Namensbereich `std`, und somit nicht global, ist.

Während bei der Template-Funktion eine spitze Klammer für die Typparameter zu sehen war, fehlen diese bei der Verwendung von `::max`. Das ist deshalb so, weil der Compiler den Typ ermittelt und automatisch die richtige Instanz verwendet. Bei Mehrdeutigkeiten wird mit einem Fehler

abgebrochen. In diesem Fall kann man mit der spitzen Klammer den Typ angeben:

```
cout << "max(3,2.0): " << ::max<int>(3,2.0) << endl;
```

Es können problemlos auch mehrere Überladungen einer Template-Funktion nebeneinander existieren, diese können wiederum Template-Funktionen, oder aber auch normale Funktionen sein. Wir wollen in einem Beispiel betrachten wie `max` für C-Strings überladen werden könnte:

Listing 3.3: cmax.hpp

```
#include <cstring>

inline char const* const& max ( char const* const& a,
                char const* const& b)
{
    return std::strcmp(a,b) < 0 ? b : a;
}
```

Der Typ `char const * const&` ist ein per konstanter Referenz übergebener konstanter Zeiger.

Die besondere Eleganz liegt aber darin, dass jeder beliebige Typ, und damit auch eigene, zusammen mit Templates verwendet werden können. Ein Typ der mit `max` verwendet werden soll, muss nur den Operator `operator<` unterstützten, wobei aber `a<b` und `b<a` nicht gleichzeitig gelten darf. Genau dieser Sachverhalt kann in C++0x mit `concept` direkt ausformuliert werden. Werden in einem Template Operatoren oder Funktionen verwendet, die nicht im `concept` beschrieben sind, wird bereits beim Übersetzen des Templates ein Compilerfehler ausgelöst. Derzeit aber werden diese Konzepte ohne Unterstützung des Compilers in der Dokumentation beschrieben.

Ein eigener Typ könnte folgendermaßen implementiert sein:

Listing 3.4: complex.hpp

```
#include <ostream>

class Complex
{
    int m_real;
    int m_img;
public:
    Complex (int real, int img=0) :
        m_real(real), m_img(img) {}
    bool operator< (Complex const& o) const
    {
        if (m_real == o.m_real)
            return m_img < o.m_img;
        else return m_real < o.m_real;
    }
```

```
    friend std::ostream& operator<<(std::ostream&, Complex const&);
};

std::ostream& operator<< (std::ostream& os, Complex const& c)
{
    return os << '(' << c.m_real << ',' << c.m_img << ')';
}
```

Hier ist wie gefordert der Operator `operator<` implementiert, wodurch problemlos `::max` verwendet werden kann:

Listing 3.5: complex.cpp

```
#include "complex.hpp"
#include "max.hpp"

#include <iostream>

int main()
{
    using namespace std;

    Complex c1 (12, 30);
    Complex c2 (20, 5);
    cout << "max(c1,c2): " << ::max(c1,c2) << endl;
}
```

Eine andere sehr praktische Möglichkeit Funktions-Templates einzusetzen ist der `lexical_cast`. Wenn beispielsweise ein `int` zu einem `std::string` umgewandelt werden soll der Wert konvertiert werden. Dies soll für alle Typen die den Operator `operator<<` für `ostream` oder den Operator `operator>>` für `istream` unterstützen, funktionieren. Eine einfache Implementierung könnte folgendermaßen aussehen:

Listing 3.6: lexicalcast.hpp

```
#include <sstream>

template <typename T, typename S>
T lexical_cast(S const& s)
{
    std::stringstream ss;
    T t;
    ss << s;
    ss >> t;
    return t;
}
```

Obiger Code verwendet die gleiche Konvertierung wie `CounterStack::count` (siehe 1.1.5). Die Verwendung von `lexical_cast` ist sehr einfach und ist gleich wie bei anderen C++-Casts. Es kann sogar der eigene Typ `Complex` verwendet werden, allerdings nur zur Konvertierung in eine Richtung, da der Operator `operator>>` nicht überladen wurde:

Listing 3.7: lexicalcast.cpp

```
#include "lexicalcast.hpp"
#include "complex.hpp"

#include <string>
#include <iostream>

using namespace std;

int main()
{
    string s = "123";
    int i = lexical_cast<int> (s);
    cout << i << endl;

    Complex c (20, 13);
    s = lexical_cast<string> (c);
    cout << s << endl;
}
```

Trotz der Kürze der Implementierung von `lexical_cast` werden die obigen Anforderungen umgesetzt. In der Praxis wird natürlich noch umfangreiche Fehlerbehandlung verlangt, wo dann `boost::lexical_cast` eingesetzt werden sollte.

Funktions-Templates sind sehr gut geeignet, Algorithmen über verschiedene Typen zu implementieren. Vor allem bei numerischen Typen kann das sehr einfach gemacht werden, da durch die Operatoren (siehe Kapitel 2.1.1) und die dahinterstehenden mathematischen Konzepte die Syntax und Semantik sehr klar definiert ist. Oft benötigt werden verschiedene Typen für Währungen oder für metrische und nicht-metrische Systeme um Verwechslungen zu vermeiden. Generische Funktionen bieten einen einfachen Weg, um für solche Typen Algorithmen und Berechnungen zu definieren. Besonders interessant werden Funktions-Template mit dem Konzept der Iteratoren (siehe 3.1.4). Dadurch wird es möglich Algorithmen über verschiedene Container zu schreiben. In anderen Bereichen sind die Konzepte leider nicht so klar definiert. Dafür gibt es Hilfsmittel, die aber relativ kompliziert in der Anwendung sind und den Rahmen dieses Skriptums sprengen würden. Zudem kommt in C++0x das bereits skizzierte neue Sprachmittel `concept`, wodurch Konzepte direkt implementiert werden können.

Verglichen mit einem *C-Makro* oder anderer Textsubstitution ergeben sich folgende Vorteile durch Funktions-Templates:

- Es wird überprüft welchen Typ die Parameter haben.
- Sollte der Typ benötigte Operatoren oder Methoden nicht definiert

haben, kann dies in einer Fehlermeldung gesagt werden. Mißverständliche Fehlermeldungen bezogen auf Code der bereits mit dem Präprozessor bearbeitet wurde, fallen weg.

- Eine generische Funktion kann in einem Namespace sein.
- Durch `const` kann man zusichern, dass die Werte nicht modifiziert werden.
- Das Verhalten ist, als wenn ein Funktionsaufruf stattfinden würde, auch wenn der Code tatsächlich `inline` ist.

Bei Makros in C wird einfach Text substituiert. Deshalb ist dieses Sprachmittel für generische Funktionen sehr problembehaftet, um nicht zu sagen ungeeignet: Wenn das Makro folgendermaßen definiert ist: `#define MAX(x,y) ((x)<(y)?(y):(x))` so würde der Präprozessor bei der Anwendung `MAX(i++,j)` folgenden Code generieren: `((i++)<(j)?(j):(i++))` Dadurch würde `i` im Falle, dass es größer ist, doppelt inkrementiert werden. Noch schlimmer werden die Effekte, wenn Elemente in Makros auch verändert werden (z.B. bei `SWAP(a[i],i)`) oder bei Verwendung von globalen Variablen die lokal im Makro überdeckt werden können. Wie wir bereits gesehen haben, kann `max` (aber auch `swap`) sehr einfach als Funktions-Template geschrieben werden, welches all diese Probleme nicht hat.

Tipp: Benutze Makros nur wenn es unabdingbar ist.

(für Interessierte)

Trotz dieser Probleme gibt es noch einige Bereiche, in denen der Präprozessor auch in C++ sinnvoll verwendet werden kann:

```
#define TEST_CASE (x,y) {if (x) {printf("%s:%d: error in %s: %s\n",\
    __FILE__, __LINE__, __FUNCTION__, y); }}
```

Solche ähnlichen Makros können für Testfälle oder Assertions (Zusicherungen zur Laufzeit) verwendet werden. Es muss hier ein Makro verwendet werden, damit die Makros `__FILE__` usw. an die richtige Stelle im Source platziert werden.

Für sehr einfache Zusicherungen, die zur Laufzeit überprüft werden, kann `assert` aus der Standardlibrary verwendet werden:

Listing 3.8: assert.cpp

```
#include <cassert>
#include <iostream>
```

```
int main()
{
    int i = 5;
    std::cin >> i;
    assert (i != 0);
}
```

Da `assert` auch als Makro implementiert ist, ist es nicht in dem Namensbereich `std`. Dieses Makro kann sehr einfach ausgeschalten werden, indem im Code oder über Compilerflags `NDEBUG` definiert wird.

Ansonsten wird der Präprozessor in C++ nur noch zum Inkludieren und Exkludieren von Dateien oder Bereichen verwendet. Um z.B. einen Bereich auszukommentieren der auch `/**/` Kommentare hat, kann folgendes verwendet werden:

```
#if 0
#endif
```

3.1.3 Klassen-Templates

Container sind in C++ sehr einfach zu verwenden. Der Typparameter des Klassen-Templates ist dabei offen gelassen. Diesen muss man bei der Definition der Variable angeben. Durch den Einsatz von Container wird einem aber sämtliches, sehr effizient implementiertes, Memory-Management abgenommen.

Es gibt verschieden Arten von Container, die für verschiedene Einsatzzwecke optimiert sind. Welche Operatoren und Methoden angeboten werden, variiert je nach dem welche auf der Datenstruktur effizient implementiert werden können.

Es ist natürlich auch möglich, generische Klassen selbst zu schreiben. Dafür wird vor der Klassendefinition das Schlüsselwort `template`, gefolgt von den Typparametern in spitzer Klammer geschrieben:

Listing 3.9: stack2.hpp

```
#include <deque>
#include <stdexcept>

template <typename T>
class Stack
{
private:
    std::deque<T> m_elems;
public:
    virtual void push(T const&);
    virtual T pop();
    virtual ~Stack() {}
```

```
    bool empty() const
    {
        return m_elems.empty();
    }
    template <typename A>
    Stack<T>& operator= (Stack <A> const&);
};

#include "stack2def.hpp"
```

Der Code erinnert sehr stark an `CStack` (siehe 1.1.2). Das ist kein Zufall – es ist eine häufige Vorgehensweise von einem konkreten, ausgetesteten Typen eine generische Version zu schreiben. Diese kann nun für jeden beliebigen Typen verwendet werden.

Die Definition der Methoden erfolgt hier aus Gründen der Übersichtlichkeit in einer eigenen Headerdatei, die aber auch direkt eingebunden sein muss:

Listing 3.10: stack2def.hpp

```
template <typename T>
void Stack<T>::push (T const& elem)
{
    m_elems.push_back(elem);
}

template <typename T>
T Stack<T>::pop ()
{
    if (m_elems.empty()) throw
        std::out_of_range("Stack<>::pop(): cant pop empty stack");
    T ret = m_elems.back();
    m_elems.pop_back();
    return ret;
}

template <typename T>
    template <typename A>
Stack<T>& Stack<T>::operator= (Stack<A> const& op2)
{
    if ((void*)this == (void*)&op2) return *this;

    Stack<A> tmp(op2); // create a copy
    m_elems.clear();
    while (!tmp.empty())
    {
        m_elems.push_front(tmp.pop());
    }

    return *this;
}
```

Abgesehen von der Template Syntax ist der Hauptunterschied, dass hier ein Container statt einem C-Array verwendet wird. `push` wirft nicht mehr

die Ausnahme `std::out_of_range`, da `deque` beliebig wachsen kann. Deshalb macht die Methode `full` auch keinen Sinn, außer man strebt ein gleiche Schnittstelle wie bei `CStack` an.

Neu ist allerdings der Operator `operator=`, der hier nicht so trivial wie bei `CStack` ist. Das ist deshalb so, weil auch ein anderer Typ akzeptiert wird und ein geschachteltes Template notwendig ist. Geschachtelte Templates werden im Gegensatz zu mehreren Typparametern durch die mehrmalige Verwendung von `template` definiert. Der erste Vergleich behandelt den Fall, dass eine Zuweisung auf sich selbst stattfindet. In diesem Fall findet kein Kopieren statt und eine Referenz auf das eigene Objekt wird zurückgegeben. Da `op2` nicht modifiziert werden darf, wird eine Kopie angelegt, bei der `pop` nun so lange aufgerufen wird, bis die Kopie leer ist. Durch einen Konvertierungsoperator zwischen den Typen `T` und `A` (man hätte aber auch ein `lexical_cast` einsetzen können) ist es möglich, dass von einem `Stack<A>` `pop` aufgerufen wird und der Ergebnistyp `A` in einem `Stack<T>` mit `push` eingefügt wird. Zum Schluss wird wieder eine Referenz auf das eigene Objekt zurückgegeben. Verwendet werden kann die Zuweisung folgendermaßen:

Listing 3.11: stack2.cpp

```
#include <iostream>
#include <string>
#include "stack2.hpp"

int main()
{
    using namespace std;

    Stack <int> si;
    Stack <float> sf;
    Stack <string> ss;

    for (int i=0; i<20; i++) si.push(i);

    sf = si;
    sf = sf;
    // ss = si; // ERROR

    cout << sf.pop() << endl;
    cout << sf.pop() << endl;
}
```

Zu Beginn werden drei Stacks verschiedenen Typs definiert. Die Zuweisung vom `Stack` von `int` auf einen `Stack` von `float` ruft obigen Operator `operator=` auf und führt für jeden `int` eine Konversion nach `float` durch. Die Zuweisung auf sich selbst ruft allerdings den default Zuweisungsoperator auf. Zum Schluss werden noch zwei `float` ausgegeben. Die

Zuweisung `ss = si` würde fehl schlagen, da keine Konvertierung von `int` auf `std::string` definiert ist. Dieses Problem lässt sich mit dem oben erwähnten Einsatz von `lexical_cast` umgehen. Auch das Testprogramm von `CStack` lässt sich nun problemlos mit dem Klassen-Template `Stack` implementieren:

Listing 3.12: stacktest2.cpp

```
#include "stack2.hpp"
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char **argv) try
{
    Stack<string> s;
    int i;

    for (i=1; i<argc; i++) s.push (argv[i]);
    for (;i>1;i--) cout << s.pop() << endl;
} catch (out_of_range const& oor) {
    cerr << argv[0] << ": " << oor.what() << endl;
    return 1;
}
```

Im ursprünglichen Testprogramm war es nur erlaubt, genau `N` Elemente mit `push` aufzunehmen. Diese Zahl war konstant. Trotzdem wurde dynamisch ein Array alloziert, obwohl das auch komplett statisch und damit viel effizienter sein könnte. Um das zu erreichen, nutzen wir die Eigenschaft aus, dass Template-Parameter nicht immer Typparameter sein müssen:

Listing 3.13: stack3.hpp

```
#include <deque>
#include <stdexcept>

template <typename T, int maxsize>
class Stack
{
private:
    T m_elems[maxsize];
    int m_numelems;
public:
    Stack() : m_numelems(0) {}
    virtual void push(T const&);
    virtual T pop();
    virtual ~Stack() {}
    bool empty() const
    {
        return m_numelems == 0;
    }
    bool full() const
    {
```

```
        return m_numelems == maxsize;
    }
};

#include "stack3def.hpp"
```

Der neue Parameter `maxsize` ist kein Typparameter sondern übergibt die maximale Größe des Stacks. Wird ein Stack instanziert, so wird diese konstante Größe verwendet, um das Array `m_elems` auf dem Stack anzulegen. Im Gegensatz zu `CStack` wird die Instanzvariable `m_maxsize` nun nicht mehr benötigt. Es wird allerdings nun wieder die Methode `full` gebraucht. Der Template-Parameter `maxsize` kann hier direkt wie eine Konstante verwendet werden. Die Definitionen von `push` und `pop` werden auch der Vollständigkeit halber angegeben:

Listing 3.14: stack3def.hpp

```
template <typename T, int maxsize>
void Stack<T, maxsize>::push (T const& elem)
{
    if (full()) throw std::out_of_range
        ("Stack<>::push(): cant push on full stack");
    m_elems[m_numelems++] = elem;
}

template <typename T, int maxsize>
T Stack<T, maxsize>::pop ()
{
    if (empty()) throw std::out_of_range
        ("Stack<>::pop(): cant pop empty stack");
    return m_elems[--m_numelems];
}
```

Hier ist darauf zu achten, dass immer alle Template-Parameter anzugeben sind. Nun kann das Testprogramm mit genau der gleichen Funktionalität wie das ursprüngliche implementiert werden:

Listing 3.15: stacktest3.cpp

```
#include "stack3.hpp"
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char **argv) try
{
    Stack<string, 5> s;
    int i;

    for (i=1; i<argc; i++) s.push (argv[i]);
    for (;i>1;i--) cout << s.pop() << endl;
} catch (out_of_range const& oor) {
    cerr << argv[0] << ": " << oor.what() << endl;
```

```
    return 1;
}
```

Mittels `<..., int maxsize = 5>` hätte man auch einen Defaultwert angeben können. Da 5 aber willkürlich ist, wurde es in diesem Beispiel unterlassen. `Stack<int,20>` ist ein anderer Typ als `Stack<int,40>` und es sind keine impliziten oder expliziten Konvertierungen definiert. Sie können nicht anstatt des anderen verwendet werden. Auch Zuweisungen untereinander sind ohne benutzerdefiniertes Überladen nicht möglich.

Ähnlich dem Einführungsbeispiel wollen wir nun einen Typ `CounterStack<T,int>`, welcher ein Untertyp von `Stack<T,int>` ist:

Listing 3.16: counterstack3.hpp

```
#include "stack3.hpp"

template <typename T, int maxsize>
class CounterStack : public Stack<T, maxsize>
{
private:
    int m_counter;
public:
    CounterStack (int c=0) :
        Stack<T, maxsize>(), m_counter(c)
    {}
    virtual void push (T const& elem);
    void count();
};

#include "counterstack3def.hpp"
```

Um von einem Klassen-Template zu erben, ist der vollständige Typ anzugeben. Da hier wieder ein Template geschrieben wird, können die Templateparameter einfach weiter übergeben werden. Ansonsten müssen die Typparameter mit konkreten Typen belegt werden. Auch bei der Initialisierung der Oberklasse muss der vollständige Typ angegeben werden. Hier sind die Definitionen der Methoden angegeben:

Listing 3.17: counterstack3def.hpp

```
template <typename T, int maxsize>
void CounterStack<T,maxsize>::push (T const& elem)
{
    ++ m_counter;
    Stack<T,maxsize>::push(elem);
}

template <typename T, int maxsize>
void CounterStack<T,maxsize>::count()
{
    Stack<T,maxsize>::push(m_counter);
}
```

Bei den Aufrufen der Methoden der Oberklasse ist besondere Vorsicht geboten. Da für Anfänger der Mechanismus, wie nach Namen bei Templates gesucht wird, sehr verwirrend ist, lautet die Empfehlung immer qualifizierte Namen wie `Stack<T,maxsize>::push` oder `this->push` zu verwenden.

> **Tipp:** Qualifiziere die Namen von Methoden bei abgeleiteten Klassen-Templates immer vollständig.

In C++ wird Code wird immer nur für aufgerufene Member-Funktionen instanziert. Das hat zu einem den Vorteil, dass die ausführbaren Dateien nicht größer sind als notwendig und außerdem ist es sogar möglich Typen zu verwenden, bei denen es gar nicht möglich ist alle Methoden zu instanzieren. `CounterStack` ist genau so ein Typ da `count` nur dann aufgerufen werden kann, wenn der Typparameter nach `int` konvertierbar ist. Es wurde absichtlich auf die Konvertierung mittels `lexical_cast` verzichtet, um dieses Verhalten einfacher zu demonstrieren:

Listing 3.18: partial3.cpp

```
#include "counterstack3.hpp"

#include <iostream>
#include <string>

using namespace std;

int main()
{
    CounterStack<int, 5> si;
    si.push(5);
    si.count();
    cout << si.pop() << endl;

    CounterStack<string, 5> cs;
    cs.push("foo");
    // cs.count(); // ERROR
    cout << cs.pop() << endl;
}
```

Wir sehen, dass ein `CounterStack<string>` instanziert werden kann, obwohl die Methode `count` für diesen Typ nicht instanziert werden könnte. Wird die Methode `count` hingegen benützt, so meldet der Compiler einen Fehler.

Klassen-Templates werden nicht nur für Container verwendet. Möglicherweise ist diese Verwendung nicht einmal die häufigste Form, obwohl die STL anderes Vermuten lässt. Wir werden uns in diesem Abschnitt

noch `auto_ptr` betrachten, ein Repräsentant von der Klasse intelligenter Zeiger. `auto_ptr` löst ein sehr konkretes Problem: Es wird sichergestellt, dass ein Speicher der im Client mit `new` angefordert wurde auch wieder freigegeben wird. Das ist somit ein Spezialfall von RAII. Der intelligente Zeiger `auto_ptr` wird durch ein Zeiger initialisiert und wie ein Zeiger dereferenziert. Die Idee ist, dass das Objekt auf welches gezeigt wird am Ende des Gültigkeitsbereiches, wenn es nicht zurückgegeben wird, mit `delete` implizit und automatisch gelöscht wird:

```
auto_ptr<Stack<int> > f (auto_ptr<Stack<int> > p1, Stack<int> *p2)
{
    auto_ptr<Stack<int> > ret (new CounterStack<int >);
    auto_ptr<Stack<int> > box (p2);
    // Arbeite mit p1, ret und box

    if (fehler()) throw stack_error();

    return ret;
} // gibt p1 und p2 frei
```

Sowohl `p1` als auch `p2` werden implizit gelöscht. `ret` hingegen wird zurückgegeben und wird außerhalb der Funktion gelöscht werden, wenn es nicht weitergegeben wird. Als kleiner Hinweis sei hier noch erwähnt, dass bei dem derzeitigen C++ Standard noch `> >` mit einem Leerzeichen getrennt werden müssen.

> **Tipp:** Wenn neuer Speicher in einer Funktion allokiert und zurückgegeben wird, sollte `auto_ptr` verwendet werden.

Die Implementierung eines `auto_ptr` ist erwartungsgemäß sehr einfach:

```
template <typename X>
class auto_ptr
{
    X* ptr;
public:
    explicit auto_ptr (X* p=0) throw()
    {
        ptr = p;
    }
    ~auto_ptr() throw()
    {
        delete ptr;
    }
    X& operator*() const throw() { return *ptr; }
    X* operator->() const throw() { return ptr; }
};
```

Allerdings fehlt hier noch etwas wichtiges um beim Kopieren mehrmaliges Freigeben zu verhindern. Der Kopierkonstruktor ist deshalb bei `std::auto_ptr`

in `<memory>` so überschrieben, dass der kopierte Zeiger nachher auf nichts zeigt. Da Container der STL aber Objekte kopieren bevor sie eingefügt werden, kann `auto_ptr` nicht bei Container eingesetzt werden. In diesen Situationen, wo Kopien erlaubt sein sollen, kann `std::tr1::shared_ptr` verwendet werden.

3.1.4 Iteratoren

Bis jetzt wurde im Detail behandelt wie Algorithmen als Funktions-Template und Container als Klassen-Templates geschrieben werden. Es fehlt aber noch ein wichtiges Konzept, der Klebstoff, der Container und Algorithmen zusammenhält. Algorithmen direkt für Container zu schreiben ist keine gute Idee. Die Kopplung ist einfach zu hoch und manifestiert sich in einer großen Schnittstelle des Containers. Es wäre dann nicht möglich, die Algorithmen direkt auf C-Arrays auszuführen. Ausserdem gäbe es ein Problem zu beschreiben, welche Art von Zugriff ein Algorithmus benötigt oder ein Container anbietet. Und zu guter Letzt müssen die Algorithmen über den Status wo gerade bearbeitet wird verfügen – ein `find` bräuchte zusätzliche Parameter wo die Suche anfangen und aufhören soll.

All diese Probleme und mehr werden durch Iteratoren gelöst. Dieses Kapitel behandelt das Konzept wie in C++ Iteratoren verstanden werden. Das gleichnamige Entwurfsmuster ist im Kapitel 4.3.1 beschrieben. Iteratoren sind eine Generalisierung von Zeigern, welche es C++ Programmen erlauben auf Elemente eines Aggregats (das ist ein Array, ein Stream oder ein Container) zuzugreifen. Alles was sich wie ein Iterator verhält ist auch einer. Die minimal unterstützten Operatoren von Iteratoren mit einer Semantik von Zeigern sind:

- das Element liefern, auf welches gerade gezeigt wird, mit den Operatoren `->` und `*`
- auf das nächste Element zeigen mit dem Operator `++`
- auf Gleichheit testen mit den Operatoren `==` und `!=`

Dem geübten C-Programmierer fällt sofort auf, dass somit auch Zeiger Iteratoren sind.

Um einen gültigen Iterator zu erhalten, bieten Container die Methoden `begin()` und `end()` an. Der Iterator bei `end()` zeigt auf etwas *hinter dem letzten Element*, eine Schleife kann dann abgebrochen werden. Bei einem C-Array verwendet man die Adressen vom Beginn bis zu eins nach dem

Ende (= $Beginn + Anzahl_{Elemente}$). Da dies sehr einfach zu merken ist, sind sogenannte Off-by-one-Error (Fehler bei Grenzen) unwahrscheinlicher. Ein Algorithmus `find` kann mit dem Wissen wie folgt implementiert werden:

Listing 3.19: find.hpp

```
template <class I, class T>
I find (I start, I end, T const& element)
{
    while (start != end
        && *start != element)
    {
        ++start;
    }
    return start;
}
```

Die erste Bedingung ist notwendig, um das Ende zu erkennen und in diesem Fall abzubrechen. So lange das Ende noch nicht erreicht ist, wird mit dem zu suchenden `element` verglichen und der Iterator zum nächsten Element weitergeschaltet. Das gefundene oder das letzte Element wird abschließend zurückgegeben. Dieser Algorithmus kann sehr einfach auf Container oder C-Arrays angewendet werden:

Listing 3.20: find.cpp

```
#include "find.hpp"
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> v;
    for (int i=0; i<20; i++) v.push_back (i);
    vector<int>::iterator it;
    it = ::find (v.begin(), v.end(), 5);
    if (it == v.end()) cout << "Nicht gefunden" << endl;
    else cout << *it << endl;

    int a [] = {1, 2, 3, 4};
    int*pt = ::find (a, a+4, 5);
    if (pt == a+4) cout << "Nicht gefunden" << endl;
    else cout << *pt << endl;

}
```

Tipp: Algorithmen sollten immer mit Hilfe von Iteratoren statt direkt mit Container implementiert werden.

Es gibt verschiedene *Iteratorkategorien*. Alle Kategorien verlangen, dass nur Operatoren, die in amortisierter konstanter Zeit ausgeführt werden können, vorhanden sind. Input-, Output- und Forward Iteratoren bieten nur die Operatoren an die in den Grundkonzepten angegeben sind. Input- und Outputiteratoren können nur einmal und nur in eine Richtung durchlaufen werden. Input Iteratoren unterstützen nur den lesenden, Output Iteratoren nur den überschreibenden Zugriff, Forward Iteratoren hingegen beides. Der Forward Iterator ermöglicht noch zusätzlich die Sicherung der Position, wodurch mehrmaliges Durchlaufen ermöglicht wird. Bidirektionale Iteratoren ergänzen die Forward Iteratoren mit einer effizienten Möglichkeit, mit Hilfe des Operators `operator--`, auch einen Schritt zurück zu machen. Random-Access Iteratoren bilden die mächtigste Kategorie und vervollständigen die Zugriffsmöglichkeiten durch den Operator `operator[]`, der es ermöglicht den Wert von einem beliebigen Element direkt zu lesen und zu schreiben. Zeiger sind somit Random-Access Iteratoren.

> **Tipp:** Algorithmen sollten nur einen Iterator fordern, den sie unbedingt benötigen und Container sollten Iteratoren zurückgeben, die möglichst vollständig die Möglichkeit, wie effizient iteriert werden kann, reflektieren.

Konstante Iteratoren, deren Typ in Containern über `const_iterator` ermittelbar ist, unterstützen keine Möglichkeit den Container zu modifizieren. *Reverse Iteratoren* durchlaufen einen Container in umgekehrter Reihenfolge von `rbegin` nach `rend`. *Insert Iteratoren* ermöglichen das Einfügen neuer Elemente in einen Container. *Stream Iteratoren* ermöglichen schließlich Algorithmen auf Streams auszuführen.

Um einen Iterator selbst zu definieren gibt es Hilfsklassen in `<iterator>`. Hier das Grundgerüst, um die Anforderungen eines einfachen Forward Iterators zu erfüllen:

Listing 3.21: iterator.h

```
#include <iterator>

template <class T>
class Container
{
    class iterator : public std::iterator
        <std::forward_iterator_tag, T>
    {
        // ...
```

```
    public:
        iterator& operator= (iterator const&)
        {
            /* ... */
            return *this;
        }
        bool operator== (iterator const&) const;
        bool operator!= (iterator const& x) const
        {
            return !(x == *this);
        }
        const T& operator*();
        const T* operator->()
        {
            return &(operator*());
        }
        iterator& operator++()
        {
            /* ... */
            return *this;
        }
        iterator operator++ (int)
        {
            iterator tmp(*this);
            ++*this;
            return tmp;
        }
        // ...
    };
    // ...
};
```

3.2 Verwendung von Generizität im Allgemeinen

Wir wollen nun betrachten, wie man Generizität in der Praxis einsetzt. Abschnitt 3.2.1 gibt einige allgemeine Ratschläge, in welchen Fällen sich die Verwendung auszahlt. In Abschnitt 3.2.2 werden wir uns mit möglichen Übersetzungen generischer Klassen beschäftigen und einige Alternativen zur Generizität vorstellen, um ein etwas umfassenderes Bild davon zu bekommen, was Generizität leisten kann.

3.2.1 Richtlinien für die Verwendung von Generizität

Wann und wie soll man Generizität einsetzen? Generell ist der Einsatz immer sinnvoll, wenn er die Wartbarkeit verbessert. Aber oft ist nur schwer entscheidbar, ob diese Voraussetzung zutrifft. Wir wollen hier einige typische Situationen als Entscheidungshilfen (oder Faustregeln) anführen:

Gleich strukturierte Klassen oder Routinen. Man soll Generizität immer verwenden, wenn es mehrere gleich strukturierte Klassen (oder Typen) beziehungsweise Routinen gibt, oder voraussehbar ist, dass es solche geben wird. Typische Beispiele dafür sind Containerklassen wie Listen, Stacks, Hashtabellen, Mengen, etc. und Algorithmen welche mit Iteratoren realisiert sind, etwa Suchfunktionen und Sortierfunktionen. Praktisch alle bisher in diesem Kapitel verwendeten Klassen und Funktionen fallen in diese Kategorie. Wenn es eine Containerklasse für Elemente eines bestimmten Typs gibt, liegt immer der Verdacht nahe, dass genau dieselbe Containerklasse auch für Instanzen anderer Typen sinnvoll ist. Falls die Typen der Elemente in der Containerklasse gleich von Anfang an als Typparameter spezifiziert sind, ist es später leicht, die Klasse unverändert mit Elementen anderer Typen zu verwenden.

Faustregel: Containerklassen sollen generisch sein.

Es zahlt sich aus, Generizität bereits beim geringsten Verdacht, dass eine Containerklasse auch für andere Elementtypen sinnvoll sein könnte, zu verwenden: Elementtypen sind in der Objektschnittstelle sichtbar, und Änderungen der Schnittstelle verursachen einen erheblichen Wartungsaufwand. Man will daher nach Möglichkeit vermeiden, dass diese Typen nachträglich geändert werden müssen. Dies kann man erreichen, indem man statt konkreter Elementtypen in der Schnittstelle nur Typparameter verwendet. Eine nachträgliche Änderung der Elementtypen in einem Container ist damit ohne großen Wartungsaufwand möglich. Andererseits verursacht die Verwendung von Generizität bei der ursprünglichen Erstellung der Containerklasse nur einen unbedeutenden, vernachlässigbaren Mehraufwand. Die Laufzeiteffizienz wird in C++ durch die Verwendung von Generizität überhaupt nicht beeinträchtigt und die Programmgröße entspricht maximal dem, wenn man für jeden Typen einen eigenen Container geschrieben hätte. Es zahlt sich daher aus, Generizität bereits frühzeitig zu verwenden.

Obwohl man auf einen vernünftigen Einsatz von Generizität achtet, passiert es leicht, dass man die Sinnhaftigkeit von Typparametern an bestimmten Stellen im Programm erst spät erkennt. In diesen Fällen soll man das Programm so schnell wie möglich refaktorisieren, also die Klasse oder Routine mit Typparametern versehen. Ein Hinauszögern der Refaktorisierung führt leicht zu unnötigem Programmcode.

Üblicher Programmcode enthält nur relativ wenige generische Containerklassen. Der Grund dafür liegt einfach darin, dass die meisten Programmierumgebungen mit umfangreichen Bibliotheken ausgestattet sind, welche die am häufigsten verwendeten, immer wieder gleich strukturierten Klassen und Routinen bereits enthalten. Man braucht diese Klassen und Routinen also nur zu verwenden, statt sie neu schreiben zu müssen.

Da Templates zusammen mit der STL bereits 1995 in den Sprachstandard aufgenommen wurden (der dann 1998 als `ISO/IEC 14882:1998;` veröffentlicht wurde) sind mittlerweile de facto alle Bibliotheken für C++ generisch. Meta-Programmierung setzt sich zudem auch immer mehr für Anwendungsentwicklung durch und ist damit ein essentieller Bestandteil von Programmierung in C++ geworden.

> **Faustregel:** Klassen und Routinen in Bibliotheken sollten generisch sein.

Abfangen erwarteter Änderungen. Generizität ermöglicht es, Programmteile unverändert zu lassen, obwohl sich Typen ändern. Insbesondere betrifft das Typen von formalen Parametern. Generizität ist dafür geeignet, erwartete Änderungen der Typen von formalen Parametern bereits im Voraus zu berücksichtigen. Man soll daher gleich von Anfang an Typparameter verwenden, wenn man sich erwartet, dass sich Typen formaler Parameter irgendwann ändern. Das gilt auch dann, wenn es sich nicht um Elementtypen in Containerklassen handelt. Anders als im vorigen Punkt brauchen nicht gleichzeitig mehrere gleich strukturierte Klassen oder Methoden sinnvoll sein, sondern es reicht, wenn zu erwarten ist, dass sich Typen in unterschiedlichen Versionen (die nicht gleichzeitig existieren müssen) voneinander unterscheiden.

> **Faustregel:** Man soll Typparameter als Typen formaler Parameter verwenden, wenn Änderungen der Parametertypen absehbar sind.

Beispielsweise schreiben wir eine Klasse, die Konten an einem Bankinstitut repräsentiert. Nehmen wir an, unsere Bank kennt derzeit nur Konten über Euro-Beträge. Trotzdem ist es vermutlich sinnvoll, sich beim Erstellen der Klasse nicht gleich auf Euro-Konten festzulegen, sondern für die Währung einen Typparameter zu verwenden, für den neben einem Euro-Typ auch ein US-Dollar-Typ eingesetzt werden kann (falls die Währung

typrelevant ist). Bei einer Erweiterung der Geschäftstätigkeit der Bank erleichtert dies die Programmänderung wesentlich. Ebenso ist es vermutlich sinnvoll, den Inhaber des Kontos nicht auf eine Person festzulegen, da ja auch Firmen Konten haben und einzelne Konten mehreren oder vielleicht auch keinem Inhaber mehr zuordenbar sein können. Auch in diesem Fall ist ein Typparameter möglicherweise hilfreich. Von einer Währung oder einem Kontoinhaber erwarten wir ja, dass sie bestimmte Eigenschaften erfüllen. Eine Untertypbeziehung im objektorientierten Sinne ist dabei bei den Währungen nicht notwendig, da ein Konto immer in einer Währung geführt wird. Man sollte sich aber ein Konzept überlegen, welche Operatoren und Methoden die Klassen unterstützen sollten. In C++0x wird das auch direkt mit `concept` in der Sprache ausdrückbar sein. Für den Inhaber ist ein einfacher generischer Typ nicht ausreichend. Hier ist ein Container, welcher verschiedene Typen aufnimmt nötig. Das ist realisierbar über einen Basistyp `Owner`, welcher die Untertypen `Person` und `Company` enthält. Ob `Owner` ein Typparameter sein soll, hängt davon ab, ob man mehrere, verschiedene Hierarchien unterstützen will. Eine einzelne Hierarchie ist auch mit Untertypen erweiterbar – Generizität ist dann nicht notwendig.

Untertyprelationen und Generizität sind manchmal eng miteinander verknüpft, wie das Beispiel zeigt. Eine weitere Parallele zwischen Generizität und Untertypbeziehungen ist erkennbar: Sowohl Generizität als auch Untertypbeziehungen helfen, notwendige Änderungen im Programmcode klein zu halten. Generizität und Untertypbeziehungen ergänzen sich dabei: Generizität ist auch dann hilfreich, um Änderungen von Typen formaler Parameter abzufangen, wenn das Ersetzbarkeitsprinzip nicht erfüllt ist, während Untertypbeziehungen den Ersatz einer Instanz eines Obertyps durch eine Instanz eines Untertyps auch unabhängig von Typen formaler Parameter ermöglichen.

> **Faustregel:** Generizität und Untertyprelationen ergänzen sich. Man soll stets überlegen, ob man eine Aufgabe besser durch Ersetzbarkeit, durch Generizität, oder (häufig sinnvoll) eine Kombination aus beiden Konzepten löst.

Verwendbarkeit. Generizität und Untertypbeziehungen sind oft gegeneinander austauschbar. Das heißt, man kann ein und dieselbe Aufgabe mit Generizität oder über Untertypbeziehungen lösen, ohne dass eine Lösung

der anderen hinsichtlich Wartbarkeit überlegen wäre. Ist es daher vielleicht möglich, dass ein Konzept das andere komplett ersetzen kann? Das ist nicht möglich, wie die folgenden zwei Beispiele zeigen:

Generizität ist sehr gut dafür geeignet, wie in obigen Beispielen eine Listenklasse zu schreiben, wobei eine Instanz nur Elemente eines Typs enthält und eine andere nur Elemente eines anderen Typs. Dabei ist statisch sichergestellt, dass alle Elemente in einer Liste denselben Typ haben. Solche Listen sind *homogen*. Ohne Generizität ist es nicht möglich, eine solche Listenklasse zu schreiben. Zwar kann man auch ohne Generizität Listen erzeugen, die Elemente beliebiger Typen enthalten können, aber es ist nicht statisch sichergestellt, dass alle Elemente in der Liste denselben Typ haben. Daher kann man mit Hilfe von Generizität etwas machen, was ohne Generizität, also beispielsweise nur durch Untertypbeziehungen, nicht machbar wäre.

Mit Generizität ohne Untertypbeziehungen, also auch ohne gebundene Generizität, ist es nicht möglich, eine Listenklasse zu schreiben, in der Elemente unterschiedliche Typen haben können. Solche Listen sind *heterogen*. Daher kann man mit Hilfe von Untertypbeziehungen etwas machen, was ohne Untertypbeziehungen, also nur durch Generizität, nicht machbar wäre. Generizität und Untertypbeziehungen ergänzen sich.

Diese Beispiele zeigen, was man mit Generizität oder Untertypbeziehungen alleine nicht machen kann. Sie zeigen damit auf, in welchen Fällen man Generizität und/oder Untertypbeziehungen zur Erreichung des Ziels unbedingt verwenden muss.

Laufzeiteffizienz. Die Verwendung von Generizität in C++ hat keine negative Auswirkungen auf die Laufzeiteffizienz. Andererseits ist die Verwendung von dynamischem Binden im Zusammenhang mit Untertypbeziehungen immer etwas weniger effizient als statisches Binden. Aufgrund dieser Überlegungen kommen ProgrammiererInnen manchmal auf die Idee, stets Generizität einzusetzen, aber dynamisches Binden nur dort zuzulassen, wo es unumgänglich ist. Da Generizität und Untertypbeziehungen oft gegeneinander austauschbar sind, kann man das im Prinzip machen. Leider sind die tatsächlichen Beziehungen in der relativen Effizienz von Generizität und dynamischem Binden keineswegs so einfach wie hier dargestellt. Durch die Verwendung von Generizität zur Vermeidung von dynamischem Binden ändert sich die Struktur des Programms, wodurch sich die Laufzeiteffizienz wesentlich stärker (eher negativ als positiv)

ändern kann als durch die Vermeidung von dynamischem Binden. Wenn beispielsweise eine switch-Anweisung zusätzlich ausgeführt werden muss, ist die Effizienz ziemlich sicher schlechter geworden.

> **Faustregel:** Man soll Effizienzüberlegungen in der Entscheidung, ob man Generizität oder Untertypbeziehungen einsetzt, beiseite lassen.

Solche Optimierungen auf der untersten Ebene sind wirklich nur etwas für Experten, die Details ihrer Compiler und ihrer Hardware sehr gut kennen, und auch dann sind die Optimierungen meist nicht portabel. Viel wichtiger ist es, auf die Einfachheit und Verständlichkeit des Programms zu achten. Wenn Effizienz entscheidend ist, sollte vor allem die Effizienz der Algorithmen betrachtet werden.

Natürlichkeit. Häufig bekommt man auf die Frage, ob man in einer bestimmten Situation Generizität oder Subtyping einsetzen soll, die Antwort, dass der natürlichere Mechanismus der am besten geeignete sei. Für erfahrene EntwicklerInnen ist diese Antwort durchaus zutreffend: Mit einem gewissen Erfahrungsschatz kommt es ihnen ganz selbstverständlich vor, den richtigen Mechanismus zu wählen, ohne die Entscheidung wirklich begründen zu können. Hinter der Natürlichkeit eines bestimmten Lösungsweges verbirgt sich oft ein großer Erfahrungsschatz. Leider sehen AnfängerInnen kaum, was natürlicher ist. Daher ist der Ratschlag an AnfängerInnen, den natürlicheren Mechanismus zu wählen, mit Vorsicht zu genießen. Es zahlt sich in jedem Fall aus, genau zu überlegen, was man mit Generizität erreichen will und erreichen kann. Wenn man sich zwischen Generizität und Subtyping entscheiden soll, ist es angebracht, auch eine Kombination von Generizität und Subtyping ins Auge zu fassen. Erst wenn diese Überlegungen zu keinem eindeutigen Ziel führen, entscheidet man sich für die natürlichere Alternative.

3.2.2 Arten der Generizität

Bisher haben wir Generizität als ein einziges Sprachkonzept betrachtet. Tatsächlich gibt es zahlreiche Varianten mit unterschiedlichen Eigenschaften. Wir wollen hier einige Varianten miteinander vergleichen.

Für die Übersetzung generischer Klassen und Routinen in ausführbaren Code gibt es zwei Möglichkeiten, die homogene und die heterogene Über-

setzung. In Sprachen wie Java wird eine *homogene* Übersetzung verwendet. Dabei wird jede generische Klasse, genauso wie jede nichtgenerische Klasse auch, in genau eine Klasse übersetzt. In diesem Code werden die Typparameter durch einen allgemeinen Zeiger ersetzt. Dies entspricht der Simulation einiger Aspekte von Generizität. Im Unterschied zur simulierten Generizität wird die Typkompatibilität aber vom Compiler garantiert.

Bei der *heterogenen* Übersetzung wird für jede Verwendung einer generischen Klasse oder Routine mit anderen Typparametern eigener übersetzter Code erzeugt. Die heterogene Übersetzung entspricht also eher der Verwendung von „copy and paste“, wie in Abschnitt 3.1.1 argumentiert, wobei in jeder Kopie alle Vorkommen von Typparametern durch die entsprechenden Typen ersetzt sind. Dem Nachteil einer größeren Anzahl übersetzter Klassen und Routinen stehen einige Vorteile gegenüber: Da für alle Typen eigener Code erzeugt wird, sind einfache Typen wie `int`, `char` oder `bool` problemlos, ohne Einbußen an Laufzeiteffizienz, als Ersatz für Typparameter geeignet. Zur Laufzeit brauchen keine Typumwandlungen und damit zusammenhängende Überprüfungen durchgeführt zu werden. Außerdem sind auf jede übersetzte Klasse eigene Optimierungen anwendbar, die von den Typen abhängen. Daher haben Programme bei heterogener Übersetzung bessere Laufzeiteffizienz. Wie erwartet setzt C++ heterogene Übersetzung ein. Homogene Übersetzung kann in C++ aber simuliert werden indem man in einem Template ein Container für Elemente mit dem Typ `void*` einsetzt. Der Compiler generiert dann nur für die Methoden in der Template-Klasse eigene Instanzen für jeden Typ. Da der tatsächliche Code aber in dem `void*` Container steckt kann ein typsicherer Container, der nur einmal übersetzt wird, implementiert werden:

Listing 3.22: homogen.h

```
#include <vector>

template <typename T>
class Homogen
{
    std::vector<void*> m_elems;
public:
    void push (T const& t)
    {
        m_elems.push_back(new T(t));
    }
    T pop ()
    {
        T* ptr = static_cast<T*>(m_elems.back());
        m_elems.pop_back();
```

```
        T ret = *ptr;
        delete ptr;
        return ret;
    }
};
```

In diesem Programm ist nur der Zugriff und die Typumwandlung heterogen. Der Container selbst verwendet, wie besprochen, immer `void*` und wird deshalb nur einmal instanziert. Da die Methoden `inline` sind, entsteht ein Code, als würde der Programmierer selbst an allen Stellen im Programm die Instanzierung und Typumwandlung durchführen. Allerdings ist die hier vorgestellte Methode im Vergleich dazu wesentlich weniger fehleranfällig, da vom Compiler Typsicherheit garantiert wird. So kann keine Typumwandlung vergessen oder falsch durchgeführt werden.

Durch diese Technik kann der Programmierer entscheiden, ob Performance oder Programmgröße wichtiger ist. Manchmal, beispielsweise in eingebetteten Systemen, ist Speichereffizienz wichtiger als Laufzeiteffizienz. Diese Technik mit homogener Übersetzung des Containers ist in diesen Situationen besser geeignet als eine vollständig heterogene Übersetzung.

Durch die heterogene Übersetzung von Templates brauchen ProgrammiererInnen in C++ keine Schranken (die bei homogener Übersetzung sehr wichtig sind) anzugeben, um Eigenschaften der Typen, die Typparameter ersetzen, verwenden zu können. Es wird einfach für jede übersetzte Klasse getrennt überprüft, ob die Typen alle vorausgesetzten Eigenschaften erfüllen. In dieser Hinsicht ist Generizität mit heterogener Übersetzung flexibler als Generizität mit homogener Übersetzung. Unterschiedliche Typen, die einen Typparameter ersetzen, brauchen keinen gemeinsamen Obertyp haben. Allerdings sind *Fehlermeldungen* sehr ausführlich, da die gesamte Hierarchie von Instanzierungen angegeben wird. Weiß man allerdings nach was man sucht (z.b. `no match for call to` bei g++ wenn ein Operator oder Methode für eine Klasse fehlt) oder unterdrückt man mit entsprechenden Filtern oder Compileroptionen unwichtige Teile, so ist es gut möglich damit zu arbeiten. Mit dem neuen Sprachfeature `concept` sind die Fehlermeldungen dann wie bei normalen Code, wo dieser Nachteil dann aufgehoben wird [GS05].

Eine andere, auch sehr flexible Variante wurde in Ada gewählt. Als Schranke geben ProgrammiererInnen keinen Typ an, sondern Eigenschaften, welche die Typen, die Typparameter ersetzen, erfüllen müssen. Beispielsweise wird explizit angegeben, dass ein Typ eine Routine mit bestimmten Parametern unterstützt. Auf den ersten Blick ist diese Variante genauso flexibel wie Templates in C++. Jedoch kann man in Ada bei jeder

Verwendung einer generischen Einheit getrennt angeben, welche Routine eines Typs für eine bestimmte Eigenschaft verwendet werden soll. Routinen werden wie Typen als generische Parameter behandelt. Generizität in Ada hat aber den Nachteil, dass generische Routinen nicht einfach aufgerufen werden können, sondern zuvor aus der generischen eine nicht-generische Routine erzeugt werden muss. Gründe dafür haben aber eher mit der Philosophie von Ada als mit dem Konzept zur Spezifikation von Schranken zu tun.

(für Interessierte)

Eine generische Funktion in Ada [22, 2] soll zeigen, welche Flexibilität Einschränkungen auf Typparametern bieten können:

```
generic
    type T is private;
    with function "<" (X, Y: T) return Boolean is (<>);
function Max (X, Y: T) return T is
begin
    if X < Y
        then return Y
        else return X
    end if
end Max;
...
function IntMax is new Max (Integer);
function IntMin is new Max (Integer, ">");
```

Die Funktion `Max` hat zwei generische Parameter: den Typparameter T und den Funktionsparameter `<`, dessen Parametertypen mit dem Typparameter in Beziehung stehen. Aufgrund der Klausel „`is (<>)`“ kann der zweite Parameter weggelassen werden. In diesem Fall wird dafür die Funktion namens `<` mit den entsprechenden Parametertypen gewählt, wie in C++. Die Funktion `IntMax` entspricht `Max`, wobei an Stelle von `T` der Typ `Integer` verwendet wird. Als Vergleichsoperator wird der kleiner-Vergleich auf ganzen Zahlen verwendet. In der Funktion `IntMin` ist `T` ebenfalls durch `Integer` ersetzt, zum Vergleich wird aber der größer-Vergleich auf ganzen Zahlen verwendet, so dass von `IntMin` das kleinere Argument zurück gegeben wird. Anders als in C++ ergeben sich die für Typparameter zu verwendenden Typen nicht implizit aus der Verwendung, sondern müssen explizit angegeben werden. Dies entspricht der Philosophie von Ada, wonach alles, was die Bedeutung eines Programms beeinflussen kann, explizit im Programm stehen soll, um die Lesbarkeit zu erhöhen.

3.3 Typabfragen und Typumwandlungen

Prozedurale und funktionale Programmiersprachen unterscheiden streng zwischen Typinformationen im Programm, die nur dem Compiler zum Zeitpunkt der Übersetzung zur Verfügung stehen, und dynamischen Programminformationen, die während der Programmausführung verwendet werden können. Es gibt in diesen Sprachen keine dynamische Typinformation. Im Gegensatz dazu wird in objektorientierten Programmiersprachen dynamische Typinformation für das dynamische Binden zur Ausführungszeit benötigt. Viele objektorientierte Sprachen erlauben den direkten Zugriff darauf. In C++ gibt es zur Laufzeit Möglichkeiten, die Klasse eines Objekts direkt zu erfragen, zu überprüfen, ob ein Objekt Instanz einer bestimmten Klasse ist, sowie zur überprüften Umwandlung des deklarierten Objekttyps. Wir wollen nun den Umgang mit dynamischer Typinformation untersuchen. Nach einer kurzen Einführung zu expliziter Konvertierung im Abschnitt 3.3.1, finden sich im Abschnitt 3.3.2 allgemeine Hinweise zu dynamischer Typumwandlung. Der letzte Abschnitt 3.3.3 behandelt schließlich kovariante Probleme.

3.3.1 Explizite Typkonvertierung

Der Operator `static_cast` konvertiert zwischen zwei verwandten und statisch bekannten Typen. Das kann zwischen einem Gleitkommatyp und einem integralen Typ sein, aber auch zwischen verschiedenen Zeigertypen in einer Klassenhierarchie. Zum Entfernen eines `const` Qualifizierers kann der Operator `const_cast` verwendet werden. Es ist auch möglich einen Speicherbereich komplett anders zu interpretieren. Dies ist naturgemäß selten portabel durchführbar. Dafür kann der Operator `reinterpret_cast` verwendet werden. Grundsätzlich ist vor dem Einsatz einer Typkonvertierung darüber nachdenken ob sie *wirklich* notwendig ist. Der C-Cast `(T)a` sollte überhaupt nicht mehr verwendet werden, da er sehr leicht übersehen werden kann. Dieser kann immer durch eine Kombination der drei C++-Casts ersetzt werden.

3.3.2 Verwendung dynamischer Typinformation

Dynamische Typinformationen, auch RTTI (run-time type information) genannt, werden durch zwei Operatoren abgedeckt: `dynamic_cast` und `typeid`. Werden diese überhaupt nicht verwendet, so kann bei vielen Com-

pilern RTTI komplett abgeschaltet werden und Klassen schleppen diese zusätzliche Information nicht mit. RTTI ist mit polymorphen Typen, das sind Referenzen und Zeiger auf eine Klasse, in der zumindest eine Methode `virtual` ist, verwendbar.

Für dynamische Typumwandlungen wird der Operator `dynamic_cast` verwendet. Dieser Operator ermöglicht es, in der Vererbungshierarchie hinauf (*Upcast*), hinunter (*Downcast*) und zur Seite (*Crosscast*) zu gehen. Dabei werden aber nur gültige Typumwandlungen zugelassen:

```
void f1 (Point *p)
{
    Point3D *p3 = dynamic_cast<Point3D*>(p);
    if (p3 == 0) { /* Fehlerbehandlung */ }
    else { /* Mit p3 arbeiten */}
}
```

Bei diesem *Downcast* wird versucht ein – zum Compilierungszeitpunkt nicht näher festgelegtes – `Point` in ein Objekt vom Untertypen `Point3D` umzuwandeln. Ist `p` tatsächlich ein `Point2D` und kein `Point3D`, so wird `0` zurückgegeben. Ist hingegen `p` ein `Point3D` oder ein Untertyp davon, so wird ein gültiger Zeiger auf die Instanz des richtigen Typs zurückgegeben.

Der Operator `dynamic_cast` kann auch mit Referenzen eingesetzt werden. Dann ist der Fehler aber nicht durch `0` darstellbar. Ein Ausdruck wie `dynamic_cast<T&>(r)` ist eine Zusicherung, dass `r` auf einen Typ T verweist. Kann sie aber nicht gehalten werden, so wird eine `bad_cast`-Ausnahme geworfen:

```
void f2 (Point &p) try
{
    Point3D &p3 = dynamic_cast<Point3D&>(p);
    // mit p3 arbeiten
} catch (bad_cast) {
    // Fehlerbehandlung
}
```

Der Code wird somit robuster und wenn man auf die Fehlerbehandlung vergisst wird das Programm sauber beendet. Allerdings ist für die Frage ob ein Objekt einen bestimmten Typ hat, `dynamic_cast` mit Zeigern besser geeignet.

Die zweite Möglichkeit Typinformationen zu erhalten ist der Operator `typeid` welcher ein Objekt des Typs `type_info` zurückgibt. Diese Objekte können verglichen werden und mit der Methode `name()` kann der Name als C-String abgefragt werden:

```
#include <typeinfo>

void f3 (Point &r, Point *p)
```

```
{
    if (typeid(r) == typeid(*p))
        cout << "Selber Typ" << endl;
    cout << typeid(r).name() << endl;
    cout << typeid(*p).name() << endl;
    cout << typeid(p).name() << endl;
}
```

Wobei die letzte Zeile fast immer einen Fehler darstellt, sie würde den Typ vom Zeiger auf `Point` zurückgeben, und das ist hier immer `Point*`. Bei der Verwendung von `typeid` ist darauf zu achten, dass `<typeinfo>` inkludiert wird.

Dynamische Typabfragen und Typumwandlungen sind sehr mächtige Werkzeuge. Man kann damit einiges machen, was sonst nicht oder nur sehr umständlich machbar wäre. Allerdings kann die Verwendung von dynamischen Typabfragen und Typumwandlungen Fehler in einem Programm verdecken und die Wartbarkeit erschweren. Fehler werden oft dadurch verdeckt, dass der deklarierte Typ einer Variablen oder eines formalen Parameters nur mehr wenig mit dem Typ zu tun hat, dessen Instanzen ProgrammiererInnen als Werte erwarten. Es ist nicht sehr empfehlenswert sehr allgemeine Typen wie `Objekt` zu verwenden. Stattdessen sollte genau der Typ, von welchem Instanzen erwartet werden, verwendet werden. Beispielweise könnten die deklarierten Typen `Person` oder `Schiff` sein. Ist aber beides möglich, werden oftmals dynamische Typabfragen und Typumwandlungen eingesetzt. Wenn in Wirklichkeit statt einer Instanz von `Schiff` oder `Person` eine Instanz von `Point2D` verwendet wird, liefert der Compiler keine Fehlermeldung. Erst zur Laufzeit kann es im günstigsten Fall zu einer Ausnahmebehandlung kommen. Es ist aber auch möglich, dass es zu keiner Ausnahmebehandlung kommt, sondern einfach nur die Ergebnisse falsch sind, oder – noch schlimmer – falsche Daten in einer Datenbank gespeichert werden. Der Grund für das mangelhafte Erkennen dieses Typfehlers liegt darin, dass mit Hilfe von dynamischen Typabfragen und Typumwandlungen keine statischen Typüberprüfungen durch den Compiler durchgeführt werden, obwohl sich ProgrammiererInnen vermutlich nach wie vor auf statische Typsicherheit in C++ verlassen.

Neben der Fehleranfälligkeit ist die schlechte Wartbarkeit ein weiterer Grund, um Typabfragen (auch ohne Typumwandlungen) nur sehr sparsam zu nutzen. Insbesondere gilt dies für geschachtelte Typabfragen:

```
void drehe1 (Form const& r)
{
    if (typeid(r) == typeid(Kreis)); // Tue nichts
    else if (typeid(r) == typeid(Dreieck))
    {
```

```
        // Drehe Dreieck
    } else if (typeid(r) == typeid(Quadrat))
    {
        // Drehe Quadrat
    }
}
```

Dieser Code kann nur unwesentlich mit `dynamic_cast` verbessert werden. Wir wissen bereits aus Abschnitt 2.1, dass `switch`-Anweisungen und (geschachtelte) `if`-Anweisungen durch dynamisches Binden ersetzt werden können. Das soll man auch tun, da dynamisches Binden in der Regel wesentlich wartungsfreundlicher ist. Dasselbe gilt für (geschachtelte) dynamische Typabfragen, welche die möglichen Typen im Programmcode fix verdrahten und daher bei Änderungen der Typhierarchie ebenfalls geändert werden müssen. Oft sind solche (geschachtelte) dynamische Typabfragen einfach durch folgende Möglichkeit ersetzbar:

```
void drehe2 (Form const& x)
{
    x.drehe();
}
```

Die Auswahl des auszuführenden Programmcodes erfolgt hier durch dynamisches Binden. Die Klasse des deklarierten Typs von `x` implementiert `drehe` entsprechend `dreheIrgendeinenTyp`, und die Unterklassen `Kreis`, `Dreieck` und so weiter entsprechend `dreheKreis`, `dreheDreieck` und so weiter.

Manchmal ist es nicht einfach, dynamische Typabfragen durch dynamisches Binden zu ersetzen. Dies trifft vor allem in diesen Fällen zu:

- Der deklarierte Typ von `x` ist zu allgemein; die einzelnen Alternativen decken nicht alle Möglichkeiten ab. Das ist genau die oben erwähnte gefährliche Situation, in der die statische Typsicherheit von C++ umgangen wird. In dieser Situation ist eine Refaktorisierung des Programms angebracht.

- Die Klassen, die dem deklarierten Typ von `x` und dessen Untertypen entsprechen, können nicht erweitert werden. Als (recht aufwändige) Lösung kann man parallel zur unveränderbaren Klassenhierarchie eine gleich strukturierte Hierarchie aufbauen, deren Klassen (Wrapper-Klassen) die zusätzlichen Methoden beschreiben.

- Manchmal ist die Verwendung dynamischen Bindens schwierig, weil die einzelnen Alternativen auf private Variablen und Methoden zugreifen. Methoden anderer Klassen haben diese Information nicht.

Oft lässt sich die fehlende Information durch Übergabe geeigneter Argumente beim Aufruf der Methode oder durch „call backs“ (wenn die Information nur selten benötigt wird) verfügbar machen.

Faustregel: Typabfragen und Typumwandlungen sollen nach Möglichkeit vermieden werden.

In wenigen Fällen ist es nötig und durchaus angebracht, diese mächtigen, aber unsicheren Werkzeuge zu verwenden, wie wir noch sehen werden.

3.3.3 Kovariante Probleme

In Abschnitt 2.1 haben wir gesehen, dass Typen von Eingangsparametern nur kontravariant sein können. Kovariante Eingangsparametertypen verletzen das Ersetzbarkeitsprinzip. In der Praxis wünscht man sich manchmal gerade kovariante Eingangsparametertypen. Entsprechende Aufgabenstellungen nennt man *kovariante Probleme*. Zur Lösung kovarianter Probleme bieten sich dynamische Typabfragen und Typumwandlungen an, wie folgendes Beispiel zeigt:

Listing 3.23: kovarianz2.h

```
#include <iostream>
#include <typeinfo>

struct Futter {virtual ~Futter() {}};
struct Gras: Futter {};
struct Fleisch: Futter {};

struct Tier
{
    virtual void friss (Futter& x) = 0;
    virtual ~Tier() {}
};

struct Rind: Tier
{
    void friss (Gras&) { std::cout << "muh" << std::endl; }
    void friss (Futter& x)
    {
        if (typeid(x) == typeid(Gras))
            friss (dynamic_cast<Gras&>(x));
        else std::cout << "Wahrscheinlichkeit␣BSE++" << std::endl;
    }
};

struct Tiger: Tier
{
    void friss (Fleisch&) { std::cout << "rrrh" << std::endl;}
```

```
    void friss (Futter& x)
    {
        if (typeid(x) == typeid(Fleisch))
            friss (dynamic_cast<Fleisch&>(x));
        else std::cout << "Fletsche Zaehne!!" << std::endl;
    }
};
```

Es ist ganz natürlich, `Gras` und `Fleisch` als Untertypen von `Futter` anzusehen. `Gras` und `Fleisch` sind offensichtlich einander ausschließende Spezialisierungen von `Futter`. Ebenso sind `Rind` und `Tiger` Spezialisierungen von `Tier`. Es entspricht der praktischen Erfahrung, dass Tiere im Allgemeinen Futter fressen, Rinder aber nur Gras und Tiger nur Fleisch. Als Parametertyp der Methode `friss` wünscht man sich daher in `Tier` `Futter`, in `Rind` `Gras` und in `Tiger` `Fleisch`.

Genau diese Beziehungen in der realen Welt sind aber nicht typsicher realisierbar. Zur Lösung des Problems bietet sich eine erweiterte Sicht der Beziehungen in der realen Welt an: Auch einem Rind kann man Fleisch und einem Tiger Gras zum Fressen anbieten. Wenn man das macht, muss man aber mit unerwünschten Reaktionen der Tiere rechnen. Obiges Programmstück beschreibt entsprechendes Verhalten: Wenn dem Tier geeignetes Futter angeboten wird, erledigen die überladenen Methoden `friss` mit den Parametertypen `Gras` beziehungsweise `Fleisch` die Aufgaben. Sonst führen die überschriebenen Methoden `friss` mit dem Parametertyp `Futter` Aktionen aus, die vermutlich nicht erwünscht sind.

Durch Umschreiben des Programms kann man zwar Typabfragen und Typumwandlungen vermeiden, aber die unerwünschten Aktionen bei kovarianten Problemen bleiben erhalten. Die einzige Möglichkeit besteht darin, kovariante Probleme zu vermeiden. Beispielsweise reicht es, `friss` aus `Tier` zu entfernen. Dann kann man zwar `friss` nur mehr mit Futter der richtigen Art in `Rind` und `Tiger` aufrufen, aber man kann Tiere nur mehr füttern, wenn man die Art der Tiere und des Futters genau kennt.

Faustregel: Kovariante Probleme soll man vermeiden.

(für Interessierte)

Kovariante Probleme treten in der Praxis so häufig auf, dass einige Programmiersprachen teilweise Lösungen dafür anbieten. Zunächst betrachten wir Eiffel: In dieser Sprache sind kovariante Eingangsparametertypen durchwegs erlaubt. Wenn die Klasse `Tier` die Methode `friss` mit dem Parametertyp `Futter` enthält, können die überschriebenen Methoden in den Klassen `Rind` und `Tiger` die Parametertypen `Gras` und `Fleisch` haben. Dies ermöglicht eine natürliche Modellierung kovarianter Probleme.

Weil dadurch aber das Ersetzbarkeitsprinzip verletzt ist, können an Stelle dieses Parameters keine Argumente von einem Untertyp des Parametertyps verwendet werden. Der Compiler kann jedoch die Art des Tieres oder die Art des Futters nicht immer statisch feststellen. Wird `friss` mit einer falschen Futterart aufgerufen, kommt es zu einer Ausnahmebehandlung zur Laufzeit. Tatsächlich ergibt sich dadurch derselbe Effekt, als ob man in C++ ohne vorhergehende Überprüfung den Typ des Arguments von `friss` auf die gewünschte Futterart umwandeln würde. Von einer echten Lösung des Problems kann man daher nicht sprechen.

Einen besseren Ansatz scheinen *virtuelle Typen* zu bieten, die derzeit noch in keiner gängigen Programmiersprache verwendet werden [14, 12]. Man kann virtuelle Typen als geschachtelte Klassen wie in C++ ansehen, die jedoch, anders als in C++ , in Unterklassen überschreibbar sind. Die beiden Klassenhierarchien mit `Tier` und `Futter` als Wurzeln werden eng verknüpft: `Futter` ist in `Tier` enthalten. In `Rind` ist `Futter` mit einer neuen Klasse überschrieben, welche die Funktionalität von `Gras` aufweist, und `Futter` in `Tiger` mit einer Klasse der Funktionalität von `Fleisch`. Statt `Gras` und `Fleisch` schreibt man dann `Rind.Futter` und `Tiger.Futter`. Der Typ `Futter` des Parameters von `friss` bezieht sich immer auf den lokal gültigen Namen, in `Rind` also auf `Rind.Futter`. Durch die Verwendung virtueller Typen hat man auf den ersten Blick nichts gewonnen: Noch immer muss man `friss` in `Rind` mit einem Argument vom Typ `Rind.Futter` und in `Tiger` mit einem Argument vom Typ `Tiger.Futter` aufrufen. Die Art des Tieres muss also mit der Art des Futters übereinstimmen, und der Compiler muss die Übereinstimmung überprüfen können. Aber virtuelle Typen haben im Gegensatz zu anderen Ansätzen einen Vorteil: Wenn `Tier` (und daher auch `Rind` und `Tiger`) eine Methode hat, die eine Instanz vom Typ `Futter` als Ergebnis liefert, kann man das Ergebnis eines solchen Methodenaufrufs als Argument eines Aufrufs von `friss` in derselben Instanz verwenden. Dabei braucht man die Art des Tieres nicht zu kennen und ist trotzdem vor Typfehlern sicher. Die Praxisrelevanz dieses Vorteils ist derzeit mangels Erfahrungen kaum abschätzbar.

Einen häufig vorkommenden Spezialfall kovarianter Probleme stellen binäre Methoden dar. Wie in Abschnitt 2.1 eingeführt, hat eine binäre Methode mindestens einen formalen Parameter, dessen Typ stets gleich der Klasse ist, die die Methode enthält. Im Prinzip kann man binäre Methoden auf dieselbe Weise behandeln wie alle anderen kovarianten Probleme. Das heißt, man könnte (wie in Abschnitt 3.3.2) dynamische Typabfragen mittels `typeid` verwenden, um den dynamischen Parametertyp zu bestimmen. Das ist aber problematisch, wie wir gleich sehen werden. Hier ist eine weitere, bessere Lösung für den binären Operator `operator==` in `Point2D` und `Point3D`:

Listing 3.24: kovarinaz3.h

```
class Point
{
```

```
public:
    bool operator == (Point const& p) const
    {
        return equal(p);
    }
    virtual bool equal (Point const& p) const = 0;
    virtual ~Point() {}
};


class Point2D: public Point
{
private:
    int x,y;
public:
    Point2D (int x1, int y1): x(x1), y(y1) {}
    virtual bool equal (Point const& p) const
    {
        Point2D const& that = dynamic_cast<Point2D const&>(p);
        return (x==that.x && y==that.y);
    }
};

class Point3D: public Point
{
private:
    int x,y,z;
public:
    Point3D (int x1, int y1, int z1): x(x1), y(y1), z(z1) {}
    virtual bool equal (Point const& p) const
    {
        Point3D const& that = dynamic_cast<Point3D const&>(p);
        return (x==that.x && y==that.y && z==that.z);
    }
};
```

Anders als in der vorhergehenden Lösung ist `Point3D` kein Untertyp von `Point2D`, sondern sowohl `Point3D` als auch `Point2D` sind von einer gemeinsamen abstrakten Oberklasse `Point` abgeleitet. Dieser Unterschied hat nichts direkt mit binären Methoden zu tun, sondern verdeutlicht, dass `Point3D` keine Spezialisierung von `Point2D` ist. Der Operator `operator==` ist in `Point` definiert und kann in Unterklassen nicht überschrieben werden. Wenn die beiden zu vergleichenden Punkte genau den gleichen Typ haben, wird in der betreffenden Unterklasse von `Point` die Methode `equal` aufgerufen, die den eigentlichen Vergleich durchführt.

(für Interessierte)

Die Programmiersprache Ada unterstützt binäre Methoden direkt: Alle Parameter, die denselben Typ wie das Äquivalent zu `this` in C++ haben, werden beim Überschreiben auf die gleiche Weise kovariant verändert. Wenn mehrere Parameter denselben überschriebenen Typ haben, handelt es sich um binäre Methoden. Eine Regel in Ada besagt, dass alle Argumente, die für diese Parameter eingesetzt werden, genau den

gleichen dynamischen Typ haben müssen. Das wird zur Laufzeit überprüft. Schlägt die Überprüfung fehl, wird eine Ausnahmebehandlung eingeleitet. Methoden wie `equal` in obigem Beispiel sind damit sehr einfach programmierbar. Falls die zu vergleichenden Objekte unterschiedliche Typen haben, kommt es zu einer Ausnahmebehandlung, die an geeigneten Stellen abgefangen werden kann.

3.4 Überladen versus Multimethoden

Dynamisches Binden erfolgt in C++ (wie in vielen anderen objektorientierten Programmiersprachen auch) über den dynamischen Typ eines speziellen Parameters. Beispielsweise wird die auszuführende Methode in `x.equal(y)` durch den dynamischen Typ von `x` festgelegt. Der dynamische Typ von `y` ist für die Methodenauswahl irrelevant. Aber der deklarierte Typ von `y` ist bei der Methodenauswahl relevant, wenn `equal` überladen ist. Bereits der Compiler kann an Hand des deklarierten Typs von `y` auswählen, welche der überladenen Methoden auszuführen ist. Für das dynamische Binden ist `y` unerheblich.

Generell, aber nicht in C++ , ist es möglich, dass dynamisches Binden auch den dynamischen Typ von `y` in die Methodenauswahl einbezieht. Dann legt nicht bereits der Compiler anhand des deklarierten Typs fest, welche überladene Methode auszuwählen ist, sondern erst zur Laufzeit des Programms wird die auszuführende Methode durch die dynamischen Typen von `x` und `y` bestimmt. In diesem Fall spricht man nicht von Überladen sondern von *Multimethoden* [7].

Leider haben C++-ProgrammiererInnen immer wieder Probleme damit, klar zwischen Überladen und Multimethoden zu unterscheiden. Das kann zu Fehlern führen. In Abschnitt 3.4.1 werden wir die Unterschiede zwischen Überladen und Multimethoden klar machen. In Abschnitt 3.4.2 werden wir sehen, dass man Multimethoden auch in Sprachen wie C++ recht einfach simulieren kann.

3.4.1 Unterschiede zwischen Überladen und Multimethoden

Folgendes Beispiel soll vor Augen führen, dass bei der Auswahl zwischen überladenen Methoden in C++ nur der deklarierte Typ eines Arguments entscheidend ist, nicht der dynamische Typ. Wir verwenden das Beispiel zu kovarianten Problemen aus Abschnitt 3.3.3:

```
void futterzeit (Rind& rind, Gras& gras)
```

```
{
    Futter& futter = static_cast<Futter&>(gras);
    rind.friss (futter); // Rind.friss (Futter& x)
    rind.friss (gras);   // Rind.friss (Gras& x)
}
```

Wegen dynamischen Bindens werden die Methoden `friss` auf jeden Fall in der Klasse `Rind` ausgeführt, unabhängig davon, ob `rind` als `Tier` oder `Rind` deklariert ist. Der Methodenaufruf in der zweiten Zeile führt die überladene Methode mit dem Parameter vom Typ `Futter` aus, da `futter` mit dem Typ `Futter` deklariert ist. Für die Methodenauswahl ist es unerheblich, dass `futter` tatsächlich eine Instanz von `Gras` enthält; es zählt nur der deklarierte Typ. Der Methodenaufruf in der dritten Zeile hingegen führt die überladene Methode mit dem Parameter vom Typ `Gras` aus, weil der deklarierte Typ von `gras` wegen der Typumwandlung an dieser Stelle `Gras` ist. Typumwandlungen ändern ja den deklarierten Typ eines Ausdrucks.

Häufig wissen ProgrammiererInnen in solchen Fällen, dass `futter` eine Instanz von `Gras` enthält, und nehmen an, dass die Methode mit dem Parameter vom Typ `Gras` gewählt wird. Diese Annahme ist aber falsch! Man muss stets auf den deklarierten Typ achten, auch wenn man den dynamischen Typ kennt.

Was wäre wenn der Parameter `Rind& rind` stattdessen `Tier& rind` lauten würde? Wegen dynamischen Bindens würde `friss` natürlich weiterhin in `Rind` ausgeführt werden. Aber zur Auswahl überladener Methoden kann der Compiler nur deklarierte Typen verwenden. Das gilt auch für den Empfänger einer Nachricht. Die überladenen Methoden werden in `Tier` gesucht, nicht in `Rind`. In `Tier` ist `friss` nicht überladen, sondern es gibt nur eine Methode mit einem Parameter vom Typ `Futter`. Daher wird in `Rind` auf jeden Fall die Methode mit dem Parameter vom Typ `Futter` ausgeführt, unabhängig davon, ob der deklarierte Typ des Arguments `Futter` oder `Gras` ist. Wie das Beispiel zeigt, kann sich die Auswahl zwischen überladenen Methoden stark von der Intuition vieler ProgrammiererInnen unterscheiden. Daher ist hier besondere Vorsicht geboten.

Die Methoden `friss` in `Rind` und `Tiger` sind so überladen, dass es (außer für die Laufzeiteffizienz) keine Rolle spielt, welche der überladenen Methoden aufgerufen wird. Wenn der dynamische Typ des Arguments `Gras` ist, wird im Endeffekt immer die Methode mit dem Parametertyp `Gras` aufgerufen. Es ist empfehlenswert, Überladen nur so zu verwenden.

> **Faustregel:** Man soll Überladen nur so verwenden, dass es keine Rolle spielt, ob bei der Methodenauswahl deklarierte oder dynamische Typen der Argumente verwendet werden.

Für je zwei überladene Methoden gleicher Parameteranzahl

- soll es zumindest eine Parameterposition geben, an der sich die Typen der Parameter unterscheiden, nicht in Untertyprelation zueinander stehen und auch keinen gemeinsamen Untertyp haben,
- oder alle Parametertypen der einen Methode sollen Obertypen der Parametertypen der anderen Methode sein, und bei Aufruf der einen Methode soll nichts anderes gemacht werden, als auf die andere Methode zu verzweigen, falls die entsprechenden dynamischen Typen der Argumente dies erlauben.

Unter diesen Bedingungen ist die strikte Unterscheidung zwischen deklarierten und dynamischen Typen bei der Methodenauswahl nicht wichtig.

Das Problem mit der häufigen Verwechslung von dynamischen und deklarierten Typen könnte man auch nachhaltig lösen, indem man zur Methodenauswahl generell die dynamischen Typen aller Argumente verwendet. Statt überladener Methoden hätte man dann Multimethoden. Unter der Annahme, dass C++ Multimethoden unterstützt, könnte man die Klasse `Rind` im Beispiel aus Abschnitt 3.3.3 kürzer und ohne dynamische Typabfragen und Typumwandlungen schreiben:

```
class Rind: public Tier
{
public:
    multi virtual void friss (Gras& x)
    {
        std::cout << "muh" << std::endl;
    }
    multi virtual void friss (Futter& x)
    {
        std::cout << "Wahrscheinlichkeit␣BSE++" << std::endl;
    }
};
```

Die Typabfrage, ob `x` den dynamischen Typ `Gras` hat, hätte man sich erspart, da `friss` mit dem Parametertyp `Futter` bei Multimethoden nur aufgerufen wird, wenn der dynamische Typ des Arguments ungleich `Gras` ist. C++ unterstützt aber keine Multimethoden.

Als Grund für die fehlende Unterstützung von Multimethoden in vielen heute üblichen Programmiersprachen wird häufig die höhere Komplexität

der Methodenauswahl genannt. Der dynamische Typ der Argumente muss ja zur Laufzeit in die Methodenauswahl einbezogen werden. Im Beispiel mit der Multimethode `friss` ist jedoch, wie in vielen Fällen, in denen Multimethoden sinnvoll sind, kein zusätzlicher Aufwand nötig; eine dynamische Typabfrage auf dem Argument ist immer nötig, wenn der statische Typ kein Untertyp von `Gras` ist. Die Multimethodenvariante von `friss` kann sogar effizienter sein als die Variante mit Überladen, wenn der statische Typ des Arguments ein Untertyp von `Gras` ist, nicht jedoch der deklarierte Typ. Die Laufzeiteffizienz ist daher kaum ein Grund für fehlende Multimethoden in einer Programmiersprache.

Unter der höheren Komplexität der Methodenauswahl von Multimethoden versteht man oft etwas anderes als die damit verbundene Laufzeiteffizienz: Für ProgrammiererInnen ist nicht gleich erkennbar, unter welchen Bedingungen welche Methode ausgeführt wird. Eine allgemeine Regel besagt, dass immer jene Methode mit den speziellsten Parametertypen, die mit den dynamischen Typen der Argumente kompatibel sind, auszuführen ist. Wenn wir `friss` mit einem Argument vom Typ `Gras` (oder einem Untertyp davon) aufrufen, sind die Parametertypen beider Methoden mit dem Argumenttyp kompatibel. Da `Gras` spezieller ist als `Futter`, wird die Methode mit dem Parametertyp `Gras` ausgeführt. Diese Regel ist für die Methodenauswahl aber nicht hinreichend, wenn Multimethoden mehrere Parameter haben, wie folgendes Beispiel zeigt:

```
multi virtual void frissDoppelt (Futter& x, Gras& y) { ... }
multi virtual void frissDoppelt (Gras& x, Futter& y) { ... }
```

Mit einem Aufruf von `frissDoppelt` mit zwei Argumenten vom Typ `Gras` sind beide Methoden kompatibel. Aber keine Methode ist spezieller als die andere. Es gibt mehrere Möglichkeiten, mit solchen Mehrdeutigkeiten umzugehen. Eine Möglichkeit besteht darin, die erste passende Methode zu wählen; das wäre die Methode in der ersten Zeile. Es ist auch möglich, die Übereinstimmung zwischen Parametertyp und Argumenttyp für jede Parameterposition getrennt zu prüfen, und dabei von links nach rechts jeweils die Methode mit den spezielleren Parametertypen zu wählen; das wäre die Methode in der zweiten Zeile. CLOS (Common Lisp Object System [13]) bietet zahlreiche weitere Auswahlmöglichkeiten. Keine dieser Möglichkeiten bietet klare Vorteile gegenüber der anderen. Daher scheint eine weitere Variante günstig zu sein: Der Compiler verlangt, dass es immer genau eine eindeutige speziellste Methode gibt. ProgrammiererInnen müssen eine weitere Methode

```
multi virtual void frissDoppelt (Gras& x, Gras& y) { ... }
```

hinzufügen, die das Auswahlproblem beseitigt. Dieses Beispiel soll klar machen, dass Multimethoden im Allgemeinen tatsächlich sowohl für den Compiler als auch für ProgrammiererInnen eine deutlich höhere Komplexität haben als überladene Methoden. In den üblichen Anwendungsbeispielen haben Multimethoden keine höhere Komplexität als überladene Methoden. Die Frage, ob ProgrammiererInnen eher Multimethoden oder eher Überladen haben wollen, bleibt offen.

3.4.2 Simulation von Multimethoden

Multimethoden verwenden mehrfaches dynamisches Binden: Die auszuführende Methode wird dynamisch durch die Typen mehrerer Argumente bestimmt. In C++ gibt es nur einfaches dynamisches Binden. Trotzdem ist es nicht schwer, mehrfaches dynamisches Binden durch wiederholtes einfaches Binden zu simulieren. Wir nutzen mehrfaches dynamisches Binden für das Beispiel aus Abschnitt 3.3.3 und eliminieren damit dynamische Typabfragen und Typumwandlungen:

Listing 3.25: multimeth.h

```
#include <iostream>
#include <typeinfo>

struct Rind;
struct Tiger;

struct Futter
{
    virtual void vonRindGefressen(Rind& r) = 0;
    virtual void vonTigerGefressen(Tiger& t) = 0;
    virtual ~Futter() {}
};

struct Gras: Futter
{
    virtual void vonRindGefressen(Rind& r)
    {
        std::cout << "muh" << std::endl;
    }
    virtual void vonTigerGefressen(Tiger& t)
    {
        std::cout << "Fletsche Zaehne!!" << std::endl;
    }
};

struct Fleisch: Futter
{
    virtual void vonRindGefressen(Rind& r)
    {
        std::cout << "Wahrscheinlichkeit BSE++" << std::endl;
```

```
    }
    virtual void vonTigerGefressen(Tiger& t)
    {
        std::cout << "rrrh" << std::endl;
    }
};

struct Tier
{
    virtual void friss (Futter& futter) = 0;
    virtual ~Tier() {}
};

struct Rind: Tier
{
    void friss (Futter& futter)
    {
        futter.vonRindGefressen(*this);
    }
};

struct Tiger: Tier
{
    void friss (Futter& futter)
    {
        futter.vonTigerGefressen(*this);
    }
};
```

Die Methoden `friss` in `Rind` und `Tiger` rufen Methoden in `Futter` auf, die die eigentlichen Aufgaben durchführen. Scheinbar verlagern wir die Arbeit nur von den Tieren zu den Futterarten. Dabei passiert aber etwas Wesentliches: In `Gras` und `Fleisch` gibt es nicht nur *eine* entsprechende Methode, sondern je eine für Instanzen von `Rind` und `Tiger`. Bei einem Aufruf von `tier.friss(futter)` wird zweimal dynamisch gebunden. Das erste dynamische Binden unterscheidet zwischen Instanzen von `Rind` und `Tiger`. Diese Unterscheidung spiegelt sich im Aufruf von `vonRindGefressen` und `vonTigerGefressen` wider. Ein zweites dynamisches Binden unterscheidet zwischen Instanzen von `Gras` und `Fleisch`. In den Unterklassen von `Futter` sind insgesamt vier Methoden implementiert, die alle möglichen Kombinationen von Tierarten mit Futterarten darstellen.

Die Namen der Methoden `vonRindGefressen` und `vonTigerGefressen` sind beliebig wählbar. Wegen der Möglichkeit des Überladens hätten wir für diese Methoden auch denselben Namen wählen können, da sie sich durch die Typen der formalen Parameter eindeutig unterscheiden.

Stellen wir uns vor, diese Lösung des Beispiels sei dadurch zu Stande gekommen, dass wir eine ursprüngliche Lösung mit Multimethoden in C++ implementiert und dabei für den formalen Parameter einen zusätz-

lichen Schritt dynamischen Bindens eingeführt hätten. Damit wird klar, wie man mehrfaches dynamisches Binden durch wiederholtes einfaches dynamisches Binden ersetzen kann. Bei Multimethoden mit mehreren Parametern muss entsprechend oft dynamisch gebunden werden. Sobald man den Übersetzungsschritt verstanden hat, kann man ihn ohne große intellektuelle Anstrengungen für vielfaches dynamisches Binden durchführen.

Diese Lösung kann auch dadurch erzeugt worden sein, dass in der ursprünglichen Lösung aus Abschnitt 3.3.3 `if`-Anweisungen mit dynamischen Typabfragen durch dynamisches Binden ersetzt wurden. Nebenbei sind auch die Typumwandlungen verschwunden. Auch diese Umformung ist automatisch durchführbar. Wir haben damit die Möglichkeit, dynamische Typabfragen genauso wie Multimethoden aus Programmen zu entfernen und damit die Struktur des Programms zu verbessern.

Mehrfaches dynamisches Binden wird in der Praxis häufig benötigt. Die Lösung wie in unserem Beispiel entspricht dem *Visitor Pattern*, einem klassischen Entwurfsmuster. Klassen wie `Futter` werden Visitorklassen genannt, und Klassen wie `Tier` heißen Elementklassen. Visitor- und Elementklassen sind oft gegeneinander austauschbar. Beispielsweise könnten die eigentlichen Implementierungen in den Tier-Klassen stehen, die nur in den Futter-Klassen aufgerufen werden.

Das Visitor Pattern hat einen großen Nachteil: Die Anzahl der benötigten Methoden wird schnell sehr groß. Nehmen wir an, wir hätten M unterschiedliche Tierarten und N Futterarten. Zusätzlich zu den M Visitor-Methoden werden $M \cdot N$ inhaltliche Methoden benötigt. Noch rascher steigt die Methodenanzahl mit der Anzahl der dynamischen Bindungen. Bei $n \geq 2$ dynamischen Bindungen mit N_i Möglichkeiten für die $i$te Bindung ($i = 1 \ldots n$) werden $N_1 \cdot N_2 \cdots N_n$ inhaltliche Methoden und $N_1 + N_1 \cdot N_2 + \cdots + N_1 \cdot N_2 \cdots N_{n-1}$ Visitor-Methoden benötigt, insgesamt also sehr viele. Für $n = 4$ und $N_1, \ldots, N_4 = 10$ kommen wir bereits auf 11.110 Methoden. Außer für sehr kleine n und kleine N_i ist diese Technik nicht sinnvoll einsetzbar. Durch Vererbung lässt sich die Zahl der nötigen Methoden nur unwesentlich reduzieren.

Lösungen mit Multimethoden oder dynamischen Typabfragen haben manchmal einen großen Vorteil gegenüber Lösungen mit dem Visitor Pattern: Die Anzahl der nötigen Methoden bleibt kleiner. Dies trifft besonders dann zu, wenn die Multimethode aus einigen speziellen Methoden mit uneinheitlicher Struktur der formalen Parametertypen und ganz wenigen allgemeinen Methoden, die den großen Rest behandeln, auskommt. Bei Verwendung dynamischer Typabfragen ist in diesen Fällen der große

Rest in wenigen `else`-Zweigen versteckt.

Zudem können wie in [Ale02] beschrieben, die dynamischen Typanfragen mit Templates generiert werden. Dem Programmierer bleibt dann noch die Aufgabe, eine Klasse mit Methoden aller Kombinationen von Parameter zu implementieren. Diese Vorgehensweise ist schon sehr ähnlich dem direkten Programmieren mit Multimethoden.

3.5 Ausnahmebehandlung

Ausnahmebehandlungen dienen vor allem dem Umgang mit unerwünschten Programmzuständen. Zum Beispiel werden in C++ Ausnahmebehandlungen ausgelöst, wenn das Objekt bei einer Typumwandlung keine Instanz des gegebenen Typs ist, oder kein Speicher mehr verfügbar ist. In diesen Fällen kann der Programmablauf nicht normal fortgeführt werden, da grundlegende Annahmen verletzt sind. Oftmals ist es aber nicht so klar was eine Ausnahmesituation tatsächlich ist. Was für einen eine Ausnahme ist, kann für jemand anderen erwartet sein.

> **Faustregel:** Verwende immer Ausnahmen wenn Semantik- und Performance- Charakteristiken von Ausnahmen die beste Wahl darstellen.

Ausnahmebehandlungen geben ProgrammiererInnen die Möglichkeit, das Programm auch in solchen Situationen noch weiter ablaufen zu lassen. In Unterabschnitt 3.5.1 gehen wir auf Ausnahmebehandlungen in C++ ein und geben danach in Unterabschnitt 3.5.2 einige Hinweise auf den sinnvollen Einsatz von Ausnahmebehandlungen.

3.5.1 Ausnahmebehandlung in C++

Ausnahmen sind in C++ gewöhnliche Objekte, die über spezielle Mechanismen als Ausnahmen verwendet werden. Obwohl beliebiges, wie `int` oder `std::string` geworfen werden kann, wird empfohlen nur Basisklassen von `std::exception` zu verwenden.

Sie können als alternativer Rückgabewert betrachtet werden, sind aber im Gegensatz dazu nicht lokal. Ausnahmen sollten gruppiert werden, damit nicht auf eine Ausnahme vergessen wird. Der Mechanismus dafür sind Untertypbeziehungen. Durch Mehrfachvererbung können auch zusammengesetzte Ausnahmen abgebildet werden.

> **Tipp:** Benutzen Sie Ausnahmen zur Fehlerbehandlung nur wenn lokale Strukturen nicht ausreichen.

Die Standardlibrary definiert in unterschiedlichen Headern bereits die wichtigsten Fehler, die auftreten können. Die Ausnahmen `bad_cast` und `bad_typeid` wurden bereits in Kapitel 3.3.2 besprochen. Sie treten auf, wenn eine Referenz nicht in einen anderen Typ umgewandelt werden kann. `bad_alloc` spielt nur in sehr robusten und gut getesteten Bibliotheken eine Rolle - für Anwendungsprogramme ist es bei Speichermangel am sinnvollsten zu terminieren. Der Aufwand, der notwendig ist, dass in diesen Situationen das Programm noch weiterhin funktioniert, rechnet sich hier meistens nicht. Die Ausnahme `out_of_range` hingegen kann durchaus sinnvoll in Anwendungsprogrammen verwendet werden. Wenn beispielsweise auf einen, vom Benutzer vorgegebenen, Arrayindex zugegriffen werden soll, kann die Methode `at` statt dem Operator `operator[]` verwendet werden. Dadurch kann undefiniertes Verhalten vermieden werden, welches auf jeden Fall unerwünscht ist. An einer zentralen Stelle können dann die Ausnahmen gefangen werden und den Benutzer informieren, dass seine Angabe nicht innerhalb des Bereiches liegt. In Situationen wo allerdings geklärt ist, dass der Zugriff legitim ist, sollte aber der Operator `operator[]` verwendet werden um die zusätzliche Abfrage in `at` zu vermeiden. Die Klasse `out_of_range` ist ein Untertyp von `logic_error`, welche prinzipiell schon vor dem Programmstart durch Überprüfen aller Argumente und Benutzereingaben vermieden werden könnten. Des weiteren sind `length_error`, `domain_error` und `invalid_argument` in dieser Gruppe, die für eigene Fehler dieser Art verwendet werden können.

Die andere Art der Fehler wie sie von der Standardlibrary gegliedert werden sind Laufzeitfehler (`runtime_error`). Hier gibt es die Repräsentanten `range_error`, `overflow_error` und `underflow_error` welche alle nicht von C++ oder der Standardlibrary geworfen werden. Die Unterscheidung zwischen diesen wenigen Arten von Fehlern reicht in der Praxis kaum aus um Fehler sinnvoll abbilden zu können. Deshalb werden weitere domain-spezifische Ausnahmen in eigenen Klassen geschrieben. Diese sollten aber immer zumindest von `std::exception`, wenn sinnvoll aber konkreter, abgeleitet sein.

Ausnahmen können jederzeit geworfen werden:

Listing 3.26: throw1.cpp

```
#include "throw1.h"
```

```
#include <iostream>

using namespace std;

bool helpNeeded() {return true;}

int main() try
{
    if (helpNeeded()) throw help();
    cout << "wird nicht angezeigt" << endl;
} catch (help const& h)
{
    cout << h.what() << endl;
}
```

Die Definition der Ausnahme sieht so aus:

Listing 3.27: throw1.h

```
#include <exception>

struct help : std::exception
{
    char const* what() const throw()
    {
        return "Help really needed!";
    }
};
```

Zum Abfangen von Ausnahmen gibt es `try-catch`-Blöcke:

```
try { ... }
catch (help const& e) { ... }
catch (exception const& e) { ... }
```

Im Block nach dem Wort `try` stehen beliebige Anweisungen, die ausgeführt werden, wenn der `try-catch`-Block ausgeführt wird. Falls während der Ausführung dieses `try`-Blocks eine Ausnahme auftritt, wird eine passende `catch`-Klausel nach dem `try`-Block gesucht. Jede `catch`-Klausel enthält nach dem Schlüsselwort `catch` (wie eine Methode mit einem Parameter) genau einen formalen Parameter. Ist die aufgetretene Ausnahme eine Instanz des Parametertyps, dann kann die `catch`-Klausel die Ausnahme abfangen. Das bedeutet, dass die Abarbeitung der Befehle im `try`-Block nach Auftreten der Ausnahme endet, dafür aber die Befehle im Block der `catch`-Klausel ausgeführt werden. Im Beispiel können beide `catch`-Klauseln eine Ausnahme vom Typ `help` abfangen, da jede Instanz von `help` auch eine Instanz von `std::exception` ist. Wenn es mehrere passende `catch`-Klauseln gibt, wird einfach die erste passende gewählt. Nach einer abgefangenen Ausnahme wird das Programm so fortgesetzt, als ob es gar keine Ausnahmebehandlung gegeben hätte. Das heißt, nach

der `catch`-Klausel wird der erste Befehl nach dem `try-catch`-Block ausgeführt.

In Konstruktoren können Ausnahmen geworfen werden, wenn es nicht möglich ist das Objekt fertig zu konstruieren. Destruktoren sollten aber auf jeden Fall keine Ausnahme werfen. Um einen Fehler beim Freigeben der Ressourcen gemeldet zu bekommen kann eine eigene Methode wie `close` verwendet werden. Der Destruktor muss dann aber geschützt sein:

```
struct File
{
    FILE *f;
    File(const char * name, const char * mode = "r")
    {
        f = fopen(name, mode);
        if (!f) throw could_not_open();
    }
    void close()
    {
        if (fclose(f) == EOF) throw could_not_close();
    }
    ~File() try
    {
        close();
    } catch (could_not_close const& e)
    {
        // Verschlucke Ausnahme
    }
};
```

Welche Ausnahmen geworfen werden haben einen direkten Einfluss auf das Ersetzbarkeitsprinzip. Deshalb kann mittels einer *Ausnahmespezifikation* welche Ausnahmen geworfen werden in die Signatur von Funktionen und Methoden aufgenommen werden. Wird die Ausnahmespezifikation weggelassen, so kann jede Ausnahme geworfen werden. Diese Designentscheidung wurde deshalb so getroffen um ProgrammierInnen nicht zum Erstellen von Mogelcode, der nur eine falsche Vorstellung von Sicherheit bringt, zu ermutigen. Bei einer leeren Liste wird zugesichert, dass keine Ausnahme geworfen wird:

```
void f() throw();
```

Eine virtuelle Funktion kann nur dann überschrieben werden, wenn sie mindestens so restriktiv wie die eigene Ausnahmespezifikation ist. Allerdings gehört die Ausnahmespezifikation nicht zu dem Typ einer Methode oder Funktion.

Es kann nicht vorausgesetzt werden, dass jede Ausnahme von `std::exception` abgeleitet ist. Um alle Ausnahmen zu fangen gibt es deshalb folgendes Konstrukt:

```
try {}
catch(...) {}
```

Wenn während der Fehlerbehandlung entschieden wird, das der Fehler doch nicht an dieser Stelle behoben werden kann, so kann die Ausnahme weiter geworfen werden:

```
try {}
catch(...)
{
    if (!rescue())
    { // weiterwerfen
        throw;
    }
}
```

Da es nicht klar ist, an welcher Stelle in einem `try` Block eine Ausnahme geworfen wird, ist ein sogenannter *ausnahmefester Code* sehr wichtig. Dies wird am besten mit RAII, z.B. `auto_ptr`, umgesetzt. Dadurch kann sehr einfach garantiert werden, dass auch alle Ressourcen eines Blockes oder Funktion wieder freigegeben werden:

```
void f (bool cond1, bool cond2) try
{
    Ressource r1;
    Ressource r2;
    if (cond1) throw error1();
    Ressource r3;
    if (cond2) return;
    Ressource r4;
}
catch (error1 const& e1) {}
```

Egal ob cond1 oder cond2 wahr ist oder ob ein Konstruktor einer Ressource eine Ausnahme wirft, es werden immer alle bereits konstruierten Ressourcen wieder freigegeben bevor der `catch` Block angesprungen wird. Und nicht nur das, es wird sogar garantiert dass die Ressourcen wieder in umgekehrter Reihenfolge abgebaut werden.

Die Suche nach einen passenden `try catch` Block erfolgt rekursiv. In jedem Schritt wird ein Block verlassen, wobei der Stack aber vorher aufgeräumt und, wie zuvor festgestellt, alle Destruktoren aufgerufen werden. Diesen Vorgang, in dem der Stack wieder abgebaut wird, nennt man *Stack-Abwicklung* (stack unwind). Spätestens der Code welcher `main` aufruft fängt die Ausnahme ab und beendet standardmäßig das Programm. In der Übung müssen aber alle Ausnahmen selber abgefangen werden, es ist demnach darauf zu achten, dass Ausnahmen spätestens in `main` abgefangen werden.

3.5.2 Einsatz von Ausnahmebehandlungen

Ausnahmen werden in folgenden Fällen eingesetzt:

Unvorhergesehene Programmabbrüche: Wird eine Ausnahme nicht abgefangen, kommt es zu einem Programmabbruch. Die entsprechende Bildschirmausgabe enthält genaue Informationen über Art und Ort des Auftretens der Ausnahme. Damit lassen sich die Ursachen von Programmfehlern leichter finden.

Kontrolliertes Wiederaufsetzen: Nach aufgetretenen Fehlern oder in außergewöhnlichen Situationen wird das Programm an genau definierbaren Punkten weiter ausgeführt. Während der Programmentwicklung ist es vielleicht sinnvoll, einen Programmlauf beim Auftreten eines Fehlers abzubrechen, aber im praktischen Einsatz soll das Programm auch dann noch funktionieren, wenn ein Fehler aufgetreten ist. Ausnahmebehandlungen wurden vor allem zu diesem Zweck eingeführt: Man kann einen Punkt festlegen, an dem es auf alle Fälle weiter geht. Leider können Ausnahmebehandlungen echte Programmfehler nicht beheben, sondern nur den Benutzer darüber informieren und dann das Programm abbrechen, oder weiterhin (eingeschränkte) Dienste anbieten. Ergebnisse bereits erfolgter Berechnungen gehen dabei oft verloren.

Ausstieg aus Sprachkonstrukten: Ausnahmen sind nicht auf den Umgang mit Programmfehlern beschränkt. Sie erlauben ganz allgemein das vorzeitige Abbrechen der Ausführung von Blöcken, Kontrollstrukturen, Methoden, etc. in außergewöhnlichen Situationen. Das Auftreten solcher Ausnahmen wird von ProgrammiererInnen erwartet (im Gegensatz zum Auftreten von bestimmten Fehlern). Es ist daher relativ leicht, entsprechende Ausnahmebehandlungen durchzuführen, die eine sinnvolle Weiterführung des Programms erlauben.

Rückgabe alternativer Ergebniswerte: In C++ und vielen anderen Sprachen kann eine Methode nur Ergebnisse eines bestimmten Typs liefern. Wenn in der Methode eine unbehandelte Ausnahme auftritt, wird an den Aufrufer statt eines Ergebnisses die Ausnahme zurückgegeben, die er abfangen kann. Damit ist es möglich, dass die Methode an den Aufrufer in Ausnahmesituationen Objekte zurückgibt, die nicht den deklarierten Ergebnistyp der Methode haben. Alternativ dazu kann der Ergebnistyp auch vom Typ `boost::variant`

sein. Dadurch ist es möglich, dass der Aufrufer Zugriff auf eine von mehreren Variablen unterschiedlichen Typs hat.

Die ersten zwei Punkte beziehen sich auf fehlerhafte Programmzustände, die durch Ausnahmen möglichst eingegrenzt werden. ProgrammiererInnen wollen solche Situationen vermeiden. Es gelingt ihnen nicht immer. Die letzten beiden Punkte beziehen sich auf Situationen, in denen Ausnahmen und Ausnahmebehandlungen von ProgrammiererInnen gezielt eingesetzt werden, um den üblichen Programmfluss abzukürzen oder Einschränkungen des Typsystems zu umgehen. Im Folgenden wollen wir uns den bewussten Einsatz von Ausnahmen genauer vor Augen führen.

Faustregel: Aus Gründen der Wartbarkeit soll man Ausnahmen und Ausnahmebehandlungen nur in echten Ausnahmesituationen und sparsam einsetzen.

Bei Auftreten einer Ausnahme wird der normale Programmfluss durch eine Ausnahmebehandlung ersetzt. Während der normale Programmfluss lokal sichtbar und durch Verwendung strukturierter Sprachkonzepte wie Schleifen und bedingte Anweisungen relativ einfach nachvollziehbar ist, sind Ausnahmebehandlungen meist nicht lokal und folgen auch nicht den gut verstandenen strukturierten Sprachkonzepten. Ein Programm, das viele Ausnahmebehandlungen enthält, ist daher oft nur schwer lesbar, und Programmänderungen bleiben selten lokal, da immer auch eine nicht direkt sichtbare `catch`-Klausel betroffen sein kann. Das sind gute Gründe, um die Verwendung von Ausnahmen zu vermeiden.

Faustregel: Man soll Ausnahmen nur einsetzen, wenn dadurch die Programmlogik vereinfacht wird.

Es gibt aber auch Fälle, in denen der Einsatz von Ausnahmen und deren Behandlungen die Programmlogik wesentlich vereinfachen kann, beispielsweise, weil viele bedingte Anweisungen durch eine einzige `catch`-Klausel ersetzbar sind. Wenn das Programm durch Verwendung von Ausnahmebehandlungen einfacher lesbar und verständlicher wird, ist der Einsatz durchaus sinnvoll. Das gilt vor allem dann, wenn die Ausnahmen lokal abgefangen werden. Oft sind aber gerade die nicht lokal abfangbaren Ausnahmen jene, die die Lesbarkeit am ehesten erhöhen können.

Wir wollen einige Beispiele betrachten, die Grenzfälle für den Einsatz von Ausnahmebehandlungen darstellen. Im ersten Beispiel geht es um eine einfache Iteration:

```
void f1 (vector const& v)
{
    int n=20;
    while (n!=0) cout << v[--n] << endl;
}
```

Die Bedingung in der `while`-Schleife kann man vermeiden, indem man die Ausnahme, dass `v[--n]` ausserhalb des Bereichs des Vektors zugreift, abfängt:

```
void f2 (vector const& v)
{
    int n=20;
    try {
        while (true) cout << v.at(--n) << endl;
    } catch (exception const& r) { }
}
```

Die zweite Variante ist dabei immer schlechter, sowohl von der Effizienz als auch von der Lesbarkeit. Ersteres ist deshalb so, weil statt dem ungeprüften Operator `operator[]` nun eine zusätzliche Abfrage, wegen der Verwendung von `at()`, benötigt wird und zusätzlich die (hier relativ teure) Ausnahmebehandlung durchgeführt wird. Das bedeutet aber nicht, dass grundsätzlich von der Verwendung von `at()` abgeraten wird. Es kann auch nicht darauf geschlossen werden, dass Ausnahmen immer weniger effizient sind.

Bei `f2` ist aber noch etwas ganz was anderes passiert. `cout` kann auch eine Ausnahme werfen, die auch von `exception` abgeleitet ist und somit gefangen werden würde. Bei `f1` hingegen würde diese Ausnahme zu den Aufrufer gelangen. Hier kann es auch sehr leicht korrigiert werden, indem `out_of_range` statt `exception` verwendet wird. Das Verhalten ist aber nicht immer so offensichtlich wie in diesem Beispiel. Ausnahmen können die Semantik des Programmes in vielfältiger Art und Weise beeinflussen. Erfahrene Programmierer können einschätzen, ob dieser Einfluss erwünscht ist oder nicht.

> **Faustregel:** Bei der Verwendung von Ausnahmen müssen nicht-lokale Effekte beachtet werden.

Das nächste Beispiel zeigt geschachtelte Typabfragen:

```
if (typeid(x) == typeid(T1)) { ... }
```

```
else if (typeid(x) == typeid(T2)) { ... }
...
else if (typeid(x) == typeid(Tn)) { ... }
else { ... }
```

Diese sind durch eine trickreiche, aber durchaus lesbare Verwendung von `catch`-Klauseln ersetzbar, die einer `switch`-Anweisung ähnelt:

```
try {throw x;}
catch (T1 const& x) { ... }
catch (T2 const& x) { ... }
...
catch (Tn const& x) { ... }
catch (...) { ... }
```

Da der `try`-Block nur eine `throw`-Klausel enthält, und spätestens in der letzten Zeile jede Ausnahme gefangen wird, kann es zu keinen nicht-lokalen Effekten kommen. Nach obigen Kriterien steht einer derartigen Verwendung von Ausnahmebehandlungen nichts im Wege. Allerdings entspringen beide Varianten einem schlechten Programmierstil: Typabfragen sollen, soweit es möglich ist, vermieden werden. Wenn, wie in diesem Beispiel, nach vielen Untertypen eines gemeinsamen Obertyps unterschieden wird, ist es sinnvoll, dynamisches Binden statt Typabfragen einzusetzen.

Das folgende Beispiel zeigt einen Fall, in dem die Verwendung von Ausnahmen sinnvoll ist. Angenommen, die Funktion `addA` addiert zwei beliebig große Zahlen, die durch Zeichenketten bestehend aus Ziffern dargestellt werden. Wenn eine Zeichenkette auch andere Zeichen enthält, gibt die Funktion die Zeichenkette `"Error"` zurück:

```
void string addA (string const& x, string const& y)
{
    if (onlyDigits(x) && onlyDigits(y))
    {
        ...
    } else {
        return "Error";
    }
}
```

Diese Art des Umgangs mit Fehlern ist problematisch, da das Ergebnis jedes Aufrufs der Methode mit `"Error"` verglichen werden muss, bevor es weiter verwendet werden kann. Wird ein Vergleich vergessen, pflanzt sich der Fehler in andere Programmzweige fort. Wird eine Ausnahme ausgelöst, gibt es dieses Problem nicht:

```
void string addA (string const& x, string const& y)
{
    if (onlyDigits(x) && onlyDigits(y))
    {
        ...
```

```
    } else {
        throw no_number_string();
    }
}
```

Bei dieser Art des Umgangs mit Fehlern kann sich der Fehler nicht leicht fortpflanzen. Immer dann, wenn ein bestimmter Ergebniswert fehlerhafte Programmzustände anzeigt, ist es ratsam, statt diesem Wert eine Ausnahme zu verwenden. Diese Verwendung von Ausnahmen ist zwar nicht lokal, aber die Verwendung der speziellen Ergebniswerte erzeugt ebenso nicht-lokale Abhängigkeiten im Programm.

3.6 Wiederholungsfragen

1. Was ist Generizität? Wozu verwendet man Generizität?
2. Welche Typen können als Typparameter eingesetzt werden? Können diese Forderungen explizit gemacht werden?
3. In welchen Fällen soll man Generizität einsetzen, in welchen nicht?
4. Was bedeutet statische Typsicherheit?
5. Welche Arten von Generizität kann man hinsichtlich ihrer Übersetzung und ihrem Umgang mit Schranken unterscheiden? Welche Art wird in C++ verwendet, und wie flexibel ist diese Lösung?
6. Was versteht man unter Instanzierung im Kontext der Templates? Kann eine Klasse instanziert werden, obwohl eine Methode nicht instanziert werden kann?
7. Was sind Typparameter in C++? Welche Parameter gibt es sonst noch in Templates? Wozu kann man sie verwenden?
8. Welche Templates können zum Umsetzen von RAII helfen? Welche Kopiersemantik hat ein `auto_ptr`?
9. Wie kann man Generizität simulieren? Worauf verzichtet man, wenn man Generizität nur simuliert?
10. Was wird bei der heterogenen bzw. homogenen Übersetzung von Generizität genau gemacht?

11. Welche Operatoren müssen Iteratoren in C++ unterstützen? Welche Arten von Iteratoren gibt es? Wie definiert man einen Iterator für einen Container?
12. Welche Möglichkeiten für dynamische Typabfragen gibt es in C++ , und wie funktionieren sie genau?
13. Was wird bei einer Typumwandlung in C++ umgewandelt – der deklarierte, dynamische oder statische Typ? Warum?
14. Welche Gefahren bestehen bei Typumwandlungen?
15. Wie kann man dynamische Typabfragen und Typumwandlungen vermeiden? In welchen Fällen kann das schwierig sein?
16. Welche Arten von Typumwandlungen sind sicher? Warum?
17. Was sind kovariante Probleme und binäre Methoden? Wie kann man mit ihnen umgehen oder sie vermeiden?
18. Wie unterscheidet sich Überschreiben von Überladen, und was sind Multimethoden?
19. Wie kann man Multimethoden simulieren? Welche Probleme können dabei auftreten?
20. Was ist das Visitor-Entwurfsmuster?
21. Wodurch ist Überladen problematisch, und in welchen Fällen ergeben sich kaum Probleme?
22. Wie werden Ausnahmebehandlungen in C++ unterstützt?
23. Wie sind Ausnahmen in Untertypbeziehungen zu berücksichtigen?
24. Wozu kann man Ausnahmen verwenden? Wozu soll man sie verwenden, wozu nicht?

Kapitel 4

Softwareentwurfsmuster

Nun beschäftigen wir uns mit dem bereits in Abschnitt 1.3 angeschnittenen Thema der Entwurfsmuster (design patterns), die der Wiederverwendung kollektiver Erfahrung dienen. Wir wollen exemplarisch einige häufig verwendete Entwurfsmuster betrachten. Da das Thema der Lehrveranstaltung die objektorientierte Programmierung ist, konzentrieren wir uns dabei auf Implementierungsaspekte und erwähnen andere in der Praxis wichtige Aspekte nur am Rande. Jedem, der sich für Entwurfsmuster in der Software interessiert, sei folgendes Buch empfohlen [8]:

> E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1994.

Es gibt eine Reihe neuerer Ausgaben, die ebenso empfehlenswert sind. Auch eine deutsche Übersetzung ist erschienen [9]:

> E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, Bonn, 1996.

Wir betrachten im Skriptum und in der Lehrveranstaltung nur einen kleinen Teil der im Buch beschriebenen und in der Praxis häufig eingesetzten Entwurfsmuster. Wie im Buch gliedern wir die beschriebenen Entwurfsmuster in drei Bereiche: Muster zur Erzeugung neuer Objekte (creational patterns) werden in Abschnitt 4.1 behandelt, jene, die die Struktur der Software beeinflussen (structural patterns) in Abschnitt 4.2, und schließlich jene, die mit dem Verhalten von Objekten zu tun haben (behavioral patterns), in Abschnitt 4.3.

4.1 Erzeugende Entwurfsmuster

Unter den erzeugenden Entwurfsmustern betrachten wir drei recht einfache Beispiele – Factory Method, Prototype und Singleton. Diese Entwurfsmuster wurden gewählt, da sie zeigen, dass man oft mit relativ einfachen Programmiertechniken die in Programmiersprachen vorgegebenen Möglichkeiten erweitern kann. Konkret wollen wir uns Möglichkeiten zur Erzeugung neuer Objekte vor Augen führen, die über die Verwendung des Operators `new` in C++ hinausgehen.

4.1.1 Factory Method

Der Zweck einer *Factory Method*, auch *Virtual Constructor* genannt, ist die Definition einer Schnittstelle für die Objekterzeugung, wobei Unterklassen entscheiden, von welcher Klasse die erzeugten Objekte sein sollen. Die tatsächliche Erzeugung der Objekte wird in Unterklassen verschoben.

Als Beispiel für eine Anwendung der Factory Method kann man sich ein System zur Verwaltung von Dokumenten unterschiedlicher Arten (Texte, Grafiken, Videos, etc.) vorstellen. Dabei gibt es eine (abstrakte) Klasse `DocCreator` mit der Aufgabe, neue Dokumente anzulegen. Nur in einer Unterklasse, der die Art des neuen Dokuments bekannt ist, kann die Erzeugung tatsächlich durchgeführt werden. Wie in `NewDocManager` ist der genaue Typ eines zu erzeugenden Objekts dem Compiler oft nicht bekannt:

Listing 4.1: docmanager.h

```
#include <vector>

class Document
{
public:
    virtual ~Document() = 0;
}; // Document::~Document() {} in .cpp

class Text : public Document { };
// + classes Picture, Video,...

class DocCreator
{
public:
    virtual Document* create() = 0;
    virtual ~DocCreator() = 0;
}; // DocCreator::~DocCreator() {} in .cpp

class TextCreator : public DocCreator
{
public:
    Document* create() {return new Text();}
```

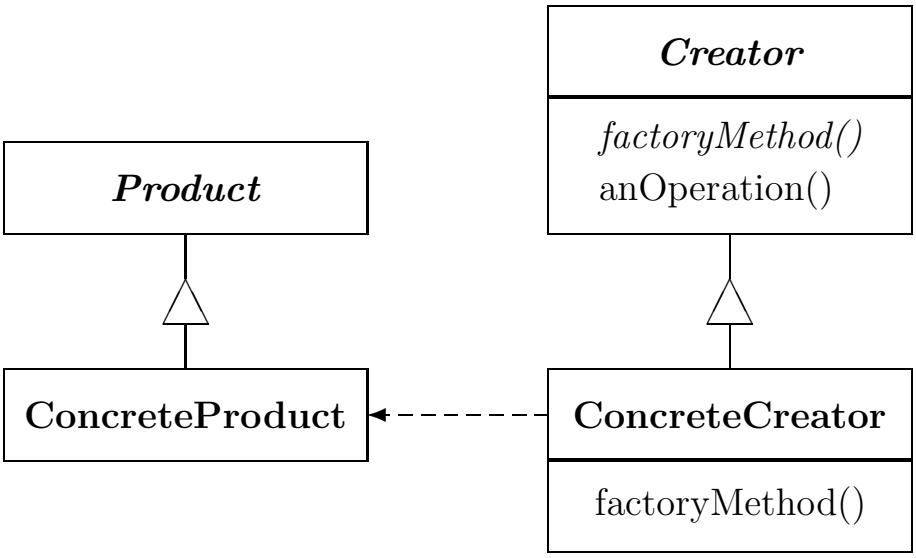
```
};
// + classes PictureCreator, VideoCreator,...

class NewDocManager
{
    DocCreator& m_c;
    std::vector<Document*> m_elems;
public:
    NewDocManager(DocCreator& c) : m_c (c) {}
    ~NewDocManager()
    {
        for (size_t i=0; i<m_elems.size(); i++)
        {
            delete m_elems[i];
        }
    }
    Document* newDoc()
    {
        Document* d = m_c.create();
        m_elems.push_back (d);
        return d;
    }
};
```

Generell ist das Entwurfsmuster anwendbar wenn

- eine Klasse neue Objekte erzeugen soll, deren Klasse aber nicht kennt;
- eine Klasse möchte, dass ihre Unterklassen die Objekte bestimmen, die die Klasse erzeugt;
- Klassen Verantwortlichkeiten an eine von mehreren Unterklassen delegieren, und man das Wissen, an welche Unterklasse delegiert wird, lokal halten möchte.
- die Allokation und Freigabe von Objekten zentral in einer Klasse verwaltet werden soll.

Die Struktur dieses Entwurfsmusters sieht wie in der folgenden Grafik aus. Wir werden Klassen als Kästchen darstellen, die die Namen der Klassen in Fettschrift enthalten. Durch einen waagrechten Strich getrennt können auch Namen von Methoden (mit einer Parameterliste) und Variablen (ohne Parameterliste) in den Klassen in nicht-fetter Schrift angegeben sein. Namen von abstrakten Klassen und Methoden sind kursiv dargestellt, konkrete Klassen und Methoden nicht kursiv. Unterklassen sind mit deren Oberklassen durch Striche und Dreiecke, deren Spitzen zu den Oberklassen zeigen, verbunden. Es wird implizit angenommen, dass jede solche Vererbungsbeziehung gleichzeitig auch eine Untertypbeziehung ist. Eine strichlierte Linie mit einem Pfeil zwischen Klassen bedeutet, dass

eine Klasse eine Instanz der Klasse, zu der der Pfeil zeigt, erzeugen kann. Namen im Programmcode, der ein Entwurfsmuster implementiert, können sich natürlich von den Namen in der Grafik unterscheiden. Die Namen in der Grafik helfen nur dem intuitiven Verständnis der Struktur und ermöglichen deren Erklärung. Sie haben keine inhaltliche Bedeutung.



Die (oft abstrakte) Klasse „Product“ ist (wie `Document` im Beispiel) ein gemeinsamer Obertyp aller Objekte, die von der Factory Method erzeugt werden können. Die Klasse „ConcreteProduct“ ist eine bestimmte Unterklasse davon, beispielsweise `Text`. Die abstrakte Klasse „Creator“ enthält neben anderen Operationen die Factory Method als (meist abstrakte) Methode. Diese Methode kann von außen, aber auch beispielsweise in „anOperation“ von der Klasse selbst verwendet werden. Eine Unterklasse „ConcreteCreator“ implementiert die Factory Method. Ausführungen dieser Methode erzeugen neue Instanzen von „ConcreteProduct“.

Factory Methods haben unter anderem folgende Eigenschaften:

- Sie bieten Anknüpfungspunkte (hooks) für Unterklassen. Die Erzeugung eines neuen Objekts mittels Factory Method ist fast immer flexibler als die direkte Objekterzeugung. Vor allem wird die Entwicklung von Unterklassen vereinfacht.

- Sie verknüpfen parallele Klassenhierarchien, die Creator-Hierarchie mit der Product-Hierarchie. Beispielsweise ist die Klassenstruktur bestehend aus `Document`, `Text`, etc. äquivalent zu der, die von den Klassen `DocCreator`, `TextCreator`, etc. gebildet wird. Dies kann unter anderem bei kovarianten Problemen hilfreich sein. Beispielsweise

erzeugt eine Methode `generiereFutter` in der Klasse `Tier` nicht direkt Futter einer bestimmten Art, sondern liefert in der Unterklasse `Rind` eine neue Instanz von `Gras` und in `Tiger` eine von `Fleisch` zurück. Meist sind parallele Klassenhierarchien (mit vielen Klassen) aber unerwünscht.

Zur Implementierung dieses Entwurfsmusters kann man die Factory Method in „Creator“ entweder als abstrakte Methode realisieren, oder eine Default-Implementierung dafür vorgeben. Im ersten Fall braucht „Creator“ keine Klasse kennen, die als „ConcreteProduct“ verwendbar ist, dafür sind alle konkreten Unterklassen gezwungen, die Factory Method zu implementieren. Im zweiten Fall kann man „Creator“ selbst zu einer konkreten Klasse machen, gibt aber Unterklassen von „Creator“ die Möglichkeit, die Factory Method zu überschreiben.

Es ist manchmal sinnvoll, der Factory Method Parameter mitzugeben, die bestimmen, welche Art von Produkt erzeugt werden soll. In diesem Fall bietet die Möglichkeit des Überschreibens noch mehr Flexibilität.

Folgendes Beispiel zeigt eine Anwendung von Factory Methods mit *lazy initialization*:

```
class Creator
{
    Product* m_product;
protected:
    virtual Product* createProduct() = 0;
public:
    Creator() : m_product(0) {}
    virtual Product* getProduct()
    {
        if (m_product == 0)
        {
            m_product = createProduct();
        }
        return m_product;
    }
    virtual ~Creator() {}
};
```

Eine neue Instanz des Produkts wird nur einmal erzeugt. Die Methode `getProduct` gibt bei jedem Aufruf dasselbe Objekt zurück.

Ein Nachteil des Entwurfsmusters besteht manchmal in der Notwendigkeit, viele Unterklassen von „Creator“ zu erzeugen, die nur `new` mit einem bestimmten „ConcreteProduct“ aufrufen. Generizität bietet hier wieder einen Ausweg:

```
template <typename Product>
class GenCreator : public Creator
```

```
{
public:
    Product* createProduct() { return new Product(); }
};
```

Bei einer homogenen Übersetzung wäre dieses Konstrukt nicht möglich, da dort neue Instanzen des Typparameters nicht erzeugt werden können.

4.1.2 Prototype

Das Entwurfsmuster *Prototype* dient dazu, die Art eines neu zu erzeugenden Objekts durch ein Prototyp-Objekt zu spezifizieren. Neue Objekte werden durch Kopieren dieses Prototyps erzeugt.

Zum Beispiel kann man in einem System, in dem verschiedene Arten von Polygonen wie Dreiecke und Rechtecke vorkommen, ein neues Polygon durch Kopieren eines bestehenden Polygons erzeugen. Das neue Polygon hat dieselbe Klasse wie das Polygon, von dem die Kopie erstellt wurde. An der Stelle im Programm, an der der Kopiervorgang aufgerufen wird (sagen wir in einem Zeichenprogramm), braucht diese Klasse nicht bekannt zu sein. Das neue Polygon kann etwa durch Ändern seiner Größe oder Position einen vom kopierten Polygon verschiedenen Zustand erhalten:

```
#include <memory>
std::auto_ptr<Polygon> duplicate (Polygon* orig)
{
    std::auto_ptr<Polygon> copy(orig->clone());
    copy->move();
    return copy;
}
```

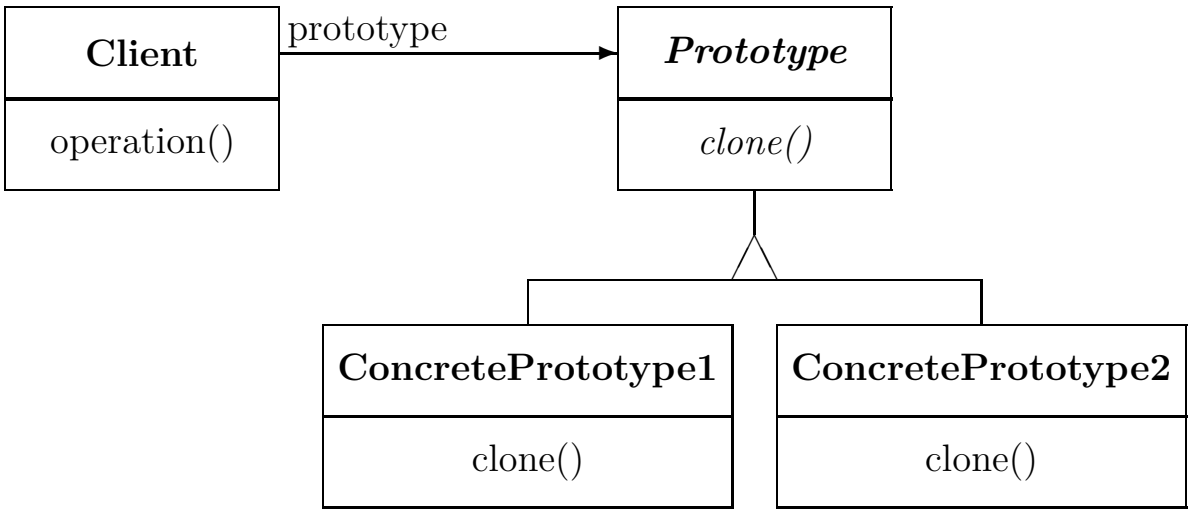
Da wir hier die Kopie eines neu allozierten Polygon zurückgeben, verwenden wir `auto_ptr` damit das Polygon automatisch freigegeben wird wenn es nicht mehr gebraucht wird oder eine Ausnahme ausgelöst wird.

Generell ist dieses Entwurfsmuster anwendbar, wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und dargestellt werden, und wenn

- die Klassen, von denen Instanzen erzeugt werden sollen, erst zur Laufzeit bekannt sind (beispielsweise wegen dynamischen Ladens), oder
- vermieden werden soll, eine Hierarchie von „Creator“-Klassen zu erzeugen, die einer parallelen Hierarchie von „Product“-Klassen entspricht (Factory Method), oder
- jede Instanz einer Klasse nur wenige unterschiedliche Zustände haben kann; es ist oft einfacher, für jeden möglichen Zustand einen Prototyp

zu erzeugen und diese Prototypen zu kopieren, als Instanzen durch `new` zu erzeugen und dabei passende Zustände anzugeben.

Das Entwurfsmuster hat folgende Struktur. Ein durchgezogener Pfeil bedeutet, dass jede Instanz der Klasse, von der der Pfeil ausgeht, auf eine Instanz der Klasse, auf die der Pfeil zeigt, verweist. Die entsprechende Variable hat den Namen, mit dem der Pfeil bezeichnet ist.



Die (möglicherweise abstrakte) Klasse „Prototype“ spezifiziert (wie „Polygon“ im Beispiel) eine (möglicherweise abstrakte) Methode „clone“ um sich selbst zu kopieren. Die konkreten Unterklassen (wie „Dreieck“ und „Rechteck“) überschreiben diese Methode. Die Klasse „Client“ entspricht im Beispiel dem Zeichenprogramm (mit der Methode `duplicate`). Zur Erzeugung eines neuen Objekts wird „clone“ in „Prototype“ oder durch dynamisches Binden in einem Untertyp von „Prototype“ aufgerufen.

Prototypes haben unter anderem folgende Eigenschaften:

- Sie verstecken die konkreten Produktklassen vor den Anwendern (clients) und reduzieren damit die Anzahl der Klassen, die Anwender kennen müssen. Die Anwender brauchen nicht geändert zu werden, wenn neue Produktklassen dazukommen oder geändert werden.

- Prototypen können auch zur Laufzeit jederzeit dazugegeben und weggenommen werden. Im Gegensatz dazu darf die Klassenstruktur zur Laufzeit in der Regel nicht verändert werden.

- Sie erlauben die Spezifikation neuer Objekte durch änderbare Werte. In hochdynamischen Systemen kann neues Verhalten durch Objektkomposition (das Zusammensetzen neuer Objekte aus mehreren bestehenden Objekten) statt durch die Definition neuer Klassen erzeugt

werden, beispielsweise durch die Spezifikation von Werten in Objektvariablen. Verweise auf andere Objekte in Variablen ersetzen dabei Vererbung. Die Erzeugung einer Kopie eines Objekts ähnelt der Erzeugung einer Klasseninstanz. Der Zustand eines Prototyps kann sich (wie der jedes beliebigen Objekts) jederzeit ändern, während Klassen zur Laufzeit unveränderlich sind.

- Sie vermeiden übertrieben große Anzahlen an Unterklassen. Im Gegensatz zu Factory Methods ist es nicht nötig, parallele Klassenhierarchien zu erzeugen.

- Sie erlauben die dynamische Konfiguration von Programmen. Prototypes sind eine Möglichkeit in C++ Objekte neuer Klassen dynamisch zu laden. Das wird beim dynamischen Laden von *Shared Libraries* oftmals eingesetzt. Die zu ladende Klasse muss dabei nicht im ursprünglichen Programm bekannt gewesen sein.

Für dieses Entwurfsmuster ist es notwendig, dass jede konkrete Unterklasse von „Prototype“ die Methode „clone“ implementiert. Gerade das ist aber oft schwierig, vor allem, wenn Klassen aus Klassenbibliotheken Verwendung finden, oder wenn es zyklische Referenzen gibt.

Bei der Implementierung von `clone` ist, ähnlich wie beim Kopierkonstruktor, darauf zu achten ob *flache* oder *tiefe* Kopien gemacht werden müssen. Wenn die Werte von Variablen nicht identisch sondern nur gleich sein sollen, müssen Variablen, die über Referenzen und Zeiger angesprochen werden, entsprechend neu allokiert und kopiert werden. Bei identischen Variablen muss eindeutig geklärt werden, welche Klasse oder Objekt für die Freigabe verantwortlich ist, ansonsten ist dieses Design zu meiden.

Eine Implementierung von `clone` zur Erzeugung tiefer Kopien kann sehr komplex sein. Das Hauptproblem stellen dabei zyklische Referenzen dar. Wenn `clone` einfach nur rekursiv auf zyklische Strukturen angewandt wird, erhält man eine Endlosschleife, die zum Programmabbruch aus Speichermangel führt. Wie solche zyklischen Referenzen aufgelöst werden sollen, hängt im Wesentlichen von der Anwendung ab. Ähnliche Probleme ergeben sich, wenn zusammenhängende Objekte ausgegeben (und wieder eingelesen) werden sollen. Das nennt man Serialisierung welches im einfachsten Fall durch die Operatoren `<<` und `>>` mit Streams erledigt werden kann. Oft ist aber eine Entkopplung der Klasse vom Archivformat erwünscht, auch dafür gibt es schon fertige Lösungen wie *boost::-Serialization*, welche gleich direkt portable Binär-, Text- und XML-Daten

generieren können. Zudem ist dann auch Versionierung vorhanden, welches ein wichtiges Problem bei Erweiterung von Klassen löst.

ProgrammiererInnen können kaum den Überblick über ein System behalten, das viele Prototypen enthält. Das gilt vor allem für Prototypen, die zur Laufzeit dazukommen. Zur Lösung dieses Problems haben sich *Prototyp-Manager* bewährt, das sind assoziative Datenstrukturen (wie `std::map`), in denen nach geeigneten Prototypen gesucht wird:

Listing 4.2: prototype.h

```
#include <memory>
#include <string>
#include <map>

class DoesNotExist : public std::exception {};

template <typename Prototype>
class PrototypeManager
{
    typedef std::map <std::string, Prototype*> map;
    map m_table;
public:
    void insert (std::string const& name, Prototype const& obj)
    {
        m_table[name] = obj.clone();
    }
    std::auto_ptr<Prototype> create (std::string const& name)
    {
        Prototype* proto = m_table[name];
        if (!proto) throw DoesNotExist();
        std::auto_ptr<Prototype> ret(proto->clone());
        return ret;
    }
    ~PrototypeManager()
    {
        for (typename map::const_iterator it = m_table.begin();
                it!=m_table.end(); ++it)
        {
            delete it->second;
        }
    }
};
```

Dabei erlaubt `insert` das Einfügen von neuen Prototypen. Mit der Methode `create` können dann beliebige Instanzen von Prototypen mit einem Identifikations-String erzeugt werden.

Oft ist es notwendig, nach Erzeugung einer Kopie den Objektzustand zu verändern. Im Gegensatz zu Konstruktoren kann „clone“ auf Grund des Ersetzbarkeitsprinzips meist nicht mit passenden Argumenten aufgerufen werden. In diesen Fällen ist es nötig, dass die Klassen Methoden zur Initialisierung beziehungsweise zum Ändern des Zustands bereitstellen.

Prototypen sind vor allem in statischen Sprachen wie C++ sinnvoll. In eher dynamischen Sprachen wie ECMAScript, Smalltalk und Objective C wird ähnliche Funktionalität bereits direkt von der Sprache unterstützt. Dieses Entwurfsmuster ist in die sehr dynamische objektorientierte Sprache *Self* [23] fest eingebaut und bildet dort die einzige Möglichkeit zur Erzeugung neuer Instanzen. Es gibt in Self keine Klassen, sondern nur Objekte, die als Prototypen verwendbar sind.

4.1.3 Singleton

Das Entwurfsmuster *Singleton* sichert zu, dass eine Klasse nur eine Instanz hat und erlaubt globalen Zugriff auf diese Instanz.

Es gibt zahlreiche Anwendungsmöglichkeiten für dieses Entwurfsmuster. Beispielsweise soll in einem System nur ein Drucker-Spooler existieren. Eine einfache Lösung besteht in der Verwendung einer globalen Variable. Aber globale Variablen verhindern nicht, dass mehrere Instanzen der Klasse erzeugt werden. Es ist besser, die Klasse selbst für die Verwaltung ihrer einzigen Instanz verantwortlich zu machen. Das ist die Aufgabe eines Singleton.

Dieses Entwurfsmuster ist anwendbar wenn

- es genau eine Instanz einer Klasse geben soll, und diese global zugreifbar sein soll;
- die Klasse durch Vererbung erweiterbar sein soll, und Anwender die erweiterte Klasse ohne Änderungen verwenden können sollen.

Auf Grund der Einfachheit dieses Entwurfsmusters verzichten wir auf eine grafische Darstellung. Ein Singleton besteht nur aus einer gleichnamigen Klasse mit einer statischen Methode „instance“, welche die einzige Instanz der Klasse zurückgibt. Obwohl die Erklärung so einfach ist, sind die Probleme bei der Implementation schwierig zu lösen, wie nachher kurz angeschnitten wird.

Singletons haben unter anderem folgende Eigenschaften:

- Sie erlauben den kontrollierten Zugriff auf die einzige Instanz.
- Sie vermeiden durch Verzicht auf globale Variablen unnötige Namen im System und alle weiteren unangenehmen Eigenschaften globaler Variablen.
- Sie unterstützen Vererbung.

- Sie ermöglichen, dass auch gar keine Instanz entsteht, wenn das Singleton nicht gebraucht wird.
- Sie verhindern, dass irgendwo Instanzen außerhalb der Kontrolle von der Klasse selber erzeugt werden.
- Sie erlauben auch mehrere Instanzen. Man kann die Entscheidung zugunsten nur einer Instanz im System jederzeit ändern und auch die Erzeugung mehrerer Instanzen ermöglichen. Die Klasse hat weiterhin vollständige Kontrolle darüber, wie viele Instanzen erzeugt werden.
- Sie sind flexibler als statische Methoden, da statische Methoden kaum Änderungen erlauben und dynamisches Binden nicht unterstützen.

Es gibt sehr einfache Implementierungen wie folgendes Beispiel zeigt:

Listing 4.3: singleton.h

```
class Singleton
{
    Singleton() { }
    Singleton(Singleton const&) {}
    Singleton& operator=(Singleton const&) { return *this; }
    ~Singleton() {}
public:
    static Singleton& instance()
    {
        static Singleton obj;
        return obj;
    }
};
```

Durch den privaten Konstruktor und Kopierkonstruktor kann keine Kopie außerhalb der Klasse selber angefertigt werden. Der private Zuweisungsoperator verhindert, dass Zuweisungen auf die Instanz von Singleton stattfinden. Die statische Methode `instance` gibt die einzige statische Instanz zurück. Wenn die Methode das erste Mal aufgerufen wird, wird automatisch der Konstruktor aufgerufen. Ganz am Ende des Programmes wenn alle globalen Variablen abgebaut werden, wird der Destruktor aufgerufen.

Klingt doch alles perfekt, ist diese Implementation die ideale Lösung? Leider nicht, wie hier kurz anhand eines Beispieles erklärt wird. In einer Applikation gibt es die Singletons Drucker und Fehlerausgabe. Der Drucker wird zuerst alloziert, später passiert irgendwann ein Fehler, z.b. der Drucker hat kein Papier. In diesem Moment wird die Fehlerausgabe instanziert und sendet den Fehler. Bei der Programmbeendigung werden die Singletons in umgekehrter Reihenfolge freigegeben, also zuerst die Fehlerausgabe. Wenn jetzt genau bei der Freigabe der Ressource Drucker ein

Fehler passiert, z.b. die Patrone meldet, sie ist leer, so will die Applikation diesen Fehler ausgeben. Die Fehlerausgabe ist aber schon freigegeben! Wir sehen, dass es durchaus realistisch zu einem undefinierten Verhalten kommen kann. Eine einfache Lösung wäre, dass man die Erzeugung einer Fehlerausgabe zu Beginn erzwingt, dies ist aber bei vielen untereinander abhängigen Singletons dann aber sehr fehleranfällig und erzeugt eventuell viele nicht benötigte Instanzen. Es gibt auch bessere Lösungen zu diesem Problem, die sind aber wesentlich komplexer [Ale02].

Man benötigt häufig Singletons, für die mehrere Implementierungen zur Verfügung stehen. Das heißt, die Klasse `Singleton` hat Unterklassen. Beispielsweise gibt es mehrere Implementierungen für Drucker-Spooler, im System sollte trotzdem immer nur ein Drucker-Spooler aktiv sein. Das soll von `Singleton` auch dann garantiert werden, wenn ProgrammiererInnen eine Auswahl zwischen den Alternativen treffen können.

Überraschenderweise ist die Implementierung eines solchen Singletons gar nicht einfach. Der einfachste Ansatz funktioniert ähnlich der obigen Lösung, nur dass mit einem `if` oder `switch` zwischen verschiedenen Instanzen gewählt wird. Hier eine einfache Implementation der statischen Methode der die Definition aller Klassen bekannt sein muss:

Listing 4.4: subsingleton.h

```
Singleton& Singleton::instance()
{
    switch (check)
    {
        case 1: {static SingletonA obj; return obj;}
            break;
        case 2: {static SingletonB obj; return obj;}
            break;
        default:{static Singleton obj; return obj;}
    }
}
```

Dabei muss allerdings `friend class Singleton;` in den abgeleiteten Klassen geschrieben werden, um Zugriff auf die privaten Konstruktoren zu geben. Die statische oder globale Variable `check` muss allerdings bereits zuvor initialisiert werden. Zudem muss sichergestellt sein, dass diese nicht verändert werden kann.

Flexibler ist ein Ansatz ähnlich `Prototype`, wo mit Hilfe einer `std::map` der richtige Instanzierer gesucht wird. Dieser Instanzierer ist eine Funktion die ein statisches Objekt zurückliefert und kann mittels Templates generisch geschrieben werden. Allerdings können dabei die die Probleme, dass die Steuerung der Instanzierung über die statische oder globale Va-

riable `check` erfolgt nicht befriedigend gelöst werden. Außerdem kann der Eintrag neuer Alternativen problematisch sein, da dies nicht in der Verantwortung von `Singleton` liegt. Es wird deshalb kein Code dafür angegeben.

Jeder Einsatz von `Singleton` hat aber auch inhärent schlechte Eigenschaften und ist deshalb möglichst zu vermeiden. Zu einem ist die Testbarkeit von `Singleton` sehr schlecht, da sich verschiedene Testmodule unbeabsichtigt beeinflussen können. Dies ist bedingt durch ein prinzipielles Problem: Durch `Singleton` werden globale Seiteneffekte ermöglicht, das sind welche die das ganze Programm betreffen. Deshalb ist `Singleton` nur als bessere Alternative zu globalen Variablen zu sehen.

4.2 Strukturelle Entwurfsmuster

Wir wollen zwei einfache Vertreter der strukturellen Entwurfsmuster betrachten, die man häufig braucht. Diese Muster haben eine ähnliche Struktur, aber unterschiedliche Verwendungen und Eigenschaften.

4.2.1 Decorator

Das Entwurfsmuster *Decorator*, auch *Wrapper* genannt, gibt Objekten dynamisch zusätzliche Verantwortlichkeiten. Decorators stellen eine flexible Alternative zur Vererbung bereit.

Manchmal möchte man einzelnen Objekten zusätzliche Verantwortlichkeiten (siehe Abschnitt 1.3.1) geben, nicht aber der ganzen Klasse. Zum Beispiel möchte man einem Fenster am Bildschirm Bestandteile wie einen scroll bar geben, anderen Fenstern aber nicht. Es ist sogar üblich, dass der scroll bar dynamisch während der Verwendung eines Fensters nach Bedarf dazukommt und auch wieder weggenommen wird:

Listing 4.5: decorator.cpp

```
#include <string>

class IWindow
{
public:
    virtual void show (std::string&) = 0;
    virtual ~IWindow() {}
};

class WindowImpl : public IWindow
{
public:
    void show (std::string &) { /* ... */ }
```

```
};

class WinDecorator : public IWindow
{
protected:
    IWindow& m_win;
public:
    WinDecorator(IWindow& win): m_win(win) {}
    void show (std::string &) = 0;
};

class ScrollBar : public WinDecorator
{
public:
    void scroll (int) { /* ... */ }
    IWindow& noScrollBar() { return m_win; }
    ScrollBar (IWindow& win) : WinDecorator(win) {}
    void show (std::string &) { /* ... */ }
};

//.cpp:
void WinDecorator::show(std::string& text) { m_win.show(text); }

int main ()
{
    WindowImpl wi;
    IWindow& w = wi;
    w = ScrollBar(w);
    w = static_cast<ScrollBar&>(w).noScrollBar();
}
```

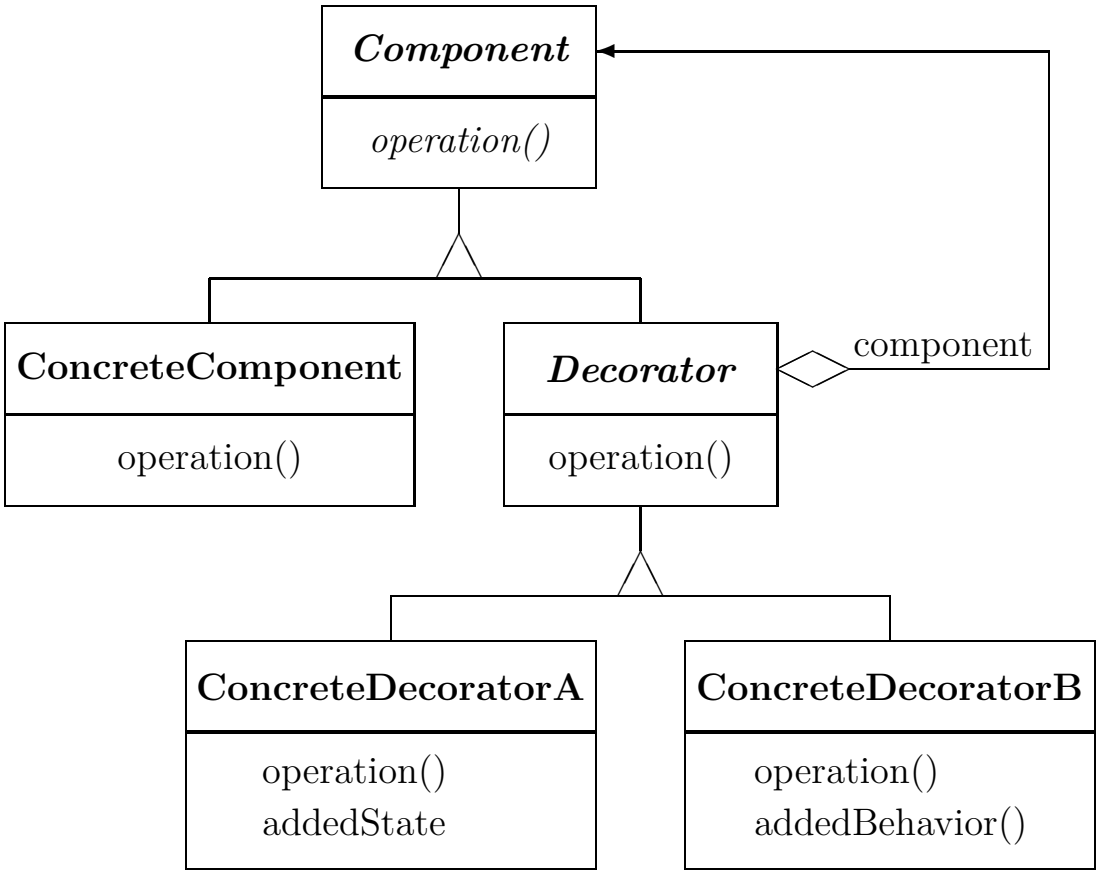
Im Allgemeinen ist dieses Entwurfsmuster anwendbar

- um dynamisch Verantwortlichkeiten zu einzelnen Objekten hinzuzufügen, ohne andere Objekte dadurch zu beeinflussen;

- für Verantwortlichkeiten, die wieder entzogen werden können;

- wenn Erweiterungen einer Klasse durch Vererbung unpraktisch sind, beispielsweise um eine sehr große Zahl an Unterklassen zu vermeiden, oder weil die Programmiersprache in einem speziellen Fall keine Vererbung unterstützt. In C++ kann das beispielsweise bei privaten oder nicht virtuellen Destruktoren auftreten.

Das Entwurfsmuster hat folgende Struktur, wobei der Pfeil mit einem Kästchen für Aggregation (also eine Referenz auf ein Objekt, dessen Bestandteil das die Referenz enthaltende Objekt ist) steht:



Die abstrakte Klasse beziehungsweise das Interface „Component“ (entspricht `IWindow`) definiert eine Schnittstelle für Objekte, an die Verantwortlichkeiten dynamisch hinzugefügt werden können. Die Klasse „ConcreteComponent“ ist, wie beispielsweise `WindowImpl`, eine konkrete Unterklasse davon. Die (abstrakte) Klasse „Decorator“ (`WinDecorator` im Beispiel) definiert eine Schnittstelle für Verantwortlichkeiten, die dynamisch zu Komponenten hinzugefügt werden können. Jede Instanz dieses Typs enthält eine Referenz namens „component“ (bzw. `m_win` im Beispiel) auf eine Instanz des Typs „Component“, das ist das Objekt, zu dem die Verantwortlichkeit hinzugefügt ist. Unterklassen von „Decorator“ sind konkrete Klassen, die bestimmte Funktionalität wie beispielsweise scroll bars bereitstellen. Sie definieren neben den Methoden, die bereits in „Component“ definiert sind, weitere Methoden und Variablen, welche die zusätzliche Funktionalität verfügbar machen. Wird eine Methode, die in „Component“ definiert ist, aufgerufen, so wird dieser Aufruf einfach an das Objekt, das über „component“ referenziert ist, weitergegeben.

Decorators haben einige positive und negative Eigenschaften:

- Sie bieten mehr Flexibilität als statische Vererbung. Wie bei stati-

scher Erweiterung einer Klasse durch Vererbung werden Verantwortlichkeiten hinzugefügt. Anders als bei Vererbung erfolgt das Hinzufügen der Verantwortlichkeiten zur Laufzeit und zu einzelnen Objekten, nicht ganzen Klassen. Die Verantwortlichkeiten können auch jederzeit wieder weggenommen werden.

- Sie vermeiden Klassen, die bereits weit oben in der Klassenhierarchie mit Eigenschaften (features) überladen sind. Es ist nicht notwendig, dass „ConcreteComponent“ die volle gewünschte Funktionalität enthält, da durch das Hinzufügen von Dekoratoren gezielt neue Funktionalität verfügbar gemacht werden kann.

- Instanzen von „Decorator“ und die dazugehörenden Instanzen von „ConcreteComponent“ sind nicht identisch. Beispielsweise hat ein Fenster-Objekt, auf das über einen Dekorator zugegriffen wird, eine andere Identität als das Fenster-Objekt selbst (ohne Dekorator) oder dasselbe Fenster-Objekt, auf das über einen anderen Dekorator zugegriffen wird. Bei Verwendung dieses Entwurfsmusters soll man sich nicht auf Objektidentität verlassen.

- Sie führen zu vielen kleinen Objekten. Ein Design, das Dekoratoren häufig verwendet, führt nicht selten zu einem System, in dem es viele kleine Objekte gibt, die einander ähneln. Solche Systeme sind zwar einfach konfigurierbar, aber schwer zu verstehen und zu warten.

Wenn es nur eine Dekorator-Klasse gibt, kann man die abstrakte Klasse „Decorator“ weglassen und statt dessen die konkrete Klasse verwenden. Bei mehreren Dekorator-Klassen zahlt sich die abstrakte Klasse aus: Alle Methoden, die bereits in „Component“ definiert sind, müssen in den Dekorator-Klassen auf gleiche Weise überschrieben werden. Sie rufen einfach dieselbe Methode in „component“ auf. Man braucht diese Methoden nur einmal in der abstrakten Klasse zu überschreiben. Von den konkreten Klassen werden sie geerbt.

Die Klasse oder das Interface „Component“ soll so klein wie möglich gehalten werden. Dies kann dadurch erreicht werden, dass „Component“ wirklich nur die notwendigen Operationen, aber keine Daten definiert. Daten und Implementierungsdetails sollen erst in „ConcreteComponent“ vorkommen. Andernfalls werden Dekoratoren umfangreich und ineffizient.

Dekoratoren eignen sich gut dazu, die Oberfläche beziehungsweise das Erscheinungsbild eines Objekts zu erweitern. Sie sind nicht gut für inhaltliche Erweiterungen geeignet. Auch für Objekte, die von Grund auf

umfangreich sind, eignen sich Dekoratoren kaum. Für solche Objekte sind andere Entwurfsmuster, beispielsweise *Strategy*, besser geeignet. Auf diese Entwurfsmuster wollen wir hier aber nicht eingehen.

4.2.2 Proxy

Ein *Proxy*, auch *Surrogate* genannt, stellt einen Platzhalter für ein anderes Objekt dar und kontrolliert Zugriffe darauf.

Es gibt zahlreiche, sehr unterschiedliche Anwendungsmöglichkeiten für Platzhalterobjekte. Ein Beispiel ist ein Objekt, dessen Erzeugung teuer ist, beispielsweise weil umfangreiche Daten aus dem Internet geladen werden müssen. Man erzeugt das eigentliche Objekt erst, wenn es wirklich gebraucht wird. Statt des eigentlichen Objekts verwendet man in der Zwischenzeit einen Platzhalter, der erst bei Bedarf durch das eigentliche Objekt ersetzt wird. Falls nie auf die Daten zugegriffen wird, erspart man sich den Aufwand der Objekterzeugung:

Listing 4.6: proxy.cpp

```
class Image
{
    int m_data [1280*1024][4];
public:
    Image() { /* Load from disc */ }
    void show() { /* ... */ }
};

class ImageProxy
{
    Image* m_image;
public:
    Image* operator->()
    {
        if (m_image == 0)
        {
            m_image = new Image();
        }
        return m_image;
    }
    Image& operator*() {return *operator->();}
    ImageProxy() : m_image(0) {}
    ~ImageProxy()
    {
        delete m_image;
    }
};

int main()
{
    ImageProxy i;
    i->show();
```

```
    (*i).show();
}
```

Es wird hier der Operator `->` und `*` überladen. Verwendet man diese, um auf den Proxy zuzugreifen, wird stattdessen das wirkliche Objekt geladen und verwendet. Es gibt aber syntaktisch keinen Unterschied zu einem Zeiger.

Jedes Platzhalterobjekt enthält im Wesentlichen einen Zeiger auf das eigentliche Objekt (sofern dieses existiert) und leitet in der Regel Nachrichten an das eigentliche Objekt weiter, möglicherweise nachdem weitere Aktionen gesetzt wurden. Einige Nachrichten werden manchmal auch direkt vom Proxy behandelt.

Das Entwurfsmuster ist anwendbar, wenn eine intelligentere Referenz auf ein Objekt als ein simpler Zeiger nötig ist. Hier sind einige übliche Situationen, in denen ein Proxy eingesetzt werden kann (keine vollständige Aufzählung):

Remote Proxies sind Platzhalter für Objekte, die nicht im selben Prozess existieren. Nachrichten an die Objekte werden von den Proxies über komplexere Kommunikationskanäle weitergeleitet.

Virtual Proxies erzeugen Objekte bei Bedarf. Da die Erzeugung eines Objekts aufwändig sein kann, wird sie so lange verzögert, bis es wirklich einen Bedarf dafür gibt.

Protection Proxies kontrollieren Zugriffe auf Objekte. Solche Proxies sind sinnvoll, wenn Objekte je nach Zugreifer oder Situation unterschiedliche Zugriffsrechte haben sollen.

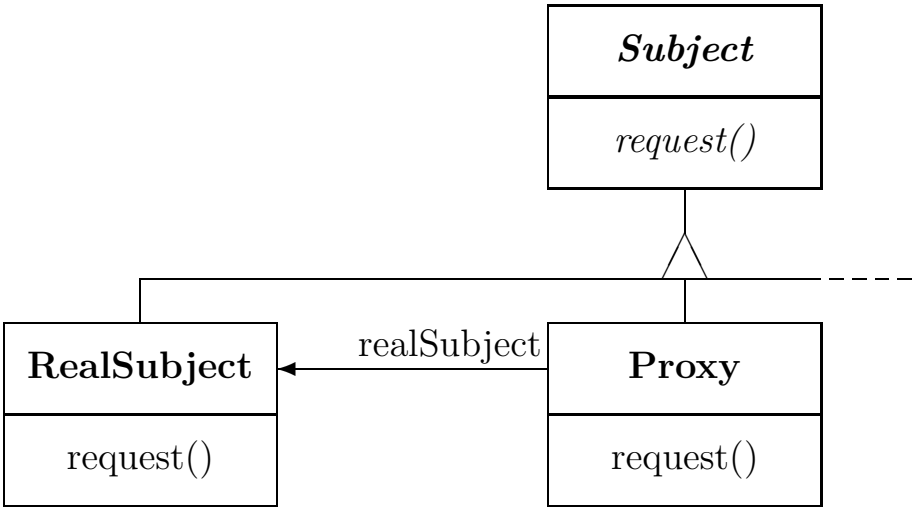
Smart References ersetzen einfache Zeiger. Sie können bei Zugriffen zusätzliche Aktionen ausführen. Typische Verwendungen sind

- das Mitzählen der Referenzen auf das eigentliche Objekt, damit das Objekt entfernt werden kann, wenn es keine Referenz mehr darauf gibt – das ist praktisch wenn ein Objekt automatisch freigegeben werden soll wenn es nicht mehr referenziert wird und kann auch problemlos in Container verwendet werden. Der intelligente Zeiger `std::tr1::shared_ptr` erfüllt genau diese Funktionalität (reference counting);
- das Laden von persistenten Objekten in den Speicher, wenn das erste Mal darauf zugegriffen wird (wobei die Unterscheidung zu Virtual Proxies manchmal unklar ist);

- das Zusichern, dass während des Zugriffs auf das Objekt kein gleichzeitiger Zugriff durch einen anderen Thread erfolgt (beispielsweise durch Setzen eines „locks“).

Es gibt zahlreiche weitere Einsatzmöglichkeiten. Der Phantasie sind hier kaum Grenzen gesetzt.

Die Struktur dieses Entwurfsmusters ist recht einfach:



Die abstrakte Klasse oder das Interface „Subject“ definiert die gemeinsame Schnittstelle für Instanzen von „RealSubject“ und „Proxy“. Instanzen von „RealSubject“ und „Proxy“ können gleichermaßen verwendet werden, wo eine Instanz von „Subject“ erwartet wird. Die Klasse „RealSubject“ definiert die eigentlichen Objekte, die durch die Proxies (Platzhalter) repräsentiert werden. Die Klasse „Proxy“ definiert schließlich die Proxies. Diese Klasse

- verwaltet eine Referenz „realSubject“, über die ein Proxy auf Instanzen von „RealSubject“ (oder auch andere Instanzen von „Subject“) zugreifen kann;
- stellt eine Schnittstelle bereit, die der von „Subject“ entspricht, damit ein Proxy als Ersatz des eigentlichen Objekts verwendet werden kann;
- kontrolliert Zugriffe auf das eigentliche Objekt und kann für dessen Erzeugung oder Entfernung verantwortlich sein;
- hat weitere Verantwortlichkeiten, die von der Art abhängen.

Es kann mehrere unterschiedliche Klassen für Proxies geben. Zugriffe auf Instanzen von „RealSubject“ können durch mehrere Proxies (möglicher-

weise unterschiedlicher Typen) kontrolliert werden, die in Form einer Kette miteinander verbunden sind.

In obiger Grafik zur Struktur des Entwurfsmusters zeigt ein Pfeil von „Proxy“ auf „RealSubject“. Das bedeutet, „Proxy“ muss „RealSubject“ kennen. Dies ist notwendig, wenn ein Proxy Instanzen von „RealSubject“ erzeugen soll. In anderen Fällen reicht es, wenn „Proxy“ nur „Subject“ kennt, der Pfeil also auf „Subject“ zeigt.

In der Implementierung muss man beachten, wie man auf ein Objekt zeigt, das in einem anderen Namensraum liegt oder noch gar nicht existiert. Für nicht existierende Objekte könnte man zum Beispiel `0` verwenden und für Objekte in einer Datei den Dateinamen.

Ein Proxy kann dieselbe Struktur wie ein Decorator haben. Aber Proxies dienen einem ganz anderen Zweck als Decorators: Ein Decorator erweitert ein Objekt um zusätzliche Verantwortlichkeiten, während ein Proxy den Zugriff auf das Objekt kontrolliert. Damit haben diese Entwurfsmuster auch gänzlich unterschiedliche Eigenschaften.

4.3 Entwurfsmuster für Verhalten

Zwei Beispiele zu Entwurfsmustern für Verhalten, nämlich Iterator und Visitor, haben wir bereits in Kapitel 3 beschrieben. Hier wollen wir nur einige ergänzende Bemerkungen zu Iteratoren machen. Ein weiteres Entwurfsmuster, nämlich Template Method soll dazu anregen, beim Entwerfen und Programmieren von Software der eigenen Fantasie freien Lauf zu lassen und auch dort Möglichkeiten für die Wiederverwendung von Programmcode zu finden, wo es keine spezielle Unterstützung durch eine Programmiersprache gibt.

4.3.1 Iterator

Ein *Iterator*, auch *Cursor* genannt, ermöglicht den Zugriff auf die Elemente eines Aggregats, ohne die innere Darstellung des Aggregats offen zu legen.

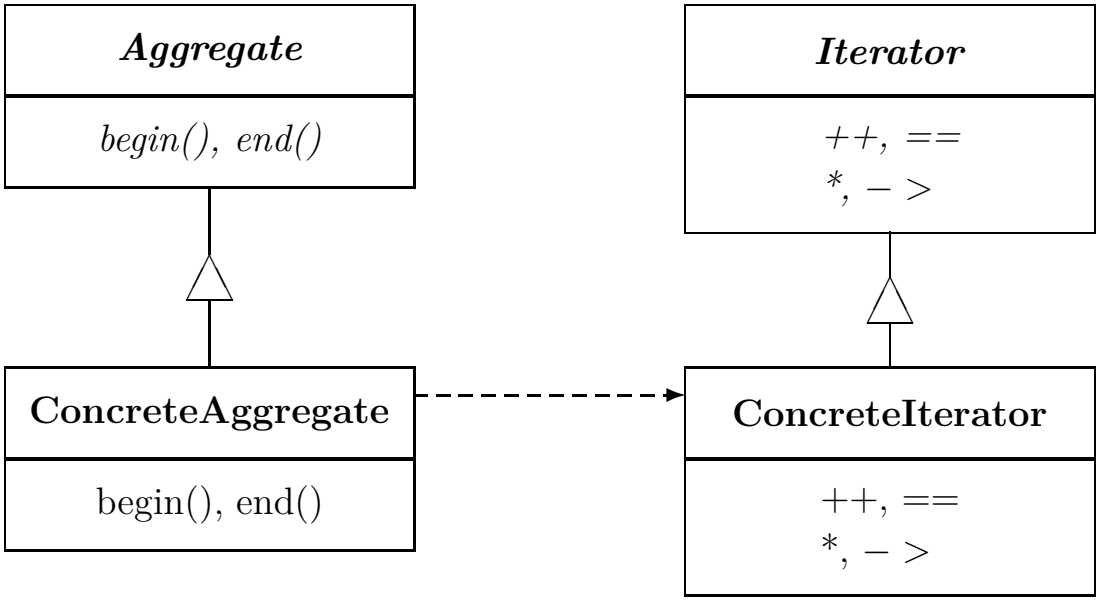
Hier wird das Entwurfsmuster Iterator behandelt. Das Konzept in C++, welches Iteratoren beschreibt, ist in Kapitel 3.1.4 erklärt.

Dieses Entwurfsmuster ist verwendbar, um

- auf den Inhalt eines Aggregats zugreifen zu können, ohne die innere Darstellung offen legen zu müssen;

- mehrere (gleichzeitige bzw. überlappende) Abarbeitungen der Elemente in einem Aggregat zu ermöglichen;
- eine einheitliche Schnittstelle für die Abarbeitung verschiedener Aggregatstrukturen zu haben, das heißt, um polymorphe Iterationen zu unterstützen.

Das Entwurfsmuster hat folgende Struktur:



„Iterator“ und „Aggregate“ sind dabei aber keine tatsächlich ausimplementierten Interfaces von denen abgeleitet werden kann, sondern Schnittstellen im abstrakteren Sinne die man zu implementieren hat. Die Klasse „ConcreteAggregate“ implementiert diese Schnittstelle. Ein Aufruf von `begin()` oder `end()` erzeugt üblicherweise eine neue Instanz von „ConcreteIterator“, was durch den strichlierten Pfeil angedeutet ist.

Iteratoren haben drei wichtige Eigenschaften:

- Sie unterstützen unterschiedliche Varianten in der Abarbeitung von Aggregaten. Für komplexe Aggregate wie beispielsweise Bäume gibt es zahlreiche Möglichkeiten, in welcher Reihenfolge die Elemente abgearbeitet werden. Es ist leicht, mehrere Iteratoren für unterschiedliche Abarbeitungsreihenfolgen zu implementieren.
- Iteratoren vereinfachen die Schnittstelle von „Aggregate“, da Zugriffsmöglichkeiten, die über Iteratoren bereitgestellt werden, durch die Schnittstelle von „Aggregate“ nicht unterstützt werden müssen. Es sind nur Methoden wie `begin()`, um einen Iterator auf den Beginn und `end()`, um einen Iterator *hinter* dem Ende zu bekommen,

in der Schnittstelle zu berücksichtigen. Es ist aber auch möglich, dass ein Iterator durch Verwendung von Funktionen erzeugt wird und in der Schnittstelle des Aggregats gar nicht vorgesehen ist. Das ist beispielsweise beim Outputstream der Fall. Ein Iterator wird hier durch das Funktion-Template `ostream_iterator` zurückgegeben.

- Auf ein und demselben Aggregat können gleichzeitig mehrere Abarbeitungen stattfinden, da jeder Iterator selbst den aktuellen Abarbeitungszustand verwaltet. Finden allerdings Änderungen im Aggregat statt, so sind alle erstellten Iteratoren unter Umständen ungültig.

Es gibt zahlreiche Möglichkeiten zur Implementierung von Iteratoren. Beispiele dafür haben wir bereits gesehen. Hier sind einige Anmerkungen zu Implementierungsvarianten:

- Man kann zwischen internen und externen Iteratoren unterscheiden. Interne Iteratoren kontrollieren selbst, wann die nächste Iteration erfolgt, bei externen Iteratoren bestimmen die Anwender, wann sie das nächste Element abarbeiten möchten. Alle Beispiele zu Iteratoren, die wir bis jetzt betrachtet haben, sind externe Iteratoren, bei denen Anwender in einer Schleife nach dem jeweils nächsten Element fragen. Ein interner Iterator enthält die Schleife selbst. Der Anwender übergibt dem Iterator eine Routine, die vom Iterator auf allen Elementen ausgeführt wird. Das kann in C++ mit `for_each` realisiert werden.

  Externe Iteratoren sind flexibler als interne Iteratoren. Zum Beispiel ist es mit externen Iteratoren leicht, zwei Aggregate miteinander zu vergleichen. Mit internen Iteratoren ist das schwierig. Andererseits sind interne Iteratoren oft einfacher zu verwenden, da eine Anwendung die Logik für die Iterationen (also die Schleife) nicht braucht. Interne Iterationen spielen vor allem in der funktionalen Programmierung eine große Rolle, da es dort gute Unterstützung für die dynamische Erzeugung und Übergabe von Routinen (in diesem Fall Funktionen) an Iteratoren gibt, andererseits aber externe Schleifen nur umständlich zu realisieren sind. In C++ werden beide Ansätze gleichermaßen unterstützt und es obliegt dem Programmierer je nach Situation oder Kenntnissen zu entscheiden.

- Oft ist es schwierig, externe Iteratoren auf Sammlungen von Elementen zu verwenden, wenn diese Elemente zueinander in komplexen Be-

ziehungen stehen. Durch die sequentielle Abarbeitung geht die Struktur dieser Beziehungen verloren. Beispielsweise erkennt man an einem vom Iterator zurückgegebenen Element nicht mehr, an welcher Stelle in einem Baum das Element steht. Wenn die Beziehungen zwischen den Elementen bei der Abarbeitung benötigt werden, ist es meist einfacher, interne statt externer Iteratoren zu verwenden.

- Löschen von Elementen, die das Aggregat verkleinern, ist mit Iteratoren gar nicht möglich. Deshalb bietet z.B. `std::list` die Methode `remove_if` an, welches als interner Iterator betrachtet werden kann. Algorithmen wie `unique` und `remove` liefern deshalb immer den Iterator auf das neue Ende zurück, damit das Aggregat dann mit einer Methode die tatsächliche Verkleinerung durchführen kann:

```
l.erase(unique(l.begin(), l.end()), l.end());
```

- In anderen Fällen benötigt ein Algorithmus mehr Funktionalität vom Iterator als ein Container anbieten kann. Beispielsweise benötigt `sort` einen Random Access Iterator, den Datenstrukturen mit verketteten Listen wie `std::list` nicht bieten können. Deshalb haben diese auch eine eigene Methode `sort` für die Sortierung, welche sogar garantiert dass keine Objekte kopiert werden müssen, da nur die Liste umgehängt wird.

- Der Algorithmus zum Durchwandern eines Aggregats muss nicht immer im Iterator selbst definiert sein. Auch das Aggregat kann den Algorithmus bereitstellen und den Iterator nur dazu benützen, eine Referenz auf das nächste Element zu speichern. Wenn der Iterator den Algorithmus definiert, ist es leichter, mehrere Iteratoren mit unterschiedlichen Algorithmen zu verwenden. In diesem Fall ist es auch leichter, Teile eines Algorithmus in einem anderen Algorithmus wiederzuverwenden. Andererseits müssen die Algorithmen oft private Implementierungsdetails des Aggregats verwenden. Das geht natürlich leichter, wenn die Algorithmen im Aggregat definiert sind. In C++ kann man Iteratoren durch geschachtelte Klassen in Aggregaten definieren. Es ist aber eher üblich das Aggregat und den Iterator nur in einem gemeinsamen Namensbereich zu halten und damit die Kopplung nicht zu stark erhöhen:

```
namespace container {
template <typename T>
class AggregatIterator
```

```
{
    // Iterator Implementierung
};

template <typename T>
class Aggregat: protected BaseAggregat
{
public:
    typedef AggregatIterator<T> iterator;
    // Aggregat Implementierung
};
} // end namespace
```

Benötigt der Iterator aber eine Referenz oder Kopie vom Aggregat, ist diese Vorgehensweise wegen der zyklischen Abhängigkeit nicht möglich und es müssen geschachtelte Klassen verwenden werden. Die Zugehörigkeit zu einem Iterator wird mit einem `typedef` festgelegt. Dadurch kann dann der genaue Typ des Iterators jederzeit durch `::iterator` im Aggregat erfragt werden.

- Es kann gefährlich sein, ein Aggregat zu verändern, während es von einem Iterator durchwandert wird. Wenn Elemente dazugefügt oder entfernt werden, passiert es leicht, dass Elemente nicht oder doppelt abgearbeitet werden. Eine einfache Lösung dieses Problems besteht darin, das Aggregat bei der Erzeugung eines Iterators zu kopieren. Meist ist diese Lösung aber zu aufwändig. Ein *robuster Iterator* erreicht dasselbe Ziel, ohne das ganze Aggregat zu kopieren. Es ist recht aufwändig, robuste Iteratoren zu schreiben. Die Detailprobleme hängen stark von der Art des Aggregats ab.

- Aus Gründen der Allgemeinheit ist es oft praktisch, Iteratoren auch auf leeren Aggregaten bereitzustellen. In einer Anwendung braucht man die Schleife nur so lange auszuführen, so lange es Elemente gibt – bei leeren Aggregaten daher nie – ohne eine eigene Behandlung für den Spezialfall zu brauchen. Das ist in C++ realisiert, indem `end` immer hinter das letzte Element zeigt. Ist nun der Iterator von `begin` und `end` gleich, so gibt es kein erstes Element, da dieses auch gleichzeitig hinter dem letzten liegt – es liegt somit ein leeres Aggregat vor. Durch die Semantik von `end` wird man gezwungen Schleifen so zu schreiben, dass dieser Fall berücksichtigt wird:

```
for(list<int>::iterator it = l.begin(); it!=l.end(); it++)
```

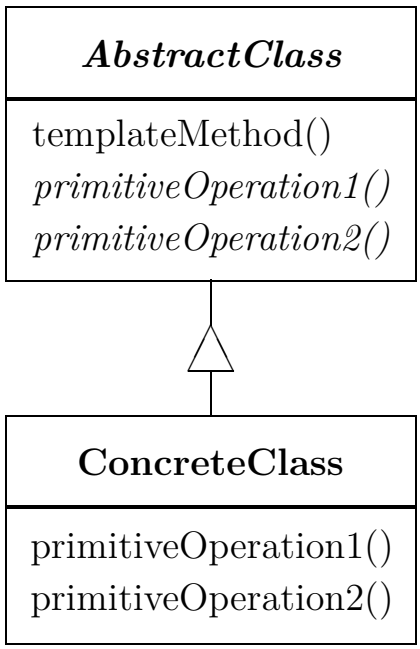
4.3.2 Template Method

Eine *Template Method* definiert das Grundgerüst eines Algorithmus in einer Operation, überlässt die Implementierung einiger Schritte aber einer Unterklasse. Template Methods erlauben einer Unterklasse, bestimmte Schritte zu überschreiben, ohne die Struktur des Algorithmus zu ändern.

Dieses Entwurfsmuster ist anwendbar

- um den unveränderlichen Teil eines Algorithmus einmal zu implementieren und es Unterklassen zu überlassen, den veränderbaren Teil des Verhaltens festzulegen;
- wenn gemeinsames Verhalten mehrerer Unterklassen (zum Beispiel im Zuge einer Refaktorisierung) in einer einzigen Klasse lokal zusammengefasst werden soll, um Duplikate im Code zu vermeiden;
- um mögliche Erweiterungen in Unterklassen zu kontrollieren, beispielsweise durch Template Methods, die *hooks* aufrufen und nur das Überschreiben dieser hooks in Unterklassen ermöglichen.

Die Struktur dieses Entwurfsmusters ist recht einfach:



Die (meist abstrakte) Klasse „AbstractClass“ definiert (abstrakte) primitive Operationen, welche konkrete Unterklassen als Schritte in einem Algorithmus implementieren, und implementiert das Grundgerüst des Algorithmus, das die primitiven Operationen aufruft. Die Klasse „ConcreteClass“ implementiert die primitiven Operationen.

Template Methods haben unter anderem folgende Eigenschaften:

- Sie stellen eine fundamentale Technik zur direkten Wiederverwendung von Programmcode dar (siehe Beispiele in Abschnitt 2.3.2).
- Sie führen zu einer umgekehrten Kontrollstruktur, die manchmal als *Hollywood-Prinzip* bezeichnet wird („Don't call us, we'll call you"). Die Oberklasse ruft die Methoden der Unterklasse auf – nicht umgekehrt.
- Sie rufen oft nur eine von mehreren Arten von Operationen auf:
  - konkrete Operationen (entweder in „ConcreteClass" oder in der Klasse, in der die Template Methods angewandt werden);
  - konkrete Operationen in „AbstractClass", also Operationen, die ganz allgemein auch für Unterklassen sinnvoll sind;
  - abstrakte primitive Operationen, die einzelne Schritte im Algorithmus ausführen;
  - Factory Methods;
  - hooks, das sind Operationen mit in „AbstractClass" definiertem Default-Verhalten, das bei Bedarf in Unterklassen überschrieben oder erweitert werden kann; oft besteht das Default-Verhalten darin, nichts zu tun.

  Es ist wichtig, dass genau spezifiziert ist, welche Operationen hooks (dürfen überschrieben werden) und welche abstrakt sind (müssen überschrieben werden). Es muss klar sein, welche Operationen dafür vorgesehen sind, überschrieben zu werden. Alle Operationen, bei denen es Sinn macht, dass sie in Unterklassen überschrieben werden, sollen hooks sein, da es beim Überschreiben anderer Operationen leicht zu Fehlern kommt.

Die primitiven Operationen, die von der Template Methode aufgerufen werden, sind in der Regel `protected` Methoden, damit sie nicht in unerwünschten Zusammenhängen aufrufbar sind. Die direkten Mittel um die Überschreibbarkeit in C++ auszudrücken sind *nicht-virtuelle* Methoden die nicht überschrieben werden dürfen und *rein-virtuelle* Methoden die überschrieben werden müssen:

```
class Base
{
public:
```

```
    /* virtual */ void nicht_ueberschreibbar() {}
    virtual void muss_ueberschrieben_werden() = 0;
    ...
};
```

Dadurch kontrolliert der Compiler ob die Unterklassen die Wiederverwendung richtig einsetzen.

Ein Ziel bei der Entwicklung einer Template Methode sollte sein, die Anzahl der primitiven Operationen möglichst klein zu halten. Je mehr Operationen überschrieben werden müssen, desto komplizierter wird die direkte Wiederverwendung von „AbstractClass“.

4.4 Wiederholungsfragen

1. Erklären Sie folgende Entwurfsmuster und beschreiben Sie jeweils das Anwendungsgebiet, die Struktur, die Eigenschaften und wichtige Details der Implementierung:
   - Factory Method
   - Prototype
   - Singleton
   - Decorator
   - Proxy
   - Iterator
   - Template Method
   - Visitor (siehe Abschnitt 3.4.2)
2. Wird die Anzahl der benötigten Klassen im System bei Verwendung von Factory Method, Prototype, Docorator und Proxy (genüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverändert?
3. Wird die Anzahl der benötigten Objekte im System bei Verwendung von Factory Method, Prototype, Docorator und Proxy (genüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverändert?
4. Vergleichen Sie Factory Method mit Prototype. Wann stellt welches Entwurfsmuster die bessere Lösung dar? Warum?

5. Welche Unterschiede gibt es zwischen Decorator und Proxy?
6. Welche Probleme kann es beim Erzeugen von Kopien im Prototype geben? Was unterscheidet flache Kopien von tiefen?
7. Für welche Arten von Problemen ist Decorator gut geeignet, für welche weniger? (Oberfläche versus Inhalt)
8. Kann man mehrere Decorators bzw. Proxies hintereinander verketten? Wozu kann so etwas gut sein?
9. Was unterscheidet hooks von abstrakten Methoden?
10. Welche Arten von Iteratoren gibt es, und wofür sind sie geeignet?
11. Inwiefern können geschachtelte Klassen bei der Implementierung von Iteratoren hilfreich sein?
12. Was ist ein robuster Iterator? Wozu braucht man Robustheit?
13. Wo liegen die Probleme in der Implementierung eines so einfachen Entwurfsmusters wie Singleton? Wann ist Singleton empfehlenswert?

Literaturverzeichnis

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects.* Springer, 1996.

[2] John Barnes. *Ada 95 Rationale.* Springer LNCS 1247, 1997.

[3] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Second edition, Benjamin-Cummings, Redwood City, California, 1994.

[4] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering,* **1**(2):199–207, June 1975.

[5] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded Polymorphism for Object-Oriented Programming. In *Proc. Conf. on Functional Programming Languages and Computer Architecture,* 273–280, 1989.

[6] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys,* **17**(4):471–522, 1985.

[7] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP'92),* Springer LNCS 615, Utrecht, The Netherlands, June 1992.

[8] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1994.

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, Bonn, 1996.

[10] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts.* Third edition, Wiley & Sons, New York, 1998.

[11] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[12] Atsushi Igarashi and Benjamin C. Pierce. Foundations for Virtual Types. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99),* Springer LNCS 1628, 161–185, Lisbon, Portugal, June 1999.

[13] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley, Reading, MA, 1989.

[14] B.B. Kristensen, O.L. Madsen, B. Moller-Pedersen, and K. Nygaard. The BETA Programming Language. In Bruce Shriver and Peter Wegner (Eds.): *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[15] Wilf LaLonde and John Pugh. Subclassing $\neq$ Subtyping $\neq$ Is-a. *Journal of Object-Oriented Programming,* **3**(5):57–62, 1991.

[16] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, ISBN 0-201-69581-2, 1996.

[17] Barbara Liskov and Jeannette M. Wing. Specifications and their Use in Defining Subtypes. *ACM SIGPLAN Notices,* **28**(10):16–28, October 1993, Proceedings OOPSLA'93.

[18] Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In *Research Directions in Concurrent Object-Oriented Programming,* MIT Press, 1993.

[19] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[20] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice Hall, 1997.

[21] Greg Nelson. *Systems Programming with Modula-3.* Prentice Hall Series in Innovative Technology, ISBN 0-13-590464-1, 1991.

[22] S.T. Taft and R.A. Duff. *Ada 95 Reference Manual.* Springer LNCS 1246, 1997.

[23] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings,* 227–241, Orlando, FL, October, 1987.

[24] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger,* **1**(1):7–87, August 1990.

[25] Peter Wegner and Stanley B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like is and isn't Like. In S. Gjessing and K. Nygaard (Eds.): *Proceedings ECOOP 1988,* Springer LNCS 322, 55–77, 1988.

C++ Literatur

[Ale02] A. Alexandrescu. *Modern C++ design*. Addison-Wesley Boston, 2002.

[GS05] D. Gregor and J. Siek. Implementing concepts. *N1848, August*, 2005.

[KM00] A. Koenig and B.E. Moo. *Accelerated C++: practical programming by example*. Addison-Wesley, 2000.

[KS05] S. Kuhlins and M. Schader. *Die C++-Standardbibliothek: Einführung und Nachschlagewerk*. Springer, 2005.

[LLM05] Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. *C++ Primer*. Addison-Wesley, 4 edition, December 2005.

[Mey95] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[Mey05] S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Professional, 2005.

[MS01] S.D. Meyers and O. Stafford. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, volume 7. 2001.

[Str87] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring 1987 European Unix Users Group Conference*, 1987.

[Str88] Bjarne Stroustrup. What is object-oriented programming? *IEEE Software*, 5(3):10–20, 1988.

[Str95] Bjarne Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 1995.

[Str00] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 4 edition, 2000.

[Wil06] A. Willms. *C++: Einstieg für Anspruchsvolle*. Addison-Wesley, January 2006.

Listings

1.1 stack1.h 16
1.2 stack1.cpp 19
1.3 stacktest1.cpp 20
1.4 Makefile 22
1.5 Makefile 22
1.6 counterstack1.h 30
1.7 counterstack1.cpp 31
1.8 interface.h 34
2.1 Kontravariante Parameter 61
2.2 Kovariante Parameter 62
2.3 kovariante Ergebnistypen 63
2.4 Dynamic Binding Test 68
2.5 hex.h 101
3.1 max.hpp 109
3.2 max.cpp 110
3.3 cmax.hpp 111
3.4 complex.hpp 111
3.5 complex.cpp 112
3.6 lexicalcast.hpp 112
3.7 lexicalcast.cpp 113
3.8 assert.cpp 114
3.9 stack2.hpp 115
3.10 stack2def.hpp 116
3.11 stack2.cpp 117
3.12 stacktest2.cpp 118
3.13 stack3.hpp 118
3.14 stack3def.hpp 119
3.15 stacktest3.cpp 119
3.16 counterstack3.hpp 120
3.17 counterstack3def.hpp 120
3.18 partial3.cpp 121
3.19 find.hpp 124
3.20 find.cpp 124
3.21 iterator.h 125

3.22 homogen.h 132
3.23 kovarianz2.h 139
3.24 kovarinaz3.h 141
3.25 multimeth.h 147
3.26 throw1.cpp 151
3.27 throw1.h 152
4.1 docmanager.h 162
4.2 prototype.h 169
4.3 singleton.h 171
4.4 subsingleton.h 172
4.5 decorator.cpp 173
4.6 proxy.cpp 177

Index

Überladen, **29**, 63, 95, 143
Überschreiben, **30**, 63

abstrakter Datentyp, **53**
Aggregat, 180
Analyse, **39**
Array, 17
assert, 115
Ausnahme, **150–159**
    ausnahmefester Code, 154
    bad_cast, 136
    Destruktor, 153
    Konstruktor, 153
    out_of_range, 157
    Stack Abwicklung, 154
auto_ptr, 122, 166

Basisklasse, *siehe* Oberklasse
Bereichsoperator, 32
Besitzsemantik, 94
Binden
    dynamisches, **29**, 32, 68, 143, 147
    statisches, **29**
black box, **15**
Brauchbarkeit, **36**

call back, 139
Cast
    const_cast, 135
    Downcast, 136
    dynamic_cast, 136
    reinterpret_cast, 135
    static_cast, 135
Client, **71**, 76
Container, 115
    at, 157
    heterogen, 130
    homogen, 130

data hiding, **15**
Datenabstraktion, **15**
Decorator, **173–177**, 180
Destruktor
    default, 98
Diamond Problem, 34
Dynamisches Laden, 26

Einfachvererbung, **34**
Entwurf, **39**
Entwurfsmuster, **47–49**, 161–187
Ersetzbarkeitsprinzip, **28**, 57–92
Erweiterung, **30**

Factory Method, 48, **162–166**
Faktorisierung, **37**
Funktor, 102

Generizität, **28**, 54, 107
    Concept, 133
    Fehlermeldung, 133
    heterogene, **132**
    homogene, **132**
    Homogene Übersetzung C++, 133

hook, 164, **186**

Identität, **13**, 15
implementieren, **14**
Implementierung, **14**, **39**
Instanz, **15**
    Template, 110
Instanzvariable, 16
Interface, 34
Invariante, **72**, 78

Invarianz, **61**
Iterator
    C++, **123–126**
    Entwurfsmuster, **180–184**
    Insert, 125
    Kategorie, 125
    Konstant, 125
    Reverse, 125
    Stream, 125

Kapselung, **13**, 38
Klasse, **15–26**, 32, 94
    abgeleitete, *siehe* Unterklasse
    abstrakte, **82**, 99
    geschachtelte, **99**
    konkrete, **83**
    spezifischste, **15**
Klassenvariable, **98**
Klassenzusammenhalt, **42**
Komponente, **53**
Konstante, 59, **98**
Konstruktor, **15**, 17, 95
    default, 97
Kontravarianz, **61**
Konvention, 95
Kopie, **13**, 166, 168
Kopierkonstruktor, 33, 95
    default, 97
Kovarianz, **60**

Lokalität, **37**, 66

Makro, 113
Makros, 114
Mangling, 26
Mehrfachvererbung, **34**
Methode, **14**
    abstrakte, **83**, 99
    binäre, **62**, 141
Modul, **53**
Multimethode, **143**, 145, 147

Nachbedingung, **72**, 78
Nachricht, 12, **14**
NDEBUG, 115

Oberklasse, **30**
Obertyp, **29**, 58
Objekt, **12–15**
    gleiches, **13**
    identisches, **13**
Objektkopplung, **43**
Operatoren
    überladen, 62

Paradigma, **49**
    deklaratives, **50**
    funktionales, **51**
    imperatives, **49**
    logikorientiertes, **51**
    objektorientiertes, **50**
    prozedurales, **50**
Pod
    Initialisieren, 97
Polymorphismus, **27–30**
    ad-hoc, **28**, 29
    enthaltender, **28**, 32, 57
    parametrischer, *siehe* Generizität
    universeller, **28**
Präprozessor, 114
private, 17, **102**
protected, **102**
Prototype, **166–170**
Proxy, **177–180**
public, 17, **102**

RAII, 17, **94**, 122, 154
Refaktorisierung, **44**

Schnittstelle, **14**, 28, 35, 58
    stabile, **65**, 66
schrittweise Verfeinerung, **40**
Serialisierung, 168
Server, **71**, 76
Shared Library, 168
shared_ptr, 178
Simulation, **38**, 44
Singleton, **170–173**
subtyping, *siehe* enthaltender Polymorphismus

Template Method, 90, **185–187**

this, **32**, 99
Typ, 27, 28, 32, 58, **75**
    Benutzerdefiniert, 100
    deklarierter, **27**, 67
    dynamischer, **27**, 67, 136
    stabiler, **65**, 66, 75, 84
    statischer, **27**, 67
typedef, 102
Typparameter, **28**, 107
Typumwandlung, **29**, 135–142

Untertyp, **29**, 58
Untertypbeziehung, **57–92**
    Ausgangsparameter in, **60**
    Durchgangsparameter in, **60**
    Eingangsparameter in, **59**
    Ergebnis in, **59**, 60
    Konstante in, **59**, 60
    Methode in, **59**
    Variable in, **59**, 61
    Zusicherung in, **77**
Untertyprelation, *siehe* Untertypbeziehung

Validierung, **39**
Verantwortlichkeit, **42**, 173
Vererbung, 30–35, 58, **85**, 174
    privat, 91
    private, 85, 88
    virtuell, 34
Verhalten, **14**, 71–82
Verifikation, **39**
Visitor, **149**
Vorbedingung, **72**, 77

Wartbarkeit, **37**, 38, 84, 89
Wartung, 15, 35, **37**
Wasserfallmodell, **39**
Wiederverwendung, **45**, 63, 65, 66
    direkte, **88**
    indirekte, **88**
Wrapper, *siehe* Decorator

Zugriffskontrolle, **102**
    friend, 99
    Vererbung, 103
Zusicherung, **73**
    Genauigkeit von, **76**
    Kommentar als, **74**, 79
Zustand, **13**, 16, 52
Zuverlässigkeit, **36**
Zuweisungsoperator
    default, 98
zyklischer Prozess, **40**