Objektorientierte Anwendungsentwicklung

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Davids

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2018

Inhalt

- *C/C++*
  - aktuelle Programmiersprache für Betriebssysteme, eingebettete Systeme, virtuelle Maschinen, Treiber und Signalprozessoren
  - gute Grundlage für C#, Java, PHP oder Perl
  - aktuell C++11: Unterstützung von Nebenläufigkeit (Threads), Erweiterung der Programmbibliothek um z.B. reguläre Ausdrücke, intelligente Zeiger (smart pointer), ungeordnete assoziative Container, eine Zufallszahlenbibliothek, numerische und mathematische Bibliotheken
- *UML* – Unified Modeling Language
  → graphische Darstellung der Systemkomponenten
- *Entwurfsmuster*
  → irgendwer hat Ihr (Entwurfs-)Problem schon gelöst
- *Refactoring*
  → Design bestehender Software verbessern

Inhalt

Auszug aus The C++ programming language von Bjarne Stroustrup:

- You don't have to know every detail of C++ to write good programs.
- Focus on programming techniques, not on language features.
- Don't reinvent the wheel, use libraries.
- Don't believe in magic: understand what your libraries do, how they do it, and at what cost they do it.

Literatur: C

- Brian W. Kernighan, Dennis M. Ritchie:
  Programmieren in C.
  Carl Hanser Verlag.
- Karlheinz Zeiner:
  Programmieren lernen mit C.
  Carl Hanser Verlag.
- Jürgen Wolf:
  C von A bis Z.
  Galileo Computing.

Literatur: C++

- Bjarne Stroustrup:
  The C++ Programming Language.
  Addison-Wesley.
- Martin Schader, Stefan Kuhlins:
  Programmieren in C++.
  Springer Verlag.
- Jürgen Wolf:
  C++ von A bis Z.
  Galileo Computing.
- Stefan Kuhlins, Martin Schader:
  Die C++ Standardbibliothek.
  Springer Verlag.

Literatur: OOA/OOD

- Bernd Oestereich:
  Objektorientierte Software-Entwicklung.
  Oldenbourg Verlag.
- Heide Balzert:
  Lehrbuch der Objektmodellierung.
  Spektrum Akademischer Verlag.
- Bernhard Rumpe:
  Modellierung mit UML.
  Springer Verlag.
- Scott W. Ambler:
  Process Patterns.
  Cambridge University Press.

Literatur: Software-Entwicklung

- Erich Gamma, Richard Helm, Ralph Johnson,
  John Vlissides:
  Entwurfsmuster.
  Addison-Wesley.
- Eric Freeman und Elisabeth Freeman
  mit Kathy Sierra und Bert Bates:
  Entwurfsmuster von Kopf bis Fuß.
  O'Reilly.
- Martin Fowler:
  Refactoring.
  Addison-Wesley.

Aktuelle Informationen, Sprechzeiten, Folien unter

`http://lionel.kr.hsnr.de/~rethmann/index.html`

Anmerkungen, Korrekturen oder Verbesserungsvorschläge sind immer willkommen! Sprechen Sie mich an oder schicken Sie mir eine E-Mail.

Büro: F 202
E-Mail: `jochen.rethmann@hs-niederrhein.de`

Büro: B 327
E-Mail: `peter.davids@hs-niederrhein.de`

Stellen Sie Fragen! Nur so kann ich beurteilen, ob Sie etwas verstanden haben oder noch im Trüben fischen.

Lernerfolg

Konfuzius:

Wer fragt, ist ein Narr für eine Minute. Wer nicht fragt, ist ein Narr sein Leben lang.

aus `www.lernen-als-weg.de`:

- Entspannen Sie sich. Richten Sie Ihre volle Aufmerksamkeit auf die Veranstaltung.
- Setzen Sie sich Ziele. Was wollen Sie in dieser Veranstaltung lernen?
- Hören Sie aktiv zu. Denken Sie mit und sorgen Sie dafür, dass alle Unklarheiten ausgeräumt werden.
- Notieren Sie Wichtiges. Machen Sie sich Notizen zur Veranstaltung und markieren Sie die wichtigsten Aspekte.
- Formulieren Sie Fragen. Notieren Sie Fragen und bringen Sie diese ein.

Lernerfolg

aus `www.lernen-als-weg.de`:

- Beteiligen Sie sich. Bringen Sie Ihre Anliegen und Ideen ein.
- Haben Sie Geduld. Lernen Sie, andere Ansichten zu akzeptieren. Helfen Sie, andere besser zu verstehen.
- Denken Sie positiv. Werden Sie sich darüber klar, wie die Veranstaltung zu Ihrem Lernerfolg beiträgt.
- Setzen Sie sich weitere Ziele. Entscheiden Sie, was Sie nach der Veranstaltung tun und wie Sie diese vertiefen.
- Handeln Sie schnell. Setzen Sie diese Ziele bald um. Verzögerung ist der erste Schritt zum Vergessen.

Lernerfolg

Der Lernerfolg wird am Ende durch eine Klausur geprüft:

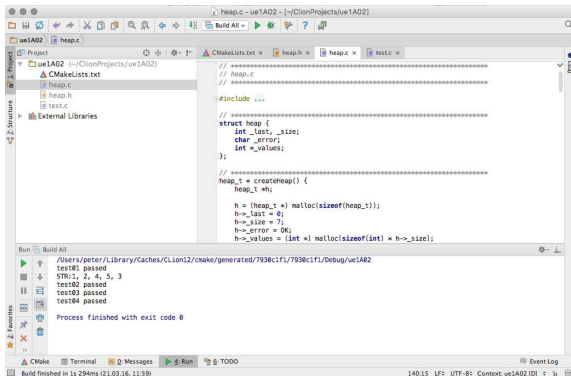
- In der Klausur steht Ihnen kein Computer, keine Online-Hilfe, kein Debugger und kein Compiler zur Verfügung.
- Die Klausursituation ist daher extrem anders als die Situation in der Übung oder dem Praktikum und muss geübt werden.

Bereiten Sie sich auf die Klausur vor, indem Sie Programme zunächst auf einem Blatt Papier entwickeln.

- Gehen Sie die einzelnen Programmschritte durch und vollziehen Sie dabei nach, ob das Programm korrekt ist.
- Implementieren Sie dann das Programm genau so, wie es auf dem Papier steht und kompilieren Sie es.
- Syntaxfehler beim nächsten Programm möglichst vermeiden!
- Nach dem Beseitigen der Syntaxfehler: Programm testen.
- Logische Fehler beim nächsten Programm vermeiden!

CLion

- Plattformunabhängige Programmierumgebung für C/C++
- Installiert unter Linux & Windows: B312, B315 und B322
- Kostenlose Studierenden-Lizenz verfügbar (..@stud.hn.de)
- URL: https://www.jetbrains.com/student



Übersicht

Evolution

- *Strukturierte Programmierung*
- Modulare Programmierung
- Objektorientierte Programmierung

Strukturierte Programmierung

Bei der strukturierten Programmierung werden

- Funktionen und Prozeduren dazu benutzt, Programme zu organisieren, z.B. `sqrt`, `sin`, `printf`, `toString`, ...
- Oft gebrauchte Funktionen, Prozeduren und Daten werden in Bibliotheken zur Verfügung gestellt: `stdio.h`, `stdlib.h`, `string.h`, `math.h`, `time.h`, ...

Funktionen reduzieren Copy-and-Paste von Programmteilen enorm. Anstelle von

```
if ((d1.jahr > d2.jahr)
        || (d1.jahr == d2.jahr
            && d1.monat > d2.monat)
        || (d1.jahr == d2.jahr
            && d1.monat == d2.monat
            && d1.tag > d2.tag)) {
    ...
}
```

würden wir die Logik eines Datumvergleichs in einer Funktion bereitstellen

```
bool isGreater(date_t a, date_t b) {
    return (a.jahr > b.jahr)
           || (a.jahr == b.jahr
               && a.monat > b.monat)
           || (a.jahr == b.jahr
               && a.monat == b.monat
               && a.tag > b.tag);
}
```

und an den jeweiligen Programmstellen die Funktion aufrufen:

```
if (isGreater(d1, d2)) {
    ...
}
```

→ das Programm wird lesbar: literarisches Programmieren

Literarisches Programmieren bezeichnet das Schreiben von Computerprogrammen in einer Form, sodass sie vor allem für Menschen lesbar sind.

Dies steht im Gegensatz zur konventionellen Ansicht, dass Programme hauptsächlich effizient sein sollen und dann oft nur noch für den Computer lesbar sind.

Jon Bentley fragte in *Communications of the ACM*: „When was the last time you spent a pleasant evening in a comfortable chair, reading a good program?“

aus: `http://de.wikipedia.org/wiki/Literate_programming`

Strukturierte Programmierung

Stellen wir die Funktion dann noch in einer Bibliothek bereit, kann die Funktion sogar projektübergreifend verwendet werden.

In C++ können wir das Ganze durch geeignete Operatorüberladung noch lesbarer schreiben: `if (d1 > d2) ...`

Ziele der strukturierten Programmierung:

- Verständlicher und übersichtlicher Code.
- Wartbarer und erweiterbarer Code.
- Wiederverwendung von Algorithmen.
- Wiederverwenden von Code durch allgemeingültige Funktionen oder Makros anstelle von Copy-and-Paste.

Problem: Typisierung

```
int ival[30];
int icmp(const void *, const void *);
...
void qsort(ival, 30, sizeof(int), icmp);
...
int icmp(const void *a, const void *b) {
    int x = *(int *) a;
    int y = *(int *) b;
    return x - y;
}
```

Lösung in C: Zeiger auf `void` bzw. Makros
besser in C++: Templates, Vererbung, Polymorphismus

Strukturierte Programmierung

Problem: globale Variablen oder lange Parameterlisten

- Programme sind einfacher zu verstehen, wenn sie aus kleinen, in sich geschlossenen, unabhängigen Teilen bestehen.
  - Globale Variablen führen zu voneinander abhängigen Funktionen. Das Ändern einer Funktion kann dazu führen, dass andere Funktionen nicht mehr korrekt funktionieren. Nach jeder Änderung muss man erneut das ganze Programm testen.
  - Übergeben wir alle benötigten Variablen als Parameter an die Funktionen, ergeben sich lange, unklare Parameterlisten.
- Keine Zugriffskontrolle: Bei den heutigen nebenläufigen Programmen ist es wichtig, den gleichzeitigen Zugriff mehrerer Threads auf gemeinsame Variablen zu synchronisieren.
- Namenskonflikte: In umfangreichen Programmen wird oft derselbe Variablenname zweimal verwendet.

Lösung in C: Module, incomplete data type
besser in C++: Klassen, private/protected, Namensräume

Strukturierte Programmierung

Problem: Lesbarkeit und Wartbarkeit

- Vergleichsoperatoren bei allgemeinen Datentypen
  Lösung in C: Funktionen
  besser in C++: Operatorüberladung
- Fehlerbehandlung
  Lösung in C: Fehlerflags als Rückgabewert einer Funktion,
  globale Fehlervariable `errno`, Signal-Handler
  besser in C++: Exceptions

Gehen wir die Probleme an! Lernen wir mit C++ eine tolle Programmiersprache kennen.

Liste

Oft müssen wir eine Liste von Elementen verwalten:

- Bücherliste in der Bibliothek
- Studentenliste im Prüfungsbüro
- Mitarbeiterliste in der Verwaltung
- KFZ-Liste im Straßenverkehrsamt
- ...

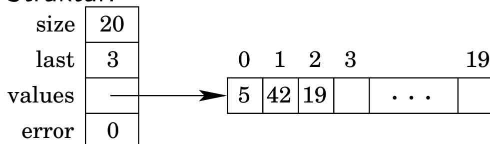
Die Anforderungen an solche Listen sind immer gleich:

- hinzufügen von Werten
- löschen von Werten
- suchen (z.B. Halter des Fahrzeugs KR-AB 123)
- ausdrucken oder anzeigen der Liste
- ...

Liste: erster Versuch

Einige Details zu unserer Implementierung:

- Die Liste beruht auf einem Array.
  - Das Array wird bei Bedarf automatisch vergrößert und
  - wird automatisch verkleinert, wenn so viele Elemente aus der Liste entfernt wurden, dass das Array nur noch zu einem Viertel gefüllt ist.
- Damit der Code wiederverwendet werden kann,
  - wurden alle wichtigen Variablen in einer Struktur zusammengefasst
  - und alle Operationen sind als Funktionen ausgeführt.
- Struktur:



Wiederholen wir zunächst kurz, wie ein dynamisch angelegtes Array vergrößert werden kann:

```
oldPtr ——→ | 7 | 19 | 32 | 17 |

newPtr = (typecast) realloc(oldPtr, newSize);

newPtr ——→ | 7 | 19 | 32 | 17 |   |   |   |   |
```

Konnte der alte Speicherbereich nicht vergrößert werden, dann wird neuer Speicherbereich allokiert, die alten Werte in den neuen Speicher kopiert und der alte Speicherbereich frei gegeben. In diesem Fall ist `oldPtr` nicht mehr gültig.

Oft soll unter dem gleichen Namen wie zuvor das Array weiterhin benutzt werden, dann ist `newPtr = oldPtr`:

```c
int *dArr = (int *) calloc(sizeof(int), 4);
    ...
dArr = (int *) realloc(dArr, sizeof(int) * 8);
```

Wichtig: Unterscheide Variablen und Strukturattribute!

```
typedef struct {
  int wert;     // Strukturattribut
  char *name;   // Strukturattribut
} foo_t;        // Name des Typs

int main(void) {
  char *h = "Hallo, Welt!";
  foo_t a;      // Variable vom Typ foo_t

  a.wert = 15; // Strukturattribut der Variablen
  a.name = (char *) malloc(strlen(h) + 1);
  strcpy(a.name, h);
  ...
}
```

Über eine Struktur kann ein Array gebildet werden!

```
typedef struct {
  int wert;     // Strukturattribut
  char *name;   // Strukturattribut
} foo_t;        // Name des Typs

int main(void) {
  char *h = "Hallo, Welt!";
  foo_t a[5];   // Array mit 5x Typ foo_t

  a[0].wert = 15;
  a[0].name = (char *) malloc(strlen(h) + 1);
  strcpy(a[0].name, h);
  ...
}
```

Liste: erster Versuch (Wiederholung: C)

Ein Array kann Zeiger auf Strukturen enthalten!

```
typedef struct {
  int wert;        // Strukturattribut
  char *name;      // Strukturattribut
} foo_t;           // Name des Typs

int main(void) {
  char *h = "Hallo, Welt!";
  foo_t *a[5];   // Array: 5x Zeiger auf foo_t

  a[0] = (foo_t *) malloc(sizeof(foo_t));
  a[0]->wert = 15;
  a[0]->name = (char *) malloc(strlen(h) + 1);
  strcpy(a[0]->name, h);
  ...
}
```

Arrays von Strukturen können auch dynamisch angelegt werden!

```
typedef struct {
  int wert;         // Strukturattribut
  char *name;       // Strukturattribut
} foo_t;            // Name des Typs

int main(void) {
  char *h = "Hallo, Welt!";
  foo_t *a;         // dynamisches Array

  a = (foo_t *) calloc(sizeof(foo_t), 5);

  a[0].wert = 15;
  a[0].name = (char *) malloc(strlen(h) + 1);
  strcpy(a[0].name, h);
  ...
}
```

Dynamische Arrays können Zeiger auf Strukturen enthalten!

```
typedef struct {
  int wert;     // Strukturattribut
  char *name; // Strukturattribut
} foo_t;        // Name des Typs

int main(void) {
  char *h = "Hallo, Welt!";
  foo_t **a;  // dyn. Array mit Zeiger auf foo_t

  a = (foo_t **) calloc(sizeof(foo_t *), 5);

  a[0] = (foo_t *) malloc(sizeof(foo_t));
  a[0]->wert = 15;
  a[0]->name = (char *) malloc(strlen(h) + 1);
  strcpy(a[0]->name, h);
  ...
}
```

Liste: erster Versuch

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

typedef struct {
    int size, last, *values;
    char error;
} list_t;

list_t *create() {
    list_t *l;

    l = (list_t *) calloc(1, sizeof(list_t));
    l->size = 8;
    l->last = 0;
    l->values = (int *) calloc(8, sizeof(int));
    l->error = 0;
    return l;
}
```

Liste: erster Versuch

```
char isFull(list_t *l) {
    return l->size == l->last;
}

void increase(list_t *l) {
    l->size *= 2;
    l->values = (int *) realloc(l->values,
            l->size * sizeof(int));
}

void append(list_t *l, int val) {
    if (isFull(l))
        increase(l);

    l->values[l->last] = val;
    l->last += 1;
}
```

Liste: erster Versuch

```
int find(list_t *l, int val) {
    int pos;

    for (pos = 0; pos < l->last; pos++)
        if (l->values[pos] == val)
            return pos;
    return -1;
}

void decrease(list_t *l) {
    l->size /= 2;
    l->values = (int *) realloc(l->values,
            l->size * sizeof(int));
}
```

Liste: erster Versuch

```
void erase(list_t *l, int val) {
    int pos = find(l, val);

    if (pos == -1)
        return;

    for (; pos < l->last -1; pos++)
        l->values[pos] = l->values[pos + 1];
    l->last -= 1;

    if (l->last < l->size / 4)
        decrease(l);
}
```

```
int getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->last) {
        l->error = 1;
        return -1;
    }
    return l->values[pos];
}

void destroy(list_t *l) {
    free(l->values);
    free(l);
}
```

Liste: erster Versuch

```c
void toScreen(list_t *l) {
    int i;

    for (i = 0; i < l->last; i++)
        printf("%d\n", l->values[i]);
}

int main(void) {
   int i;
   list_t *l;

   l = create();
   for (i = 1; i < 30; i++)
       append(l, i);
   toScreen(l);
```

Liste: erster Versuch

```
    for (i = 1; i < 30; i += 2)
        erase(l, i);
    toScreen(l);

    i = getValueAt(l, 20);
    if (l->error == 0)
        printf("value[%2d] = %2d\n", 20, i);
    else printf("20 out of range\n");

    destroy(l);
    return 0;
}
```

Frage: Was halten Sie von der Implementierung?

Liste: zweiter Versuch

liste.h

```
typedef struct {
    int size, last, *values;
    char error;
} list_t;

list_t *create();
char isFull(list_t *l);
void increase(list_t *l);
void decrease(list_t *l);
void append(list_t *l, int val);
int find(list_t *l, int val);
int getValueAt(list_t *l, int pos);
void erase(list_t *l, int val);
void toScreen(list_t *l);
void destroy(list_t *l);
```

Liste: zweiter Versuch

liste.c

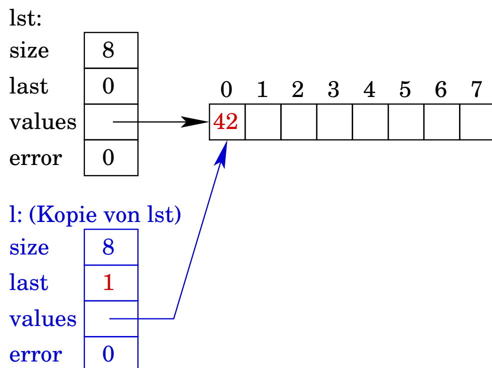
```
#include <stdio.h>
#include <stdlib.h>
#include "liste.h"

list_t *create() {
    list_t *l;

    l = (list_t *) calloc(1, sizeof(list_t));
    l->size = 8;
    l->last = 0;
    l->values = (int *) calloc(8, sizeof(int));
    l->error = 0;

    return l;
}

.....
```

Liste: zweiter Versuch

main.c

```c
#include <stdio.h>
#include "liste.h"

void main(void) {
    int i;
    list_t *l = create();

    for (i = 1; i < 30; i++)
        append(l, i);
    toScreen(l);

    i = getValueAt(l, 30);
    if (l->error != 0)
        printf("value[%2d] = %2d\n", 30, i);
    destroy(l);
}
```

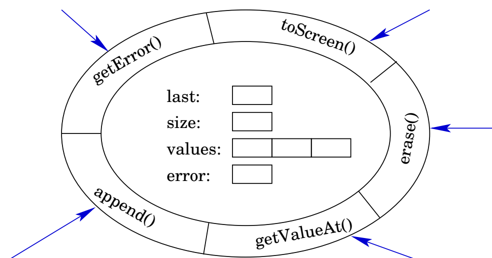
Warum wird die Liste als Zeiger übergeben? Weil sonst eine Kopie erzeugt würde, mit der die Funktionen wie `append(lst, 42)` dann arbeiten, und die Liste beim Aufrufer nicht geändert würde:



Liste: Verbesserungsmöglichkeiten

- Die Funktionen `increase()` und `decrease()` sind öffentlich bekannt, obwohl sie nur innerhalb der Liste verwendet werden und nicht von außen aufrufbar sein sollten.
- Der innere Aufbau der Liste ist durch das `typedef` in der Header-Datei nach außen hin bekannt.

⇒ Keine Datenkapselung! Im Hauptprogramm kann die Funktionalität der Liste durch einen schreibenden Zugriff wie `l->size = 0;` zerstört werden.



Übersicht

Evolution

- Strukturierte Programmierung
- *Modulare Programmierung*
- Objektorientierte Programmierung

Modulare Programmierung

Modulare Programmierung versucht der wachsenden Größe von Softwareprojekten Herr zu werden. Module können einzeln geplant, programmiert und getestet werden.

Universelle Module müssen nur einmal programmiert und können wiederverwendet werden. Je öfter ein Modul wiederverwendet wurde, desto sicherer kann man sein, dass es fehlerfrei ist.

Wenn alle Module erfolgreich getestet sind, können diese Einzelteile logisch miteinander verknüpft und zu einer größeren Anwendung zusammengesetzt werden.

Die modulare Programmierung erweitert den prozeduralen Ansatz, indem *Prozeduren zusammen mit Daten* in logischen Einheiten zusammengefasst werden.

aus: `http://de.wikipedia.org/wiki/Modulare_Programmierung`

Modulare Programmierung

In unserem Beispiel haben wir folgendes zu tun, um eine Liste als wiederverwendbares Modul zur Verfügung stellen zu können:

- In `liste.h` die Strukturvereinbarung entfernen und durch einen unvollständigen Typen (Vorwärtsdeklaration) ersetzen.
- In `liste.c` die aus `liste.h` entfernte Strukturvereinbarung aufnehmen.
- Wir müssen die Schnittstelle erweitern: Methoden zum Zugriff auf interne Variablen definieren, die im Hauptprogramm benötigt werden: `getError()`
- Datenkapselung sicherstellen, indem wir mit `static` die Funktionen „verstecken", die nach außen nicht sichtbar sein sollen: `isFull()`, `increase()`, usw.

Liste: dritter Versuch

liste.h

```
// ==================================
// incomplete data type
// (forward declaration)
// ==================================
typedef struct list list_t;

// ==================================
// interface
// ==================================
list_t *create();
void append(list_t *l, int val);
int getValueAt(list_t *l, int pos);
void erase(list_t *l, int val);
void toScreen(list_t *l);
char getError(list_t *l);     // neu !!!
void destroy(list_t *l);
```

Liste: dritter Versuch

liste.c

```
#include ....

struct list {
    int size, last, *values;
    char error;
};

list_t *create() {
    list_t *l;

    l = (list_t *) calloc(1, sizeof(list_t));
    l->size = 8;
    l->last = 0;
    l->values = (int *) calloc(8, sizeof(int));
    l->error = 0;

    return l;
}
```

Liste: dritter Versuch

```
// private !
static void increase(list_t *l) {
    l->size *= 2;
    l->values = (int *) realloc(l->values,
            l->size * sizeof(int));
}

// private !
static void decrease(list_t *l) {
    l->size /= 2;
    l->values = (int *) realloc(l->values,
            l->size * sizeof(int));
}

// private !
static char isFull(list_t *l) {
    return l->size == l->last;
}
```

Liste: dritter Versuch

```
void append(list_t *l, int val) {
    if (isFull(l))
        increase(l);

    l->values[l->last] = val;
    l->last += 1;
}

// private !
static int find(list_t *l, int val) {
    int pos;

    for (pos = 0; pos < l->last; pos++)
        if (l->values[pos] == val)
            return pos;
    return -1;
}
```

Liste: dritter Versuch

```
int getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->last) {
        l->error = 2;
        return -1;
    }
    return l->values[pos];
}

.....

char getError(list_t *l) {    // neu !
    return l->error;
}

void destroy(list_t *l) {
    free(l->values);
    free(l);
}
```

main.c

```c
#include <stdio.h>
#include "liste.h"

int main(void) {
    int i;
    list_t *l = create();

    for (i = 1; i < 30; i++)
        append(l, i);
    toScreen(l);

    i = getValueAt(l, 30);
    if (getError(l) == 0)
        printf("value[%2d] = %2d\n", 30, i);

    destroy(l);
    return 0;
}
```

Modulare Programmierung in C

Kommt Ihnen diese Art der Programmierung fremd vor? Finden Sie diese Art der Programmierung seltsam und zu kompliziert?

Wir kennen diese Art der Programmierung bereits aus C von den Dateioperationen. In der Header-Datei `stdio.h` ist definiert:

```
typedef struct _IO_FILE FILE;
```

In unseren Programmen konnten wir Funktionen darauf nutzen:

```
FILE *f;
f = fopen("dat.txt", "rw"); // vgl. create()
fprintf(f, ...);            // vgl. append()
fscanf(f, ...);             // vgl. getValueAt()
fgets(..., f);
fclose(f);                  // vgl. destroy()
```

Liste: dritter Versuch

Unsere Liste sieht doch schon ganz gut aus!

In C++ werden Module als Klassen realisiert. Klassen sind Grundelemente in der objektorientierten Programmierung.

Schauen wir es uns an!

Übersicht

Evolution

- Strukturierte Programmierung
- Modulare Programmierung
- *Objektorientierte Programmierung*

liste.h

```
class Liste {
private:     // nicht sichtbar
    int _size, _last, *_values;
    char _error;
    bool isFull();   // neuer Datentyp
    int find(int value);
    void increase();
    void decrease();

public:      // sichtbar
    Liste(int size); // Konstruktor statt create
    ~Liste();        // Destruktor statt destroy
    void append(int val);
    int getValueAt(int pos);
    void erase(int val);
    void toScreen();
    char getError();
};
```

Liste in C++

liste.cpp

```
#include <iostream>
#include "liste.h"
using namespace std;

Liste::Liste(int size) {
    _size = size;
    _last = 0;
    _error = 0;
    _values = new int[size];    // statt malloc
}

Liste::~Liste() {
    delete[] _values;           // statt free
}
```

Liste in C++

```
void Liste::increase() {
    int *tmp = new int[_size * 2];

    for (int i = 0; i < _size; i++)
        tmp[i] = _values[i];

    delete[] _values;
    _values = tmp;
    _size *= 2;
}

void Liste::append(int val) {
    if (isFull())
        increase();

    _values[_last] = val;
    _last += 1;
}
```

Liste in C++

```
int Liste::getValueAt(int pos) {
    if (pos < 0 || pos >= _last) {
        _error = 1;
        return -1;
    }
    return _values[pos];
}

int Liste::find(int val) {
    int pos;

    for (pos = 0; pos < _last; pos++)
        if (_values[pos] == val)
            return pos;
    return -1;
}
```

Liste in C++

```
bool Liste::isFull() {
    return _last == _size;
}

void Liste::decrease() {
    _size /= 2;
    int *tmp = new int[_size];

    for (int i = 0; i < _last; i++)
        tmp[i] = _values[i];

    delete[] _values;
    _values = tmp;
}
```

Liste in C++

```
void Liste::erase(int val) {
    int pos = find(val);

    if (pos == -1)
        return;

    for (; pos < _last -1; pos++)
        _values[pos] = _values[pos + 1];
    _last -= 1;

    if (_last < _size / 4)
        decrease();
}
```

Liste in C++

```
void Liste::toScreen() {
    for (int i = 0; i < _last; i++)
        cout << i << ": " << _values[i] << endl;
}

char Liste::getError() {
    return _error;
}
```

Anmerkung zu der Notation:

- Die Variablen, die mit einem Unterstrich beginnen, sind Attribute der Klasse, also Klassenvariablen.
- Im Unterschied dazu beginnen die lokalen Variablen und Parameter von Methoden immer mit einem Buchstaben.
- Dies ist eine eigene Notation und weder normiert noch in irgendwelchen Richtlinien empfohlen.

main.cpp

```
#include <iostream>
#include "liste.h"
using namespace std;

int main(void) {
    Liste l(10);

    for (int i = 1; i < 60; i++)
        l.append(i);
    l.toScreen();

    cout << endl;
    for (int i = 10; i < 60; i++)
        l.erase(i);
    l.toScreen();

    return 0;
}
```

Übersicht

Grundlagen C++

- *Klassen und Objekte*
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

Klassenkonzept in C++

- Klassen definieren einen neuen Datentyp.
- Klassen enthalten:
  - Datenelemente (auch Variablen oder Attribute genannt)
  - Elementfunktionen (auch Methoden genannt)
  - eingebettete Typen (z.B. andere Klassen)
  - Elementkonstanten (`enum`)
- Eine Klasse ist ein Bauplan für gleichartige Objekte.
- Im Allgemeinen beginnen Klassennamen mit einem Großbuchstaben und sind Substantive.

  Klassennamen bei Microsoft Windows beginnen meist mit einem C für Class. Die Attribute beginnen mit `m_` für Member und unterscheiden sich damit von lokalen Variablen und Parametern von Methoden.

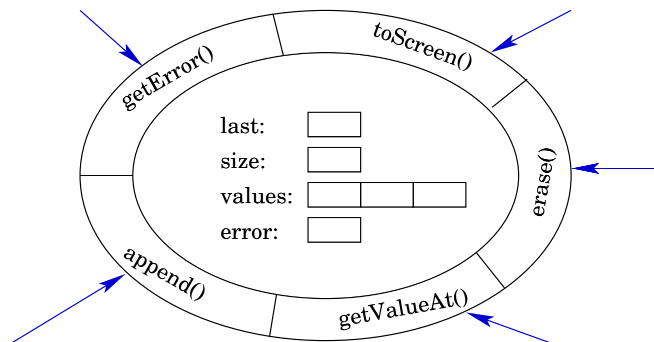
Klassenkonzept in C++

- Der Klassenname kann wie jeder andere vordefinierte Datentyp benutzt werden.
- Die Klassendeklaration alleine belegt keinen Speicherplatz. Sie legt nur die Struktur fest, nach der der Compiler ein Objekt der Klasse erzeugt.
- Jedes Objekt (auch Exemplar, fälschlicherweise oft auch Instanz genannt) hat seine eigenen Variablen.

  Ausnahme: `static` deklarierte Attribute
- Initialisierung von Variablen ist innerhalb einer Klassendeklaration nicht möglich, dies erfolgt im Konstruktor.

Klassenkonzept in C++

Wichtiges Prinzip bei der Programmierung allgemein und bei objektorientierter Programmierung im Speziellen: Datenkapselung!



Die Attribute sind vor direkten Zugriffen von außen geschützt, ein Zugriff kann nur über die öffentlichen Methoden erfolgen.

Klassenkonzept in C++

Angabe von Zugriffsrechten: `public` und `private`

- Auf `public`-Membervariablen und -funktionen darf von jeder Stelle eines Programms zugegriffen werden, also insbesondere von anderen Klassen.
- Auf `private`-Elemente darf nur von den Methoden einer Klasse selber zugegriffen werden. Nach außen hin sind diese Variablen und Funktionen unsichtbar.
- Das Sichtbarkeitsattribut `protected` wird im Abschnitt *Vererbung* behandelt.

Klassenkonzept in C++

Geltungsbereichsoperator (scope resolution operator):

- Trennung von Deklaration und Implementation mittels `::`.
- Zugriff auf Membervariable, die den gleichen Namen wie lokale Variable oder Parameter besitzt, oder mittels `this`.
- Innnerhalb einer Methode statt auf ein Member auf eine globale Variable/Funktion gleichen Namens zugreifen.

```
void klassenname::methode(int x, int y) {
    var1 = 10;     // Zugriff auf Membervariable
    x = 20;        // Zugriff auf Parameter x
    ::var1 = 30;   // Zugriff auf globale Variable
    ::x = 40;      // Zugriff auf globale Variable
    func();        // Zugriff auf Memberfunktion
    ::func();      // Zugriff auf globale Funktion
}
```

Unified Modeling Language

- Heutiger Standard der Darstellung der objektorientierten Sichtweise eines realen Problems.
- Basis sind die Klassen (bzw. Objekte) mit Attributen und Methoden eines Systems.
- Das statische Modell beschreibt die Zusammenhänge zwischen den Klassen.
- Das dynamische Modell beschreibt die Zustände und Abläufe innerhalb des Systems.
- Grafische Darstellung eines komplexen Systems zur Reduzierung der Komplexität.

  „The art of programming is the art of organizing complexity.“ (Edsgar W. Dijkstra)

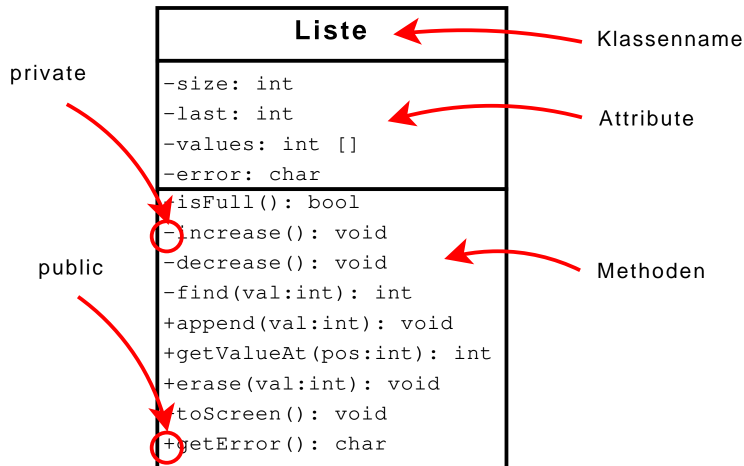
Modellierung mit UML

Eine Klasse wird in UML als Rechteck dargestellt. Das Rechteck enthält jeweils einen Bereich für

- den Namen der Klasse,
- die Attribute der Klasse und
- die Methoden der Klasse.

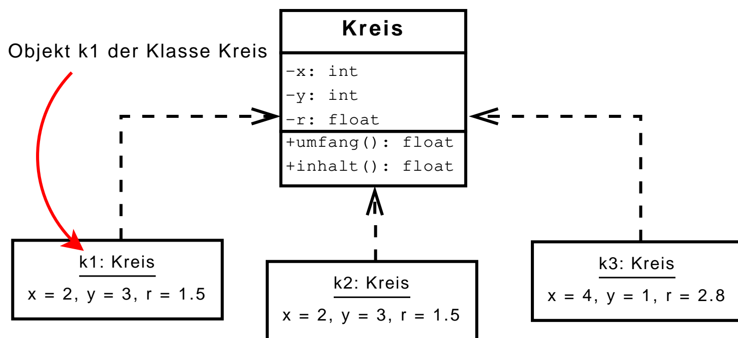
Die Sichtbarkeit der Attribute wird gekennzeichnet mit

- `-` für `private`
- `+` für `public`
- `#` für `protected` (später)
- `~` für `package` (nicht in C++)


Liste
Klassenname
private
-size: int
-last: int
-values: int []
-error: char
Attribute
+isFull(): bool
-increase(): void
-decrease(): void
-find(val:int): int
+append(val:int): void
+getValueAt(pos:int): int
+erase(val:int): void
+toScreen(): void
+getError(): char
Methoden
public

Variablen eines Objekts

Eine Klasse ist ein Bauplan für gleichartige Objekte. Jedes anhand eines solchen Bauplans erstelltes Objekt hat seinen eigenen Satz von Variablen.



Die Objekte k1 und k2 sind gleich, aber nicht identisch! Oder anders gesagt: k1 und k2 sind die gleichen Objekte, aber nicht dieselben.

Variablen eines Objekts

Wann zwei Objekte als gleich angesehen werden, bestimmt der Programmierer:

- `Konto:` gleiche Kontonummer und Bankleitzahl
- `Student:` gleiche Hochschule und Matrikelnummer

In Java wird dazu die Methode `equals` überschrieben, in C++ kann der Operator `==` überladen werden.

Variablen eines Objekts

Der C++-Compiler generiert bei jedem Methodenaufruf die Übergabe eines Zeigers auf das Objekt und setzt in der Methode vor jede Zustandsvariable diesen Zeiger:

```
Liste l(10);           Liste l(10);
l.append(i);     ⟶     Liste::append(&l, i);
```

Unsere Methode append entspricht also:

```
void Liste::append(Liste *l, int val) {
    if (isFull(l))
        increase(l);

    l->_values[l->_last] = val;
    l->_last += 1;
}
```

Beachte: So setzt es der Compiler um, programmiert wird anders!

Variablen eines Objekts

Der Zeiger ist eine implizite Objektreferenz. Explizit ist diese Referenz durch das Schlüsselwort `this` ansprechbar.

Unsere Methode `append` können wir auch so schreiben:

```
void Liste::append(int val) {
    if (this->isFull())
        this->increase();

    this->_values[this->_last] = val;
    this->_last += 1;
}
```

Das Schlüsselwort `this` wird überall dort automatisch vom Compiler eingefügt, wo es eindeutig ist. Im obigen Beispiel kann `this` entfallen.

Variablen eines Objekts

Im folgenden Beispiel ist es nicht eindeutig und muss angegeben werden:

```
class C {
private:
    int var;

public:
    C(int var) {
        this->var = var;
    }

    void fkt(int var) {
        cout << "Attribut: " << this->var;
        cout << ", Parameter: " << var << endl;
    }
};
```

Übersicht

Grundlagen C++

- Klassen und Objekte
- *Konstruktor und Destruktor*
- Strukturen
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

Konstruktor

- Der Konstruktor hat den gleichen Namen wie die Klasse.
- Der Konstruktor ist eine typlose – also auch nicht void! – Methode.
- *default-Konstruktor:* Konstruktor ohne Argumente

  Wurde kein Konstruktor programmiert, wird ein default-Konstruktor vom Compiler hinzugefügt.
- Wenn eine Klasse einen Konstruktor besitzt, wird jedes Klassenobjekt vor seiner ersten Verwendung durch den Konstruktoraufruf initialisiert.
- Innerhalb eines Konstruktors können auch Methoden der Klasse aufgerufen werden.
- Der Vollständigkeit halber: Ein Konstruktor darf nicht als `const`, `static` oder `virtual` spezifiziert werden. Was das heißt, sehen wir später.

Konstruktor

- bisher: Initialisieren der Datenelemente durch Zuweisung im Rumpf des Konstruktors.

```
Liste::Liste(int size) {
    _size = size;
    _last = 0;
    _values = new int[size];
}
```

- jetzt: Konstruktor kann zwischen Parameterliste und Funktionsrumpf eine *Initialisiererliste* enthalten.

```
Liste::Liste(int size): _last(0) {
    _size = size;
    _values = new int[size];
}
```

Sollen mehrere Attribute in der Initialisiererliste initialisiert werden, so werden die Einträge mittels Komma getrennt.

Konstruktor

Auch die eingebauten Datentypen wie `int` oder `double` haben einen Konstruktor. Diesem Konstruktor kann ein Wert übergeben werden, der initial zugewiesen wird:

```
int *ip = new int(17);
int *dynArr = new int[13];
```

- Die erste Anweisung erstellt einen Zeiger `ip` auf einen einzigen `int`-Wert, der mit 17 initialisiert wird.
- Der Zeiger `dynArr` zeigt auf einen Speicherbereich, in dem 13 Werte vom Typ `int` gespeichert werden können.

  Für jeden der 13 Werte wird der Standard-Konstruktor aufgerufen, der jeden einzelnen Wert initialisiert.

Wird ein Array von Objekten dynamisch erzeugt, so wird für jeden Wert im Array der Standard-Konstruktor der entsprechenden Klasse aufgerufen.

```
class Foo {
   int _a, _b;
   int *_x, *_y;

public:
   Foo() {
      _a = 1; _b = 2;
      _x = nullptr; _y = nullptr;
   }
   void toScreen() {
      cout << "a:" << _a << " b:" << _b << endl;
      cout << "x:" << _x << " y:" << _y << endl;
   }
};
```

```
int main(void) {
    Foo *p = new Foo[3];

    for (int i = 0; i < 3; i++)
        p[i].toScreen();
    return 0;
}
```

Als Ausgabe wird erzeugt:

```
a:1 b:2
x:0 y:0
a:1 b:2
x:0 y:0
a:1 b:2
x:0 y:0
```

Welche Ausgabe erzeugt das Programm?

```
#include <iostream>
using namespace std;

class B {
public:
    B() {
        cout << "Jetzt geht's los!" << endl;
    }
    ~B() {
        cout << "Jetzt ist Schluss!" << endl;
    }
} b;     // Definition einer globalen Variablen!

int main(void) {
    cout << "Hello World!" << endl;
}
```

Default Parameterwerte: Der Konstruktor kann wie jede andere Methode Default-Werte für die formalen Parameter haben.

- In der Header-Datei:

```
Liste(int size = 18);
```

- In der Anwendung:

```
.....
Liste l;            // Liste mit size = 18;
Liste ll(5);        // Liste mit size = 5;
.....
```

- Wird der Konstruktor mit Parameter aufgerufen, erhält die entsprechende Variable den Wert des Parameters, ansonsten wird die Variable mit dem vordefinierten Wert belegt.

Destruktor

Löschen von Objekten

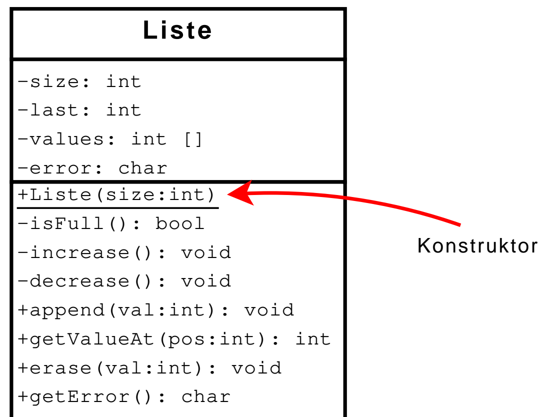
- globale Objekte implizit bei Programmende
- lokale Objekte implizit bei Prozedur- oder Blockende
- dynamisch erzeugte Objekte mit `delete`-Operator

Destruktoren

- Zweck: Aufräumarbeiten beim Löschen eines Objektes.
- Der Name des Destruktors ist *Tilde* + *Klassenname*. In unserem Beispiel der Klasse `Liste` also `~Liste()`.
- Automatischer Aufruf, wenn ein Objekt zerstört wird.
- Ergebnistyp ist ungenannt, die Parameterliste ist leer.
- Ein default-Destruktor wird vom Compiler hinzugefügt, falls kein Destruktor programmiert wurde.

Konstruktor und Destruktor in UML

In dieser Vorlesung: Konstruktoren werden in der Form `Klasse(Argumente)` dargestellt und unterstrichen.



Achtung: Entspricht nicht dem UML-Standard!

Betrachten Sie folgende C-Funktion. Was berechnet die Funktion und welcher Programmierfehler versteckt sich hier?

```
char *itoa(unsigned int val) {
    char res[12];
    int pot = 1;
    int i = 0;

    // Spezialfall val = 0 behandeln
    if (val == 0) {
        res[0] = '0';
        res[1] = '\0';
        return res;
    }
    .....
```

Wiederholung: Lokale Variablen und Zeiger

```
    while (pot <= val)
        pot *= 10;
    pot /= 10;

    while (pot > 0) {
        int ziffer = val / pot;

        res[i] = ziffer + '0';
        val -= ziffer * pot;
        pot /= 10;
        i += 1;
    }
    res[i] = '\0';

    return res;
}
```

Abgesehen vom Fehler: Ist die Funktion verständlich geschrieben?

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- *Strukturen*
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

struct in C++

Strukturen in C++ sind spezielle Klassen:

- Eine Struktur kann Methoden enthalten.
- Es sind Zugriffsbeschränkungen mittels `public`, `private` und `protected` möglich.
- `this`: Zeiger auf Strukturobjekt selbst.
- Die Memberfunktionen können überladen werden und default-Parameter besitzen.
- Es können Konstruktoren und Destruktoren definiert werden.

  Standardmäßig wird sowohl ein default-Konstruktor als auch ein default-Destruktor bereitgestellt.
- Sind wie Klassen parametrisierbar, siehe Abschnitt Templates.

Im Gegensatz zu einer C-Struktur ist kein `typedef` notwendig.

struct in C++

Unterschied zu Klassen: Alle Daten und Methoden einer Struktur sind per default `public`, in Klassen `private`.

alt:

```
typedef struct elem {
    int value;
    struct elem *next;
} elem_t;
```

neu:

```
struct elem_t {
    int value;
    elem_t *next;
};
```

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- *Wichtige Klassen*
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

Include-Dateien von C nutzen

Alte Bibliotheken aus C können in C++ weiterhin verwendet werden, allerdings wird beim Einbinden die Endung `.h` weggelassen, und stattdessen vor den Namen der Bibliothek der Buchstabe c voran gestellt.

Beispiele:

- aus `#include <math.h>` wird `#include <cmath>`
- aus `#include <stdlib.h>` wird `#include <cstdlib>`

Übersicht

Wichtige Klassen:

- `iostream`
- `iomanip`
- `string`
- `fstream`
- `sstream`

Input-Streams ermöglichen Eingaben über die Tastatur,
Output-Streams ermöglichen Ausgaben auf dem Bildschirm.

Sie werden über die Operatoren << zur Ausgabe auf dem Bildschirm und >> zum Einlesen von der Tastatur angesprochen:

```
#include <iostream>      // keine Endung .h !

int main(void) {
    int a;

    std::cout << "Bitte int-Wert eingeben: ";
    std::cin >> a;       // kein Adressoperator !

    char c = 'a';
    std::cout << c << " = " << a << std::endl;
    return 0;
}
```

Damit wir nicht überall `std::` schreiben müssen, können wir die `using namespace`-Anweisung benutzen:

```
#include <iostream>
using namespace std;        // damit kein "std::"

int main(void) {
    int a;

    cout << "Bitte int-Wert eingeben: ";
    cin >> a;

    char c = 'a';
    cout << c << " = " << a << endl;
    return 0;
}
```

Übersicht

Wichtige Klassen

- `iostream`
- `iomanip`
- `string`
- `fstream`
- `sstream`

Formatierte Ausgabe: `iomanip`

- `setw`: konstante Breite festlegen
- `setfill(char)`: Zeichen zum Auffüllen festlegen
- `setbase`: Basis der Zahl festlegen (oktal, dezimal oder hexadezimal)
- `setprecision`: Anzahl der Nachkommastellen festlegen
- `fixed`: Fließkommadarstellung
- `scientific`: Exponentialdarstellung

Formatierte Ausgabe: iomanip

```
#include .....

int main(void) {
    cout << setbase(8) << endl;
    for (int i = 1; i <= 1000000; i *= 10)
        cout << i << endl;
}
```

Ausgabe:

```
1
12
144
1750
23420
303240
3641100
```

Formatierte Ausgabe: iomanip

```
#include .....

int main(void) {
    cout << setfill('_') << endl;
    for (int i = 1; i <= 1000000; i *= 10)
        cout << setw(10) << i << endl;
}
```

Ausgabe:

```
_________1
________10
_______100
______1000
_____10000
____100000
___1000000
```

Formatierte Ausgabe: iomanip

```
#include .....

int main(void) {
    cout << setprecision(5) << endl;
    for (double d= 0.1; d >= 0.0000001; d /= 10)
        cout << d << endl;
}
```

Ausgabe:

```
0.1
0.01
0.001
0.0001
1e-05
1e-06
1e-07
```

Formatierte Ausgabe: iomanip

```
#include .....

int main(void) {
    cout << setprecision(5) << fixed << endl;
    for (double d= 0.1; d >= 0.0000001; d /= 10)
        cout << d << endl;
}
```

Ausgabe:

```
0.10000
0.01000
0.00100
0.00010
0.00001
0.00000
0.00000
```

Formatierte Ausgabe: iomanip

```
#include .....

int main(void) {
    cout << scientific << endl;
    for (double d= 0.1; d >= 0.0000001; d /= 10)
        cout << d << endl;
}
```

Ausgabe:

```
1.000000e-01
1.000000e-02
1.000000e-03
1.000000e-04
1.000000e-05
1.000000e-06
1.000000e-07
```

Übersicht

Wichtige Klassen

- `iostream`
- `iomanip`
- `string`
- `fstream`
- `sstream`

Strings

```
#include <string>
#include <iostream>
using namespace std;

int main() {
    string t("Ene mene muh und raus bist du!");

    cout << t << endl;
    cout << "size: " << t.size() << endl;
    cout << "substr: " << t.substr(13, 8)
         << endl;

    size_t pos = t.find("muh");
    cout << "substr: " << t.substr(pos) << endl;

    return 0;
}
```

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

int main() {
    std::string s("Ene mene muh");
    std::string t(" und raus bist du!");
    std::string u(" noch lange nicht");

    s.append(t);              // oder:  s += t;
    std::cout << s << std::endl;

    size_t pos = s.find("!");
    s.insert(pos, u);
    std::cout << s << std::endl;

    s.replace(pos, u.size(), " jetzt doch");
    std::cout << s << std::endl;
}
```

Wo ist der Fehler in folgendem Programm?

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

int main(void) {
    char *s = "Hallo Welt";
    char *t = strtok(s, " ");

    printf("%s\n", t);
    t = strtok(NULL, " ");
    printf("%s\n", t);

    return 0;
}
```

Übersicht

Wichtige Klassen

- `iostream`
- `iomanip`
- `string`
- `fstream`
- `sstream`

File I/O

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    fstream f;

    f.open("test1.dat", ios::app | ios::out);
    if (f.is_open())
        f.write("Hallo, Welt!\n", 13);
    else cerr << "failed to open test1.dat\n";
    f.close();

    return 0;
}
```

File I/O

Beim Öffnen verschiedene Modi mittels ODER verknüpfen:

- `ios::app` append output
- `ios::ate` seek to EOF when opened
- `ios::binary` open the file in binary mode
- `ios::in` open the file for reading
- `ios::out` open the file for writing
- `ios::trunc` overwrite the existing file

Online-Tutorials:

- http://www.cplusplus.com/
- http://www.cppreference.com/wiki/

Vereinfachung: Im Konstruktor kann die zu öffnende Datei angegeben werden, außerdem sind Operator-Überladungen definiert.

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    fstream f("test1.dat", ios::app | ios::out);

    if (!f)
        cerr << "failed to open test1.dat\n";
    else f << "Hallo, Welt!\n";
    f.close();
}
```

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    fstream f("test1.dat", ios::in);
    char line[256];         // kein string !!!!!

    if (! f.is_open()) {
        cerr << "failed to open test1.dat\n";
        return 1;
    }
    while (! f.eof()) {
        f.getline(line, 256);
        cout << line << endl;
    }
    f.close();
}
```

Dasselbe mit Strings:

```
#include .....

int main() {
    fstream f("test1.dat", ios::in);
    string line;          // string statt char *

    if (! f.is_open()) {
        cerr << "failed to open test1.dat\n";
        return 1;
    }
    while (! f.eof()) {
        getline(f, line);  // globale Funktion
        cout << line << endl;
    }
    f.close();
}
```

File I/O

Auch beim Lesen sind Operatorüberladungen definiert.
Positionieren in Dateien ist mittels `tellg` und `seekg` möglich.

```
int main() {
    char word[256];
    long start, end;
    fstream f("test1.dat", ios::in);

    start = f.tellg();
    f.seekg(0, ios::end);  // offset, direction
    end = f.tellg();
    cout << "size is: " << (end-start)
         << " bytes.\n";

    f.seekg(0, ios::beg);  // back to beginning
    f >> word;
    cout << word << endl;
}
```

Die Streams `cin`, `cout` und `cerr` kennen wir schon aus C, dort hießen sie allerdings anders:

```
fprintf(stdout, "Hallo!");   // printf("Hallo!");
fscanf(stdin, "%d", &i);     // scanf("%d", &i);
fprintf(stderr, "Hallo!");   // ????
```

Der Stream `cerr` oder `stderr` ist nicht gebuffert, so dass die Nachrichten direkt angezeigt werden, und nicht erst nach einem `endl`.

Lernkontrolle

Was wird bei dem folgenden Programm in die Datei geschrieben?

```
#include <fstream>
.....
class Klasse {
private:
   ofstream file;
public:
   Klasse(char *name = "default.txt") {
      file.open(name);
      if (!file) {
         cout << "could not open file" << endl;
         return;
      } else file << "Eins" << endl;
   }
   void function() {
      file << "Zwei" << endl;
   }
```

```
    ~Klasse() {
        file << "Drei" << endl;
        file.close();
    }
};

int main(void) {
    Klasse *k = new Klasse();

    k->function();
    return 0;
}
```

Was wird in die Datei geschrieben?

Übersicht

Wichtige Klassen

- `iostream`
- `iomanip`
- `string`
- `fstream`
- `sstream`

```
class Datum {
private:
    int tag, monat, jahr;

public:
    Datum(int t = 1, int m = 1, int j = 2000);
    Datum(string dat);     // Implementierung ?

    bool istSchaltjahr();
    int kalenderwoche();
    int tagImJahr();
    string toString();     // Implementierung ?
};

int main() {
    Datum d(7, 4, 2009);
    cout << d.toString() << endl;
}
```

So funktioniert es leider nicht:

```
string toString() {
    string str;

    str += tag;
    str += ".";
    str += monat;
    str += ".";
    str += jahr;

    return str;
}
```

Die Attribute `tag`, `monat` und `jahr` sind vom Typ `int`, für den weder eine Operatorüberladung noch eine Methode `append` in der Klasse `string` definiert ist!

So funktioniert es:

```
string toString() {
    ostringstream os;

    os << setw(2) << tag << ".";
    os << setw(2) << monat << ".";
    os << setw(4) << jahr;

    return os.str();
}
```

Ein Objekt der Klasse `ostringstream` verhält sich wie ein Objekt der Klasse `ostream`, nur das die Werte nicht auf dem Bildschirm ausgegeben werden sondern in einen `string` geschrieben werden.

Damit obiger Code funktioniert, müssen die Header `sstream` und `iomanip` eingebunden werden.

Wir können auch einen Konstruktor definieren, der die Daten aus einem `string` ausliest:

```
Datum(string dat) {
    istringstream is(dat);
    char t;

    is >> tag;
    is >> t;
    is >> monat;
    is >> t;
    is >> jahr;
}
```

Der Aufruf `Datum d("27.3.2011");` würde jeweils den Punkt in der Variablen `t` speichern und verwerfen, Tag, Monat und Jahr werden in die entsprechenden Klassenattribute eingetragen.

Streams

Würde das obige Beispiel auch ohne die Variable `t` funktionieren?

```
Datum(string dat) {
    istringstream is(dat);

    is >> tag >> monat >> jahr;
}
```

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- *Statische Attribute*
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

Statische Attribute

bisher:

- Attribute sind die Zustandsvariablen der Objekte einer Klasse.

→ Jede Zustandsvariable ist an ein Objekt gebunden und daher nur mit dem Objekt existent.

statische Attribute:

- Durch `static` sind die Attribute nicht objektbezogen.
- Die Werte sind für alle Objekte einer Klasse gleich.
- `static`-Attribute müssen (außer `const static`) außerhalb der Klassendeklaration mit einem Anfangswert definiert werden.
- Sie haben eine durchgehende Lebensdauer.

Anwendung: Um globale, objektunabhängige Daten zu definieren.

- Grundgebühr bei Telefonanschlüssen
- fortlaufende Nummern (vgl. Sequenz in SQL)
- Entwurfsmuster Singleton (später)

bisher: Wir würden dem Konstruktor einer Klasse `Konto` den Wert des Attributs `nr` als Parameter übergeben. Das Konto erhält damit die entsprechende Kontonummer von außen. Irgendwo im Programm muss die fortlaufende Nummer verwaltet werden.

```
Konto::Konto(string inhaber, int nr, int pin) {
    stand = 0;
    this->inhaber = inhaber;
    this->nr = nr;              // !!!!!!!!!!!!!!!
    this->pin = pin;
}
```

jetzt: Die Klasse `Konto` ist für die fortlaufende Vergabe der Kontonummern verantwortlich.

konto.h

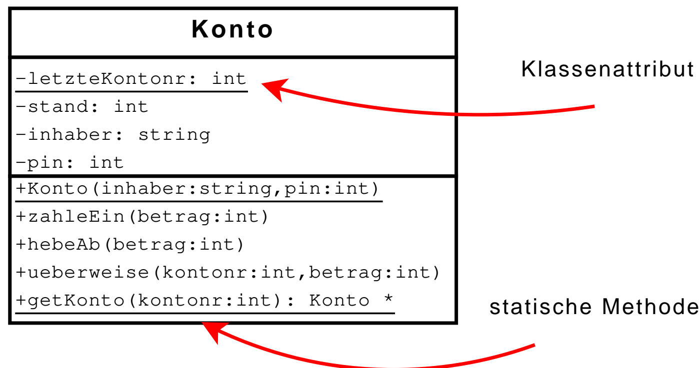
```
class Konto {
private:
    static int neueKontonr;
    ....
public:
    Konto(string inhaber, int pin) : stand(0) {
        this->inhaber = inhaber;
        nr = Konto::neueKontonr++;   // !!!!!!!
        this->pin = pin;
    }
    ...
};
```

konto.cpp

```
int Konto::neueKontonr = 1;
```

Statische Attribute in UML

Statische Attribute und Methoden werden unterstrichen.



Anmerkung: Die Methode `getKonto` soll zu einer Kontonummer das entsprechende Konto-Objekt liefern. Daher kann die Methode nicht mit einem Objekt der Klasse `Konto` aufgerufen werden und ist deshalb als statisch zu deklarieren.

Statische Methoden

Statische Methoden werden oft anstelle von Konstruktoren verwendet:

```
class Date {
private:
    int day, month, year;
public:
    Date(int d, int m, int y) {
        day = d;
        month = m;
        year = y;
    }
    static Date getCurrentDate();    // !!!!!
    .....
};
```

In statischen Methoden steht die implizite Objektreferenz `this` nicht zur Verfügung, da die Methode anhand des Klassennamens, und nicht anhand eines Objekts aufgerufen wird.

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- Statische Attribute
- *Ausnahmebehandlung*
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

Exceptions – Ausnahmebehandlung

Zur Laufzeit eines Programms können Fehlersituationen auftreten, die eine weitere Programmausführung nur bedingt oder gar nicht mehr erlauben:

- Division durch 0
- Überlauf (zu kleiner/großer Wert für einen Datentypen)
- nicht genügend Speicherplatz vorhanden (`malloc`)
- fehlerhafte Eingaben durch Benutzer
- Zugriff auf ungültige Adressen im Hauptspeicher usw.

Frage: Wie erfolgte in C eine Ausnahmebehandlung? Wie konnte in C festgestellt werden, ob eine Funktion korrekt beendet wurde oder nicht?

Die Funktion kann einen speziellen Fehlerwert zurückliefern, der im Programm mittels `if` abgefragt wird.

```
.....
int find(list_t *l, int val) {
    for (int pos = 0; pos < l->last; pos++)
        if (l->values[pos] == val)
            return pos;
    return -1;
}
.....
pos = find(myList, 10);
if (pos < 0)
    printf("Value not found\n");
else .....
```

Leider gibt es Funktionen, bei denen jeder Rückgabewert gültig ist, z.B. Funktionen aus `math.h` wie `pow` oder `ldexp`.

Exceptions – Ausnahmebehandlung

Bei einigen Funktionen in C wird daher nachträglich abgefragt, ob die zuletzt ausgeführte Operation erfolgreich war.

```
....
int main(void) {
    list_t *l;
    ....

    for (i = 0; i < 20; i++)
        append(l, i);

    i = getValueAt(l, 30);    // Fehlerwert ?????
    if (getError(l) == 0)     // Fehlerbehandlung
        printf("value[%2d] = %2d\n", 30, i);
    ....
}
```

Leider wird das oft vergessen und führt so zu Programmabbrüchen.

In C können Signal-Handler mittels `signal` eingerichtet werden.

```
#include <signal.h>
#include .....

void nullDiv(int sig) {
    printf("0-Division\n");
    exit(1);
}

void main(void) {
    int z, erg;

    signal(SIGFPE, nullDiv);

    scanf("%d", &z);
    erg = 123 / z;
    printf("123 / %d = %d\n", z, erg);
}
```

Exceptions – Ausnahmebehandlung

In C++ gibt es eine spezielle Fehlerbehandlung:

- Wir klammern die Anweisungen, in denen Exceptions, also Ausnahmen, evtl. Fehler auftreten können, mit einem `try`-Block.
- In den `catch`-Block, der dem `try`-Block unmittelbar folgt, schreiben wir die Anweisungen, die beim Auftreten von Exceptions ausgeführt werden sollen.
- Die `throw`-Anweisung löst eine Exception aus:
  - `const char *`-Exception: `throw "Division durch 0";`
  - `int`-Exception: `throw 4711;`

→ Der Code zur „normalen“ Programmausführung ist vom Code zur Fehlerbehandlung getrennt. Man verspricht sich davon eine bessere Lesbarkeit des Codes!

```
#ifndef _EXCEPTION_H
#define _EXCEPTION_H

#include <string>

class Exception {
private:
    std::string _error;

public:
    Exception(std::string error);
    std::string toString();
};

#endif
```

exception.h

exception.cpp

```
#include "exception.h"
using namespace std;

// Konstruktor
Exception::Exception(string error) {
    _error = error;
}

// zur Ausgabe
string Exception::toString() {
    return _error;
}
```

Liste mit Exceptions

liste.cpp

```
#include "liste.h"
#include "exception.h"

.....

int Liste::getValueAt(int idx) {
    if (idx < 0 || idx >= _last)
        throw Exception("out of bounds");
    return _values[idx];
}
```

Die Abarbeitung der Methode `getValueAt` wird durch das `throw` abgebrochen, falls ein unzulässiger Index angeben wird. Es erfolgt dann unmittelbar ein Rücksprung an die aufrufende Methode.

Wie im aufrufenden Programmteil eine solche Ausnahme abgefangen werden kann, zeigt die nächste Folie:

Liste mit Exceptions

main.cpp

```
#include .....
using namespace std;

int main(void) {
    Liste l(10);

    for (int value = 3; value < 8; value++)
        l.append(value);

    try {
        for (int i = 0; i < 8; i++)
            cout << i + 1 << ":  "
                 << l.getValueAt(i) << endl;
    } catch (Exception e) {
        cout << e.toString() << endl;
    }
}
```

Eine Schachtelung von `try`-Blöcken ist erlaubt.

Wenn eine Exception außerhalb eines `try`-Blocks auftritt, gilt:

- Es wird die Funktion `terminate()` aufgerufen, die standardmäßig die Funktion `abort()` aufruft, die das Programm beendet.
- Wenn unser Programm vor dem Halten noch etwas anderes tun soll, können wir eine andere Funktion als `abort()` hinterlegen.

  Dazu rufen wir die Funktion `set_terminate()` auf, der wir einen Zeiger auf eine Funktion übergeben.

Exceptions – Ausnahmebehandlung

Für jeden möglichen Typ, der geworfen werden kann, müssen wir einen entsprechenden `catch`-Block angeben:

```
try {
    // some statements
} catch (OutOfBoundsException e) {
    // other statements
} catch (OverflowError e) {
    // more statements
} catch (BadAllocException e) {
    // even more statements
} catch (...) {  // Syntax korrekt!!!!!
    // handle any other exception!!!!!!
}
```

- Die Syntax ist analog zu einer Funktionsdeklaration.
- Der Parameter `e` kann weggelassen werden, falls der Wert innerhalb der geschweiften Klammern nicht benötigt wird.

Exceptions – Ausnahmebehandlung

In C++ sind einige Klassen zur Ausnahmebehandlung bereits vorhanden:

- `#include <exception>` stellt die Basisklasse `exception` bereit. (Basisklasse: siehe Kapitel Vererbung)
- `#include <new>` stellt die Klasse `bad_alloc` bereit, die geworfen wird, wenn kein Speicher bereit gestellt werden konnte.
- `#include <typeinfo>` stellt die Klassen `bad_cast`, `bad_typeid` bereit.
- `#include <stdexcept>` stellt die Klassen `domain_error`, `invalid_argument`, `length_error` und `out_of_range` bereit.
- Jede dieser Klassen implementiert die Methode `what()`, die ein `const char*` liefert, also eine C-Zeichenkette, die den Fehler näher beschreibt.

Wie geht es weiter?

Anmerkung zur Liste:

- Unsere Liste kann aufgrund der Typisierung nur `int`-Werte speichern.

→ Eine Wiederverwendung ist daher nur durch Copy-and-Paste möglich: Erstelle für jeden Datentypen eine eigene Liste.

Copy-and-Paste ist natürlich keine Lösung. Alternativen

- in C:
  - Datentyp `int` durch `void *` ersetzen oder
  - Funktionen durch parametrisierte Makros ersetzen.
- in C++: Generische Programmierung mittels Templates.

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- *Generische Programmierung*
- Referenzen
- Kopieren von Objekten
- Operatorüberladung

main.c

```
#include <stdio.h>
#include "liste.h"

int main(void) {
    int i;
    list_t *li, *ld;
    // list of int-values !!!!!
    li = create(2, sizeof(int));
    for (i = 1; i <= 10; i++)
        append(li, &i);

    for (i = 0; i < 20 && !getError(li); i++) {
        void *val = getValueAt(li, i);

        if (getError(li) == 0)
            printf("%d: %d\n", i, *(int *)val);
    }
    destroy(li);
```

Liste in C

```c
    float f;
    // list of float-values !!!!!
    ld = create(2, sizeof(float));
    for (i = 1, f = 1.25; i <= 10;
            i++, f += 0.25)
        append(ld, &f);

    for (i = 0; i < 20 && !getError(ld); i++) {
        void *val = getValueAt(ld, i);
        float fval = *(float *)val;

        if (getError(ld) == 0)
            printf("%d: %f\n", i, fval);
    }
    destroy(ld);

    return 0;
}
```

Liste in C

Damit wir Werte eines beliebigen Datentyps in der Liste speichern können, definieren wir ein dynamisches Array, dass an jedem Index einen Zeiger auf `void` speichert.

liste.h

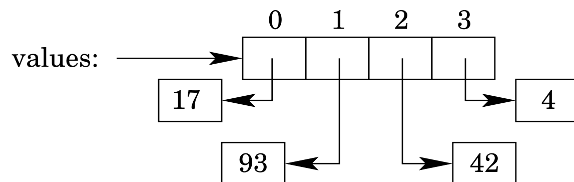
```
/*
 * incomplete data type
 */
typedef struct list list_t;

/*
 * interface
 */
list_t * create(int nmemb, int esize);
void append(list_t *l, void *val);
void * getValueAt(list_t *l, int pos);
char getError(list_t *l);
void destroy(list_t *l);
```

Anmerkungen:

- Der Parameter `val` der Funktion `append()` ist nun vom Typ 'Zeiger auf `void`'.
- Passend dazu ist der Typ der Funktion `getValueAt()` nun ebenfalls 'Zeiger auf `void`'.
- Der Funktion `create()` muss nun nicht nur die initiale Größe des Arrays übergeben werden, es muss auch die Größe der einzelnen, zu speichernden Elemente übergeben werden.

  Dies ist notwendig, damit die Funktion `append` entsprechend viel Speicherplatz für die Kopie des zu speichernden Wertes allokieren kann.

Der Typ des Attributes `values` ist nun 'Zeiger auf Zeiger auf `void`', denn es soll ein dynamisches Array (erster Zeiger) erzeugt werden, dass an jedem Index einen Zeiger auf `void` (zweiter Zeiger) speichert.



Ein 'Zeiger auf `void`' hat auf jedem Rechner eine feste Größe, daher kann die Größe des Speichers, die das Array belegt, berechnet werden.

liste.c

```c
#include <stdlib.h>
#include <string.h>
#include "liste.h"

struct list {
    void **value;
    int nmemb;
    int last;
    char error;
    int esize;
};

static void increase(list_t *l) {    // private
    l->nmemb *= 2;
    l->value = (void **) realloc(l->value,
            l->nmemb * sizeof(void *));
}
```

Liste in C

```
list_t * create(int nmemb, int esize) {
    list_t *l;

    l = (list_t *) malloc(sizeof(list_t));
    l->last = 0;
    l->nmemb = nmemb;
    l->esize = esize;
    l->error = 0;
    l->value = (void **) calloc(nmemb,
            sizeof(void *));

    return l;
}

char getError(list_t *l) {
    return l->error;
}
```

```
static char isFull(list_t *l) {      // private
    return l->last == l->nmemb;
}

void append(list_t *l, void *val) {
    void *elem;

    if (isFull(l))
        increase(l);

    elem = malloc(l->esize);
    memcpy(elem, val, l->esize);
    l->value[l->last] = elem;
    l->last += 1;
}
```

Die Funktion `append()` prüft zunächst, ob noch Platz im Array vorhanden ist und allokiert ggf. mehr Speicher durch aufrufen der Funktion `increase()`.

Danach wird Speicher allokiert, um eine Kopie des zu speichernden Wertes ablegen zu können. Die Kopie wird mittels der Funktion `memcpy` aus der Standard-Bibliothek erzeugt, die byte-weise den Speicher, auf den `val` zeigt, an die Stelle kopiert, auf die `elem` zeigt.

Übung: Warum wird nicht einfach der Zeiger `val`, also die Speicheradresse, auf die `val` zeigt, gespeichert? Warum wird Speicherplatz allokiert und eine Kopie erzeugt?

Die Antwort kommt später.

```
void * getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->last) {
        l->error = 1;
        return NULL;
    }
    return l->value[pos]; // oder Kopie liefern?
}

void destroy(list_t *l) {
    int i;

    for (i = 0; i < l->last; i++)
        free(l->value[i]);
    free(l->value);
    free(l);
}
```

Liste in C

Wenn wir aus der Funktion `append()` (wie oben) *keine* Kopie des Wertes zurück geben, wird das Prinzip der Datenkapselung verletzt:

```
.....
int main(void) {
    int i;
    list_t *l = create(2, sizeof(int));

    .....
    for (i = 0; i < 20 && !getError(l); i++) {
        void *val = getValueAt(l, i);

        if (getError(l) == 0)
            *(int *)val = 42;   // !!!!!!
    }
    .....
}
```

Wie kann eine Kopie zurück gegeben werden?

Im einfachsten Fall ändern wir die Funktion `getValueAt()` unserer Liste wie folgt:

```
void * getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->last) {
        l->error = 1;
        return NULL;
    }

    // Kopie erzeugen !!!!!
    void *result = malloc(esize);
    memcpy(result, values[i], esize);
    return result;
}
```

Problem: Wer gibt den Speicher wieder frei, der mit `malloc` allokiert wurde?

Liste in C

Um das Problem der Speicherfreigabe zu vermeiden, legen wir in unserer Struktur ein weiteres Attribut `result` an und allokieren Speicher für die Kopie in der Funktion `create()`:

```
struct list {
    void **value;
    .....
    int esize;
    void *result;                    // neu !!!!!
};

list_t * create(int nmemb, int esize) {
    list_t *l= (list_t *) malloc(sizeof(list_t));

    l->result = malloc(esize);   // neu !!!!!
    return l;
}
```

Liste in C

Wir belegen den Speicher in der Funktion `append()`, und geben den Speicher in der Funktion `destroy` wieder frei:

```
void *getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->last) {
        .....    // Fehlerbehandlung
    }
    memcpy(l->result, l->values[pos], l->esize);
    return l->result;
}

void destroy(list_t *l) {
    for (int i = 0; i < l->last; i++)
        free(l->value[i]);
    free(l->value);
    free(l->result);    // neu !!!!!
    free(l);
}
```

Liste in C

Nachteile einer solch generischen Lösung:

- Keine Typsicherheit, da Zeiger vom Typ `void *` mit allen Zeigertypen kompatibel sind.
  Der Vorteil, dass der Compiler für uns Überprüfungen auf Datentyp-Verträglichkeit vornehmen kann, geht verloren.
- Komplizierte Syntax durch explizite Typumwandlungen (type cast) beim Auslesen der Daten aus der Datenstruktur.

In C++ geht das alles viel eleganter.

Templateklasse Liste

liste.h

```
#include "exception.h"

template <typename T>
class Liste {
    T *_values;
    int _last, _size;
    bool isFull();
    int find(T val);
    void increase();
    void decrease();
public:
    Liste(int size);
    ~Liste();
    void append(T val);
    T getValueAt(int pos);
    void erase(T val);
    void toScreen();
};
```

Templateklasse Liste

```
template <typename T>
Liste<T>::Liste(int size) {
    _size = size;
    _last = 0;
    _values = new T[size];
}

template <typename T>
Liste<T>::~Liste() {
    delete[] _values;
}

template <typename T>
T Liste<T>::getValueAt(int pos) {
    if (pos < 0 || pos >= _last)
        throw Exception("out of bounds");
    return _values[pos];
}
```

Templateklasse Liste

```
template <typename T>
void Liste<T>::append(T val) {
    if (isFull())
        increase();

    _values[_last] = val;
    _last += 1;
}

template <typename T>
bool Liste<T>::isFull() {
    return _last == _size;
}
```

```
template <typename T>
void Liste<T>::increase() {
    T *tmp = new T[_size * 2];

    for (int i = 0; i < _size; i++)
        tmp[i] = _values[i];

    delete[] _values;
    _values = tmp;
    _size *= 2;
}

template <typename T>
void Liste<T>::toScreen() {
    for (int i = 0; i < _last; i++)
        cout << i << ": " << _values[i] << endl;
}
```

Templateklasse Liste

```
template <typename T>
int Liste<T>::find(T val) {
    for (int pos = 0; pos < _last; pos++)
        if (_values[pos] == val)
            return pos;
    return -1;
}

template <typename T>
void Liste<T>::decrease() {
    _size /= 2;
    T *tmp = new T[_size];

    for (int i = 0; i < _size; i++)
        tmp[i] = _values[i];
    delete[] _values;
    _values = tmp;
}
```

Templateklasse Liste

```
template <typename T>
void Liste<T>::erase(T val) {
    int pos = find(val);

    if (pos == -1)
        throw Exception("value not found");

    for (; pos < _last -1; pos++)
         _values[pos] = _values[pos + 1];
    _last -= 1;

    if (_last < _size / 4)
        decrease();
}
```

Templateklasse Liste

main.cpp

```
#include .....

int main(void) {
    Liste<int> l(4);

    for (int i = 1; i <= 20; i++)
        l.append(i);
    l.toScreen();

    try {
        for (int i = 1; i <= 20; i += 2)
            l.erase(i);
        l.toScreen()
    } catch (Exception e) {
        cout << e.toString() << endl;
    }
```

Templateklasse Liste

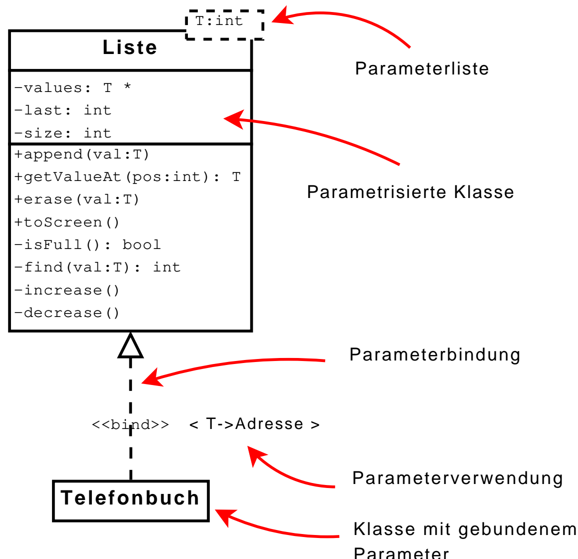
```
    Liste<float> ll(4);
    float f;

    for (f = 1.25; f <= 5.5; f += 0.25)
        ll.append(f);
    ll.toScreen();

    try {
        for (f = 1.5; f <= 5.5; f += 0.5)
            ll.erase(f);

        for (int i = 0; i <= 20; i++)
            cout << ll.getValueAt(i) << endl;
    } catch (Exception e) {
        cout << e.toString() << endl;
    }
}
```

Templateklasse Liste

Anmerkungen zur Liste:

- Die Listen-Implementierung muss in die Header-Datei!
- Die Datei `liste.cpp` entfällt.
- Eine konkrete Liste speichert immer nur einen einzigen Datentyp! $\rightarrow$ *Homogene Datenstruktur*

Templateklassen in UML



Generische Programmierung

Templates:

- Parametrisierung von Funktionen und Klassen.
- Beschreibung von Datenstrukturen und Algorithmen unabhängig von bestimmtem Typ. (in C: `void *`)
- Durch Instanziierung konkrete Formulierung erzeugen: An die Stelle des unbestimmten Typs tritt ein konkreter Typ.

Vorteile:

- Instanziierung durch den Compiler.
- Typprüfung zur Compile-Zeit $\rightarrow$ große Typsicherheit
- Die verwendeten Typen müssen nicht auf einer allgemeinen Basisklasse beruhen. (siehe Kapitel Vererbung)

Nachteile:

- Größe des Codes: Jede Schablonen-Instanziierung führt zu weiterem Objekt-Code, anders als bspw. bei Java Generics.

Anwendung:

- Überall dort, wo der Algorithmus nicht von den Daten abhängt.

Verwendet wird heute das Schlüsselwort `typename`, veraltet ist dagegen `class`.

Template der Klassendeklaration

```
template <typename T>
class Liste {
private:
    T *values;
    ...
public:
    ...
    void append(T val);
    T getValueAt(int pos);
};
```

- Der Name `T` für den Platzhalter hat sich eingebürgert.
- Eine Instanziierung erfolgt erst, wenn Objekte vom Typ `Liste<T>` für einen konkreten Typ `T` deklariert werden:

```
Liste<int> iList;
```

Template der Methodenimplementierung

```
template <typename T>
int Liste<T>::getSize() {
    return size;
}

template <typename T>
T Liste<T>::getValueAt(int pos) {
    return values[pos];
}
```

Wichtig: Die Methoden beziehen sich auf die parametrisierte Liste, daher muss `Liste<T>::` vor jeder Methode angegeben werden.

Werden die Methoden direkt bei deren Deklaration implementiert, entfällt die Angabe `template <typename T>` vor jeder Methode.

Funktions-Templates (keiner Klasse zugehörig)

```
template <typename T>
T summe(T *array, int n) {
    T sum = array[0];
    for (int i = 1; i < n; i++)
        sum = sum + array[i];
    return sum;
}
...
int a[] = {1, 1, 2, -3, 2};
int b[] = {11, 2, 3, 2, 7, 5};
double c[] = {1.1, 1.001, -12.8};

cout << summe<int>(a,5) << endl;
cout << summe<int>(b,6) << endl;
cout << summe<double>(c,3) << endl;
```

Funktions-Templates:

- mehrere Parameter: spezifiziere Template-Argumente in der Reihenfolge, in der die Parameter deklariert sind.

```
template<typename S, typename T>
void func(S x, T y, S* z) { ... }
```

- Spezifizierer wie `inline` oder `friend` stehen hinter `template<...>`

- Template-Funktionen können überladen werden:

```
template<typename T> T methode(T);
template<typename T> T methode(T, int);
```

Die Syntax hat aber Tücken: Folgender Code wird nicht kompiliert!

```
#include <iostream>
#include <map>
using namespace std;

template <typename T, typename S>
S summe(map<T, S> &toAdd) {
    map<T, S>::iterator it = toAdd.begin();

    if (it == toAdd.end())
        throw "empty map exception";

    S sum = it->second;
    for ( ; it != toAdd.end(); it++)
        sum += it->second;
    return sum;
}
```

```
int main(void) {
    map<string, double> aMap;
    aMap["eins"] = 1.0;
    aMap["zwei"] = 2.0;
    aMap["drei"] = 3.0;

    cout << "Summe: " << summe(aMap) << endl;
}
```

Bei der Deklaration des Iterators fehlt die Angabe `typename`. Es muss heißen:

```
typename map<T, S>::iterator it = toAdd.begin();
```

Aber warum?

Die folgende Funktion möchte eine innere Klasse `iterator` der Klasse `T` nutzen, um irgendwelche sinnvollen Dinge zu tun:

```
template <typename T>
void foo() {
    T::iterator *iter;
    .....
}
```

Wir gehen also davon aus, dass es eine Klasse wie `Blubb` gibt, die eine innere Klasse `iterator` hat und wir `foo<T>()` mit dieser Klasse parametrisieren möchten:

```
class Blubb {
    class iterator { ..... };
    .....
};
foo<Blubb>();
```

Was wir eher nicht erwarten, ist, dass `iterator` eine Variable der Klasse ist:

```
class BlaBlubb {
    static int iterator;
    .....
};
```

Parametrisieren wir die Funktion `foo<T>()` mit dieser Klasse, dann gibt es ein Problem: Die Zeile

```
T::iterator *iter;
```

wird zu

```
BlaBlubb::iterator *iter;
```

und der Compiler interpretiert `iter` nicht als Zeiger auf eine innere Klasse, sondern die Zeile als Multiplikation, was zu einem Fehler führt!

Was gemeint ist, kann also erst bei der Exemplifizierung (auch Instanziierung genannt) entschieden werden.

Anstatt die Interpretation bis zur Exemplifizierung aufzuschieben, wurde festgelegt:

> Fehlt das Schlüsselwort `typename`, dann werden qualifizierte, abhängige Namen (qualified dependent names) nicht als Typen interpretiert, auch wenn das zu einem Fehler führt.

Ein abhängiger Name ist ein Name, der von einer Parametrisierung (also einem Template) abhängig ist.

Ein qualifizierter Name ist z.B. `std::cout`, wenn wir allerdings mit einer `using`-Direktive arbeiten, ist `cout` ein nicht-qualifizierter Name.

String-Tokenizer

Unsere parametrisierte Liste können wir nutzen, um einen String-Tokenizer zu erstellen.

main.cpp

```
#include <iostream>
#include "tokenizer.h"
using namespace std;

int main(void) {
    string str = "Hans Meier:Gabi Fischer:"
                 "Franz Schulz:Anne Mayer:";
    Tokenizer tok(str, ":;,");

    while (tok.hasMoreTokens()) {
        string mitarb = tok.nextToken();
        .....
    }
    return 0;
}
```

tokenizer.h

```
#include <string>
#include "liste.h"
using namespace std;

class Tokenizer {
private:
    unsigned int _pos;
    Liste<string> _tokens;

public:
    Tokenizer(string data, string separators);

    int countTokens();
    string nextToken();
    bool hasMoreTokens();
};
```

tokenizer.cpp

```
#include "tokenizer.h"
using namespace std;

Tokenizer::Tokenizer(string data, string sep) {
    _pos = 0;

    string::size_type beg, end;
    beg = data.find_first_not_of(sep, 0);
    end = data.find_first_of(sep, beg);

    while (string::npos != beg
            || string::npos != end) {
        string s = data.substr(beg, end - beg);
        _tokens.append(s);
        beg = data.find_first_not_of(sep, end);
        end = data.find_first_of(sep, beg);
    }
}
```

String-Tokenizer

```
int Tokenizer::countTokens() {
    return _tokens.size();
}

string Tokenizer::nextToken() {
    return _tokens.getValueAt(_pos++);
}

bool Tokenizer::hasMoreTokens() {
    return _pos < _tokens.size();
}
```

Reicht der Standard-Destruktor? Werden damit auch die Einträge in der Liste gelöscht?

Übung

Damit der obige String-Tokenizer funktioniert, müssen wir unsere Klasse `Liste` um eine Methode erweitern:

- `size()` liefert die Anzahl der gespeicherten Elemente

Überlegen Sie, in welchen Fällen der obige String-Tokenizer nicht korrekt funktioniert.

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- *Referenzen*
- Kopieren von Objekten
- Operatorüberladung

Referenzen erübrigen die Übergabe von Zeigern, wenn Werte innerhalb einer Funktion geändert werden sollen:

```
void swap(int &x, int &y) {
    int t = x;

    x = y;
    y = t;
}

int main(void) {
    int a, b;
    ...
    swap(a, b);
    ...
}
```

Pointer sind klarer und sollten vorgezogen werden: Bei Verwendung von Pointern ist für den *Anwender* einer Funktion ersichtlich, dass die Werte in der Funktion verändert werden können, vergleiche dazu die Aufrufe `swap(a, b)` und `swap(&a, &b)`.

Bjarne Stroustrup, the designer and original implementor of C++: Consequently "plain" reference arguments should be used only when the name of the function gives a strong hint that the reference argument is modified.

Referenzen sind implementiert über die Verwendung von konstanten Zeigern!

Übung: Voriges `swap(int &x, int &y)` entspricht welcher der folgenden Implementierungen?

```
void swap1(const int *x, const int *y) {
    ...            // Implementierung siehe swap3

void swap2(int const *x, int const *y) {
    ...            // Implementierung siehe swap3

void swap3(int * const x, int * const y) {
    int t = *x;
    *x = *y;
    *y = t;
}
```

Referenzen

Eine Referenz ist ein alternativer Name für eine Variable, Konstante oder Funktion.

```
// typische Deklaration:
T & Bezeichner = Ausdruck;
// Beispiel:
const int &x = 4711;
```

Einige Aspekte von Referenzen:

- Referenzen müssen bei ihrer Deklaration initialisiert werden.
- Referenzen können nicht durch `new` erzeugt werden.
- Zeiger auf Referenzen sind nicht möglich.
- Felder von Referenzen sind nicht möglich.
- Es gibt keine Referenzen auf Referenzen.
- Es gibt keine Referenzen auf `void`.

Lernkontrolle

Ist das folgende Programm korrekt? Falls ja, welche Ausgabe erzeugt es?

```
#include <iostream>
using namespace std;

int summe(int &a, int &b) {
    return a + b;
}

int main() {
    cout << summe(1, 2) << endl;
}
```

Lernkontrolle

Gegeben seien folgende Funktionen:

```
int  g(int);
int& h(const int &);
```

Welche der folgenden Aufrufe der obigen Funktionen sind richtig?

		richtig	falsch
(a)	`int i = ++g(42);`	☐	☐
(b)	`int i = ++h(42);`	☐	☐
(c)	`int *p = &g(42);`	☐	☐
(d)	`int *p = &h(42);`	☐	☐
(e)	`g(42) = 42;`	☐	☐
(f)	`h(42) = 42;`	☐	☐
(g)	`int i = g(h(42));`	☐	☐
(h)	`int i = h(g(42));`	☐	☐

Lernkontrolle

Gegeben seien folgende Funktionen:

```
void f(int);
void g(int &);
void h(const int &);
```

Welche Aufrufe der Funktionen sind richtig, falls `i` eine Variable vom Typ `int` ist?

		richtig	falsch
(a)	`f(7);`	☐	☐
(b)	`f(i);`	☐	☐
(c)	`g(7);`	☐	☐
(d)	`g(i);`	☐	☐
(e)	`h(7);`	☐	☐
(f)	`h(i);`	☐	☐

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- *Kopieren von Objekten*
- Operatorüberladung

Kopieren von Objekten

In folgenden Fällen werden die Attributwerte eines Objekts in ein anderes Objekt derselben Klasse kopiert:

- Initialisieren eines Objekts mit einem anderen Objekt derselben Klasse: Zuweisung bei Deklaration.
- Übergabe eines Objekts als Argument bei einem Funktionsaufruf, nicht bei Zeigern oder Referenzen!
- Rückgabe eines Objekts als Funktionswert.

Dabei wird nicht der Konstruktor aufgerufen, sondern der *Copy-Konstruktor*!

Syntax:

```
Klasse(const Klasse &zuKopierendesObjekt)
```

Kopieren von Objekten

Wird kein Copy-Konstruktor programmiert, dann erzeugt der Compiler einen *default copy constructor*:

- Alle Werte des Objekts werden elementweise kopiert, unabhängig davon, ob es Adressen sind.

$\rightarrow$ Verweise im Original und in der Kopie sind identisch!

Eine solche Kopie wird als *flache Kopie* bezeichnet.

Soll eine echte Kopie unserer Liste erzeugt werden, muss das Array, das die eigentlichen Werte enthält, kopiert werden.

Kopieren von Objekten

```
.....
template <typename T>
Liste<T>::Liste(const Liste& l) {
    _size = l._size;
    _last = l._last;
    _values = new T[_size];

    for (int i = 0; i < _last; i++)
        _values[i] = l._values[i];
}
```

Lernkontrolle

Welche Ausgabe erzeugt das Programm?

```
#include <iostream>

class CTest {
public:
    CTest() { std::cout << "K\n"; }
    ~CTest() { std::cout << "D\n"; }
    CTest(const CTest &c) { std::cout << "C\n"; }
};

void down(CTest t) { return; }

int main(void) {
    CTest u;
    CTest v = u;
    down(u);
}
```

Anmerkungen:

- Ein Copy-Konstruktor kann auch explizit aufgerufen werden:

```
CTest *t = new CTest(orig);
```

- Weitere Parameter sind möglich, sie müssen aber alle default-Werte haben.
- Der Copy-Konstruktor kann wie andere Konstruktoren eine Initialisierungsliste haben.

Übersicht

Grundlagen C++

- Klassen und Objekte
- Konstruktor und Destruktor
- Strukturen
- Wichtige Klassen
- Statische Attribute
- Ausnahmebehandlung
- Generische Programmierung
- Referenzen
- Kopieren von Objekten
- *Operatorüberladung*

Rationale Zahlen in C mit modularer Programmierung:

rational.h

```
// incomplete data type
typedef struct rat rat_t;

// interface
rat_t *createRat(int zaehler, int nenner);
void destroyRat(rat_t *r);

rat_t *addRat(rat_t *a, rat_t *b);  // +
rat_t *subRat(rat_t *a, rat_t *b);  // -
rat_t *mulRat(rat_t *a, rat_t *b);  // *
rat_t *divRat(rat_t *a, rat_t *b);  // /
.....
void printRat(rat_t *a);
```

Aufruf mit:

```
rat_t *a, *b, *c; *d;
a = createRat(1, 2);
b = createRat(3, 7);

c = addRat(a, b);
d = mulRat(a, b);
printRat(c);
if (isGreater(c, d))
    ...
else ...
.....
```

Schöner wäre:

```
Rational a(1, 2);
Rational b(3, 7);
Rational c, d;

c = a + b;
d = a * b;
cout << c << endl;
if (c > d)
    ...
else ...
.....
```

rational.h

```
#include <iostream>
using namespace std;

class Rational {
    friend ostream& operator<<(ostream& os,
            const Rational& x) {
        os << x.zaehler << "/" << x.nenner;
        return os;
    }
private:
    int zaehler, nenner;
    Rational add(Rational x);  // Addition
    Rational sub(Rational x);  // Subtraktion
    Rational mul(Rational x);  // Multiplikation
    Rational div(Rational x);  // Division
    void kehrwert();
    void kuerzen();
```

```
public:
    Rational(int z = 1, int n = 1);

    Rational operator-();      // Vorzeichen
    Rational operator+(const Rational& x);
    Rational operator-(const Rational& x);
    Rational operator*(const Rational& x);
    Rational operator/(const Rational& x);

    bool operator<(const Rational& x);
    bool operator>(const Rational& x);
    bool operator==(const Rational& x);

    int getZaehler();
    int getNenner();
};
```

rational.cpp

```
#include "rational.h"

// Konstruktor
Rational::Rational(int z, int n) {
    zaehler = z;
    nenner = n;
    kuerzen();
}

void Rational::kehrwert() {
    int t = zaehler;

    zaehler = nenner;
    nenner = t;
}
```

Rational Zahlen in C++

```
Rational Rational::add(Rational x) {
    Rational r;

    r.zaehler = zaehler * x.nenner
              + nenner * x.zaehler;
    r.nenner = nenner * x.nenner;
    r.kuerzen();

    return r;
}

Rational Rational::sub(Rational x) {
    return add(-x);
}
```

```
Rational Rational::mul(Rational x) {
    return Rational(zaehler * x.zaehler,
                    nenner * x.nenner);
}

Rational Rational::div(Rational x) {
    x.kehrwert();
    return mul(x);
}

int Rational::getZaehler() {
    return zaehler;
}

int Rational::getNenner() {
    return nenner;
}
```

Rationale Zahlen in C++

```
Rational Rational::operator-() {    // Vorzeichen
    return Rational(-zaehler, nenner);
}

Rational Rational::operator+(const Rational& x){
    return add(x);
}
Rational Rational::operator-(const Rational& x){
    return sub(x);
}
Rational Rational::operator*(const Rational& x){
    return mul(x);
}
Rational Rational::operator/(const Rational& x){
    return div(x);
}
```

```cpp
bool Rational::operator<(const Rational& x) {
    return zaehler * x.nenner
           < nenner * x.zaehler;
}

bool Rational::operator>(const Rational& x) {
    return zaehler * x.nenner
           > nenner * x.zaehler;
}

bool Rational::operator==(const Rational& x) {
    return zaehler * x.nenner
           == nenner * x.zaehler;
}
```

Rationale Zahlen in C++

main.cpp

```
#include <iostream>
#include "rational.h"
using namespace std;

int main(void) {
    Rational a(2), b(1, 3), c;

    c = a + b;
    cout << "a + b = " << c << endl;
    c = a - b;
    cout << "a - b = " << c << endl;
    c = a * b;
    cout << "a * b = " << c << endl;
    c = a / b;
    cout << "a / b = " << c << endl;
```

Rationale Zahlen in C++

```
    if (a < b)
        cout << "a < b" << endl;

    if (a > b)
        cout << "a > b" << endl;

    if (a == b)
        cout << "a == b" << endl;

    cout << "c.zaehler = " << c.getZaehler();
    cout << "c.nenner  = " << c.getNenner();

    return 0;
}
```

Überladen von Operatoren

Der Operator << in der Klasse ostream ist vielfach überladen, um beliebige Datentypen in ein ostream-Objekt schreiben zu können:

```
class ostream {
public:
    ostream& operator<< (bool& wert);
    ostream& operator<< (short& wert);
    ostream& operator<< (unsigned short& wert);
    ostream& operator<< (int& wert);
    ostream& operator<< (unsigned int& wert);
    ostream& operator<< (long& wert);
    ostream& operator<< (unsigned long& wert);
    .....
};
```

Überladen von Operatoren

in C++: `cout` ist ein vordefiniertes Objekt der Klasse `ostream`
in C: `stdout` ist ein vordefinierter Zeiger vom Typ `FILE *`

```
cout << 1;
```

wird vom Compiler übersetzt nach

```
cout.operator<<(1);
```

Alle `<<`-Operatorfunktionen liefern eine Referenz auf das aktuelle Stream-Objekt als Rückgabewert, wodurch eine Verkettung mehrerer Ausgaben möglich wird:

```
cout << 1 << ", " << 2 << endl;
```

Überladen von Operatoren

Der Operator << wird von links nach rechts abgearbeitet:

```
cout << 1 << ", " << 2 << endl;
```

entspricht also

```
(((cout << 1) << ", ") << 2) << endl;
```

und daher

```
cout.operator<<(1).operator<<(", ")
    .operator<<(2).operator<<(endl);
```

Auch die Priorität kann nicht geändert werden:

```
cout << a + b;       // Klammern unnoetig
cout << (a & b);     // Klammern notwendig
```

Überladen von Operatoren

Wenn Sie den Operator `<<` auf eigene Klassen erweitern wollen, müssen Sie die globale Operatorfunktion `operator<<()` überladen:

```
ostream& operator<< (ostream& os,
                      const Rational& r) {
    os << r.getZaehler();
    os << "/";
    os << r.getNenner();
    return os;
}
```

Die Getter-Methoden `getZaehler()` und `getNenner()` sind notwendig, da der Zugriff auf die privaten Variablen `zaehler` und `nenner` nicht möglich ist.

Es sei denn ....

Überladen von Operatoren

.... die Operatorfunktion `operator<<()` wird als Freund der Klasse `Rational` definiert. Freunde dürfen private Variablen sehen!

```
class Rational {
    friend ostream& operator<< (ostream& os,
            const Rational& r) {
        os << r.zaehler << "/" << r.nenner;
        return os;
    }
.....
};
```

Anmerkungen:

- Zur besseren Lesbarkeit stehen `friend`-Deklarationen am Anfang der Klasse.
- Es können ganze Klassen als Freund definiert werden.
- Das Konzept der Freunde gibt es nicht in allen objektorientierten Programmiersprachen.

Überladen von Operatoren

Operatoren können wie Funktionen überladen werden.

Jede Operatoranwendung kann aufgefasst werden als Aufruf einer Operatorfunktion der Form:

```
<Typ> operator<Zeichen>(<Formalparameter>)
Rational operator+(Rational a)
```

Zum Überladen muss die Operatorfunktion entweder

- als Klassenfunktion definiert werden, dann besitzt sie einen impliziten Objektparameter der Form `T &` oder `const T &` oder
- als Funktion mit mindestens einem Parameter vom Typ Klasse, oder Referenz auf Klasse definiert werden.

Überladen von Operatoren

Anmerkungen:

- Operatoren für Standardtypen können nicht überladen werden.
- Operatorfunktionen können explizit aufgerufen werden.
- Priorität und Assoziativität eines Operators wird nicht verändert.
- Operandenzahl eines Operators kann nicht verändert werden.
- Es können keine neuen Operatorzeichen definiert werden.

  Zum Beispiel ist ** für Potenzieren nicht möglich.

Übungen

Überladen Sie den Index-Operator für unsere Liste, sodass anstelle von `val = l.getValueAt(10)` einfacher `val = l[10]` zum Zugriff auf das zehnte Element geschrieben werden kann.

Warum ist beim Einlesen eines Wertes von der Tastatur kein Adress-Operator notwendig, wenn doch das Einlesen mittels `>>` in einen Funktionsaufruf umgewandelt wird?

```
int a;
cin >> a;    // entspricht: cin.operator>>(a);
```

Überladen von Operatoren

Wir hatten bereits festgestellt: Die Programmiererin legt fest, wann zwei Objekte als gleich angesehen werden. In C++ wird dazu der `==`-Operator überladen.

- `Konto`: gleiche Kontonummer und Bankleitzahl
- `Student`: gleiche Hochschule und Matrikelnummer

Wunscheigenschaften an den `==`-Operator:

- *Reflexivität:* Ein Objekt ist gleich zu sich selbst. Es sollte also immer `obj == obj` gelten.
- *Symmetrie:* Wenn `obj1 == obj2` gilt, dann sollte natürlich auch `obj2 == obj1` gelten.
- *Transitivität:* Gleichheit kann verkettet werden und gilt für mehrere Objekte: Wenn `ob1 == obj2` und `obj2 == obj3` gilt, dann sollte auch `obj1 == obj3` gelten.

Überladen von Operatoren

Frage: Gibt es Probleme, wenn wir folgendes schreiben?

```
#include <iostream>
#include "rational.h"
using namespace std;

int main(void) {
    int x = 5;
    Rational b(1, 3), c;

    c = x + b;
    cout << "a + b = " << c << endl;

    c = b - x;
    cout << "a - b = " << c << endl;
}
```

Überladen von Operatoren

Frage: Was wäre zu tun, damit eine solche Schreibweise auch für Werte vom Typ `double` funktioniert?

Frage: Wie wandelt man einen `double`-Wert wie 2.0815 in einen Bruch um?

Übung: Formulieren Sie einen Konstruktor der Klasse `Rational`, der einen Wert vom Typ `double` als Bruch darstellt.

Frage: Warum multiplizieren wir Zähler und Nenner mit 2, anstatt mit 10?

Übung: Formulieren Sie einen Konstruktor der Klasse `Rational`, der eine Zahl, gegeben als `string`, als Bruch darstellt.

Übersicht

Softwaretest

- *Motivation*
- Software-Entwicklungsprozess
- Komponententest

Software-Fehler haben Konsequenzen

- 1979: erste Venussonde flog am Ziel vorbei

  Grund: im Programm Punkt mit Komma vertauscht

  Kosten: mehrere hundert Millionen Dollar
- 1984: Überschwemmung in Frankreich

  Grund: Überlaufgefahr nicht erkannt

  Kosten: mehrere Tote
- 1985-87: computergesteuertes Bestrahlungsgerät Therac-25 zur Behandlung von Tumoren ist fehlerhaft

  Anmerkungen: Programm in PDP-11 Assembler war unzureichend dokumentiert; es gab keine Hinweise darauf, dass es jemals getestet wurde

  Kosten: mehrere Tote; mehrere Schwerverletzte mit Lähmungen, Verbrennungen und Verstrahlungen

Software-Fehler haben Konsequenzen

- 1991: Patriot-Raketen-Fehler

  Grund: ungenaue Berechnung der Zeit seit Systemstart wegen Rundungsfehler

  Kosten: 28 Tote, weil die zu treffende Rakete verfehlt wurde und diese in eine Kaserne einschlug
- 1993: Gepäcksortierung am Flughafen Denver

  Grund: Sortieranweisungen kamen zu spät wegen Überlastung des Netzwerks

  Kosten: 3 Milliarden Dollar, Gepäcksortierung musste zunächst per Hand durchgeführt werden; Eröffnung des Flughafens musste mehrfach um mehrere Monate verlegt werden
- 1994: Pentium-Bug: Fehler bei Division

  Grund: Tabelle mit Schätzwerten für nächste Stelle war zu klein

  Kosten: 400 Millionen Dollar

Software-Fehler haben Konsequenzen

- 1996: Ariane 5 verglüht

  Grund: Software-Problem im Trägheitsnavigationssystem

  Kosten: ca. 1 Milliarde Euro
- Bank of New York buchte 32 Milliarden Dollar zuviel

  Grund: Überlauf eines 16-Bit Zählers

  Kosten: 5 Millionen Dollar (Zinsverlust)

Literatur

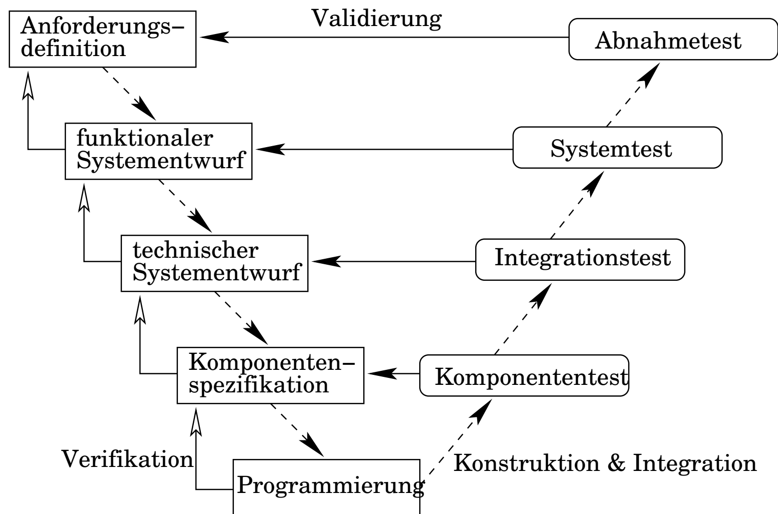
- Andreas Spillner, Tilo Linz:
  Basiswissen Softwaretest.
  dpunkt.verlag
- Harry M. Sneed, Mario Winter:
  Testen objektorientierter Software.
  Hanser Verlag.
- Wolfgang Ehrenberger:
  Software-Verifikation.
  Hanser Verlag.
- Andreas Zeller, Jens Krinke:
  Programmierwerkzeuge.
  dpunkt.verlag.

Übersicht

Softwaretest

- Motivation
- *Software-Entwicklungsprozess*
- Komponententest

Allgemeines V-Modell



Allgemeines V-Modell

Anforderungsdefinition Wünsche und Anforderungen des Auftraggebers sammeln, spezifizieren und verabschieden.

> Konfuzius: Wenn die Sprache nicht stimmt, ist das was gesagt wird nicht das was gemeint ist. So kommen keine guten Werke zustande. Also dulde man keine Willkür in den Worten.

Funktionaler Systementwurf Anforderungen auf Funktionen und Dialogabläufe des neuen Systems abbilden.

Technischer Systementwurf Technische Realisierung des Systems entwerfen (Definition der Schnittstellen, Zerlegung in Teilsysteme)

Komponentenspezifikation Für jedes Teilsystem werden Aufgabe, Verhalten, innerer Aufbau und Schnittstelle zu anderen Teilsystemen spezifiziert.

Programmierung Implementierung jedes Bausteins

Allgemeines V-Modell

Komponententest Erfüllt jeder Baustein für sich die Vorgaben seiner Spezifikation?

Integrationstest Spielen Gruppen von Komponenten so zusammen, wie im technischen Systementwurf vorgesehen?

Systemtest Erfüllt das System als Ganzes die spezifizierten Anforderungen?

Abnahmetest Weist das System aus Kundensicht die vertraglich vereinbarten Leistungsmerkmale auf?

In jeder Phase der Software-Entwicklung sind Qualitätsprüfungen durchzuführen!

Agile Methoden*

Softwaretechnik muss effizienter werden:

- Software mit immer höherer Qualität
- ist mit immer weniger personellen Ressourcen
- in immer kürzeren Zeiten zu erstellen.

Der *Rational Unified Process* oder das *V-Modell*

- sind mit Tätigkeiten überfrachtet, die für das Endergebnis nicht essentiell sind und
- sind zu schwerfällig, um auf die sich schnell ändernde Umgebung (Technik, Anforderungen, Konkurrenz) eines Software-Entwicklungsprojekts reagieren zu können.

Agile Softwareentwicklung ist Mainstream. iX 3/2010

---

* aus Bernhard Rumpe: Modellierung mit UML. Springer Verlag.

Agile Methoden

Projekte scheitern wegen schlechten Projektmanagements:

- Es wird nicht adäquat kommuniziert,
- zu viel, zu wenig oder das Falsche dokumentiert,
- Risiken nicht rechtzeitig entgegengesteuert oder
- zu spät Rückkopplung von den Anwendern eingefordert.

Die Produktivität wird bei den neuen, agilen Methoden durch das Weglassen von Arbeiten und Hierarchien gesteigert.

Derek Coleman:

- There is a move away from software processes that are hierarchical and management driven.
- The trend is to cooperative styles of development where management dictate is replaced by ethical considerations of community membership.

Es gibt nicht *den* Prozess für Software-Entwicklung. Stattdessen: Sammlung von

Konzepten, Best Practices und Werkzeugen

um projektspezifischen Erfordernissen in einem individuellen Prozess Rechnung zu tragen.

Agile Methoden

Die Softwareentwicklung hat sich durch die Verfügbarkeit verbesserter Programmiersprachen, Compiler und Entwicklungswerkzeugen geändert: Es ist heute bspw. genauso effizient eine GUI direkt zu programmieren, wie sie zu spezifizieren.

→ Die Spezifikation kann durch einen für den Anwender ausprobierbaren Prototypen ersetzt werden.

Weniger Projektbeteiligte bedeuten weniger Projektmanagement.

→ Mündige und motivierte Projektbeteiligte handeln von sich aus verantwortungsvoll und couragiert, vorausgesetzt das Projektumfeld stimmt.

Siehe hierzu auch „Das Wettrudern“ unter `http://www.scheissprojekt.de/wettrudern.html`.

Extreme Programming (XP)

Extreme Programming fokussiert auf das ultimative Ziel der Softwareentwicklung: den Code.

- Alles an zusätzlicher Dokumentation wird als vermeidbarer Ballast betrachtet.
- Eine Dokumentation ist aufwändig zu erstellen und oft sehr viel fehlerhafter als der Code, weil sie nicht ausreichend automatisiert analysierbar und testbar ist.
- Dokumentation reduziert Flexibilität bei Weiterentwicklung und Anpassung des Systems als schnelle Reaktion auf neue oder veränderte Anforderungen des Kunden.

→ Keine Dokumentation erstellen, aber Wert legen auf
  - gute Kommentierung des Quellcodes (`javadoc`, `doxygen`)
  - und eine umfangreiche Testsammlung.

Test-First-Ansatz bei XP

Vor der Entwicklung der eigentlichen Funktionalität sind Testfälle zu überlegen:

- Der Testentwurf ist de facto die Spezifikation.
- Fertig gestellter Code kann sofort überprüft werden.
- Der logische Entwurf wird von der Implementierung getrennt, denn: XP ist kein Hacking.

Egal ob V-Modell, agile Methoden oder anderer SWE-Prozess: Tests sind wichtig und immer durchzuführen!

Tests müssen voll automatisiert ablaufen und ihre Ergebnisse selbst überprüfen!

XP nicht ohne Refactoring

In der Praxis: Inkrementelle Verbesserungen und Modifikation von Programmcode.

- Um Komplexität beherrschbar zu machen.
  „Wir fangen erstmal klein an."
- Um auf geänderte Kundenwünsche zu reagieren.
  „Wir wissen auch noch nicht so genau, was wir wollen."

Bereits geschriebener Code muss evtl. umgeschrieben werden, um Änderungen oder Erweiterungen einfacher durchführen zu können.

Siehe hierzu auch „Softwareentwicklung als Hausbau" unter `http://www.scheissprojekt.de/hausbau.html`.

XP nicht ohne Refactoring

Refactoring:

- Semantikerhaltende Transformation eines bestehenden Programms.
- Dient nicht zur Erweiterung der Funktionalität, sondern zur Verbesserung der Qualität des Entwurfs unter Beibehaltung der Funktionalität.
- Beispiel: Anstelle einer Liste soll in einem Programm jetzt ein Suchbaum verwendet werden, um die Applikation zu beschleunigen.

→ Automatische Tests sind wichtig: Tut das Programm noch das Gleiche wie vor der Änderung?

XP nicht ohne Refactoring

Weiteres Beispiel:

- Um die neue Suchfunktion besser unterstützen zu können, wird eine Adresse nicht mehr als `string` abgelegt, sondern als Struktur mit den Elementen Straße, Hausnummer, Land, Postleitzahl und Ort.

→ Tests stellen sicher, dass die Software nach der Änderung immer noch funktioniert!

Entwickler lieben es, schöne Architekturen zu erstellen, die man von allen Seiten verändern kann. In den meisten Fällen ist das unnötig: Es erzeugt Komplexität, es kostet Zeit und damit dem Kunden Geld, und es ist oft schwer nachzuvollziehen.

Halten Sie Ihren Entwurf so einfach wie möglich!

Übersicht

Softwaretest

- Motivation
- Software-Entwicklungsprozess
- *Komponententest*

Wege zur Software-Qualität

Edsger W. Dijkstra:

- The art of programming is the art of organizing complexity.
- Testen kann die Anwesenheit, aber nicht die Abwesenheit von Fehlern zeigen.

Auswahl der Testfälle:

- aus Spezifikation der Testobjekte → Black-Box
- auf Basis des Programmtextes → White-Box

- Die unmittelbar nach der Programmierphase erstellten Software-Bausteine, bei uns also die Klassen, werden erstmalig systematisch getestet.
- Wird ein Software-Baustein isoliert von anderen Komponenten des Systems geprüft, lässt sich eine Fehlerursache eindeutig der Komponente zuordnen.
- Ein *Testtreiber* ist ein Programm, dass Schnittstellenaufrufe absetzt und die Reaktion des Testobjekts entgegennimmt.

  Damit wollen wir sicherstellen, dass das Testobjekt die geforderte Funktionalität, also das Ein-/Ausgabeverhalten, korrekt und vollständig realisiert.

Testtreiber für die Liste

```
#include <iostream>
#include .....
using namespace std;
const bool OK = true;

bool test01(void) { // Black-Box: append() ok?
    Liste<int> l(4);

    l.append(1);
    if (l.getValueAt(0) != 1)
        return !OK;
    try {         // Kein weiterer Wert vorhanden?
        l.getValueAt(1);
    } catch (const char *e) {
        return OK;
    }
    return !OK;
}
```

Testtreiber für die Liste

```
// Black-Box Test: Ist erase() ok?
bool test02(void) {
    Liste<int> l(4);

    l.append(1);
    try {
        l.erase(1);
    } catch (Exception e) {
        return !OK;
    }
    try {         // Wert wirklich entfernt?
        l.getValueAt(0);
    } catch (const char *e) {
        return OK;
    }
    return !OK;
}
```

Testtreiber für die Liste

```
// White-Box Test: Ist increase() ok?
bool test03(void) {
    Liste<int> l(4);

    for (int i = 0; i < 50; i++)
        l.append(i);

    for (int i = 0; i < 50; i++)
        if (l.getValueAt(i) != i)
            return !OK;
    return OK;
}

....
```

Testtreiber für die Liste

```
int main(void) {
    bool ok = true;

    if (ok) {
        if (true == (ok = test01()))
            cout << "test01 passed\n";
        else cout << "!!! test01 failed !!!\n";
    }
    if (ok) {
        if (true == (ok = test02()))
            cout << "test02 passed\n";
        else cout << "!!! test02 failed !!!\n";
    }
    ....
    if (ok)
        cout << "\nTest passed\n";
    else cout << "\n!!! Test failed !!!\n";
}
```

Testüberdeckung

Wieviele Tests sind durchzuführen?

- C_0-Test: Anweisungsüberdeckung

$$\frac{\text{Anzahl durchlaufene Anweisungen}}{\text{Anzahl Anweisungen}}$$

- C_1-Test: Zweigüberdeckung

$$\frac{\text{Anzahl durchlaufene Zweige}}{\text{Anzahl Zweige}}$$

Anmerkungen:

- $C_0 = 100\% \not\Rightarrow C_1 = 100\%$ (`if`-Anweisung ohne `else`-Zweig)
- In der Regel sollte $C_1 = 100\%$ erreicht werden.

Testüberdeckung

Mit dem Gnu COVerage tool `gcov` kann man die Testüberdeckung ermitteln und anzeigen lassen:

- Dazu wird das Programm kompiliert mittels
  `g++ -g -fprofile-arcs -ftest-coverage <prog.cpp>`
- anschließend wird der Testtreiber ausgeführt
- schließlich wird mittels `gcov -b <prog>` die Testüberdeckung angezeigt

Es gibt weitere Kriterien für Testüberdeckung:

- Pfadüberdeckung
- Bedingungsüberdeckung

Testüberdeckung: Beispiel

Zähle in einer Textdatei die Anzahl der Zeilen, Wörter und Zeichen. Ein Wort ist definiert als eine Zeichenfolge, die nicht durch ein Blank getrennt wird. Die zu untersuchende Datei wird als Parameter an das Programm übergeben.

```
#include <stdio.h>
#include <ctype.h>
#define N 4000

int main(int argc, char *argv[]) {
   int i;
   long linec, wordc, charc;
   char l[N];
   FILE *file;
```

Testüberdeckung: Beispiel

```
if (argc != 2) {
   printf("usage: %s filename\n", argv[0]);
   return 1;
}

linec = 0;
wordc = 0;
charc = 0;

file = fopen(argv[1], "r");
if (file == NULL) {
   perror(argv[1]);
   return 2;
}
```

Testüberdeckung: Beispiel

```
fgets(l, N, file);
while (!feof(file)) {
   i = 0;
   linec += 1;

   while (l[i] != '\n') {
      for (; isblank(l[i]) &&
             l[i] != '\n'; i++)
         charc += 1;

      for (; !isblank(l[i]) &&
             l[i] != '\n'; i++)
         charc += 1;
      wordc += 1;
   }
   fgets(l, N, file);
}
```

Testüberdeckung: Beispiel

```
    fclose(file);
    printf("%ld %ld %ld\n", linec, wordc, charc);
    return 0;
}
```

- Übersetzen mit:
  ```
  gcc -g -fprofile-arcs -ftest-coverage \
        wordcount.c -o wordcount
  ```
- Programmausführung mit: `./wordcount _wc.txt`
- `gcov -b wordcount` liefert:

  ```
  Lines executed:84.00% of 33
  Branches executed:100.00% of 14
  Taken at least once:81.25% of 14
  Calls executed:80.00% of 12
  ```

- In der Datei `wordcount.c.gcov` finden wir weitere Informationen, unter anderem den Hinweis, dass die Zeile

  ```
  perror(argv[1]);
  ```

  nicht durchlaufen wurde. Daher starten wir das Programm erneut, aber jetzt geben wir eine Datei an, die nicht existiert, und erhalten die folgende Ausgabe von `gcov`:

  ```
  Lines executed:92.00% of 33
  Branches executed:100.00% of 14
  Taken at least once:87.50% of 14
  Calls executed:90.00% of 12
  ```

Testüberdeckung: Beispiel

- Diesmal finden wir in der Datei `wordcount.c.gcov` den Hinweis, dass die Zeile

  ```
  printf("usage: %s filename\n", argv[0]);
  ```

  nicht durchlaufen wurde. Daher starten wir das Programm erneut, geben aber keinen Dateinamen an, und erhalten diesmal die folgende Ausgabe von `gcov`:

  ```
  Lines executed: 100.00% of 33
  Branches executed: 100.00% of 14
  Taken at least once: 92.86% of 14
  Calls executed: 100.00% of 12
  ```

Das sieht doch schon ganz gut aus.

Testüberdeckung: Beispiel

Überdeckung hängt (leider) von Compiler-Optimierung ab!

- Compile ohne Optimierung, dann obige Testfälle ausführen liefert

```
      for (; isblank(l[i]) && l[i] != '\n'; i++)
branch  0 taken 67%
branch  1 taken 100%
branch  2 never executed
branch  3 taken 100%
```

- Compile mit Optimierung `O3`, dann obige Testfälle ausführen liefert

```
      for (; isblank(l[i]) && l[i] != '\n'; i++)
call    0 returns 100%
branch  1 taken 100%
branch  2 taken 67%
branch  3 taken 100%
```

Übersicht

Objektorientierung

- *Motivation*
- Funktionalität einer Klasse erweitern
- Polymorphie
- Abstrakte Klassen
- Klassen im Zusammenspiel
- Modellierung
- Sequenz-, Aktivitäts- und Zustandsdiagramme
- Standard Template Library

Warum sprechen wir von Objektorientierung und nicht von Klassenorientierung? Wir definieren in C++ doch Klassen.

- Wir Menschen nehmen einzelne Objekte war und
- finden gemeinsame Verhaltensweisen und Eigenschaften.

→ Wir klassifizieren und abstrahieren!

Objektorientierung

Beispiel: Bello und Rex haben vier Beine und machen Wau-Wau.

- Wir klassifizieren: Beides sind Hunde. Wir fassen Gegenstände allgemein auf und finden einen Begriff dafür. Und wir grenzen ab: Hunde unterscheiden sich von Katzen.
- Wir abstrahieren: Innere Abläufe wie Atmung, Verdauung oder Durchblutung spielen für uns keine Rolle. Wir vereinfachen die komplexe Welt, um sie leichter verstehen zu können.

Objektorientierung

Objekte, die ein gemeinsames Verhalten aufweisen, werden in Klassen allgemein beschrieben.

- im Vordergrund: Verhalten der Objekte nach außen
- im Hintergrund: Daten, die intern notwendig sind, um das Verhalten nach außen zu erfüllen

$\rightarrow$ *konzentrieren auf das Verhalten (Schnittstelle)*

Klassen werden eingesetzt,

- um Objekte mit gleichem Sachverhalt allgemein gültig zu beschreiben und
- um einen komplexen Sachverhalt zu kapseln.

Eine Klasse ist ein Bauplan für gleichartige Objekte.

Betrachten wir als Beispiel unsere Liste:

- Bei einem Array
  - findet keine Bereichsüberprüfung statt und
  - die Größe wird nicht automatisch angepasst.
- Kapsele den Datentyp Array als Klasse `Liste` und schütze den Zugriff auf die Daten.
- Nach ausgiebigen Tests können ohne Bedenken viele Objekte der Klasse angelegt werden.

Vorteile dieses Vorgehens:

- Die Komplexität wird reduziert.
- Der Programmierer kann sich auf die Lösung seines eigentlichen Problems konzentrieren.
- Eine Wiederverwendung von Code wird möglich.

Beispiele aus anderen Bereichen:

- Ein Fernseher hat eine einfache Benutzer-Schnittstelle.
  - → Die komplexe Technik im Innern muss vom Nutzer nicht verstanden werden.
- Im Computer sind einzelne Komponenten durch ein einheitliches Bus-System verbunden.
  - → Die einzelnen Komponenten können unabhängig von den anderen entworfen und betrieben werden, solange die Bus-Spezifikation eingehalten wird.

→ So sollte auch Software entworfen sein!

Abstraktionsmittel

- Oberbegriff (*ist-ein*-Beziehung)

  Fasst mehrere Arten oder Varianten von Objekten unter einem Begriff zusammen.

  Man sagt: *Da hinten fahren 3 Autos.*
  nicht: *ein Opel, ein Ford und ein VW*

- Teile/Ganzes (*hat-eine*-Beziehung)

  Fasst mehrere Einzelteile zu einem Objekt zusammen.

  Man sagt: *Da hinten fährt ein Auto.*
  nicht: *ein Motor mit Karosserie, Lenkrad und 4 Rädern*

Ursprung der Objektorientierung

- Objektorientierte Sprachen wie Simula und Smalltalk gibt es schon seit Ende der 1960er Jahre.
- OO-Sprachen fristeten aber zunächst ein Nischendasein, die prozeduralen Programmiersprachen wie Algol-68, C und Fortran setzten sich durch.
- Die Abstraktionsmöglichkeiten der prozeduralen Sprachen sind sehr beschränkt, reichen aber für kleine Programme aus.
- Computer wurden immer leistungsfähiger, was zu steigenden Anforderungen an Software führte. Als Folge davon wurden die Programme immer komplizierter.
- Um diese Komplexität noch beherrschen zu können, müssen die Programmiersprachen andere Konzepte bereitstellen.
- Edsger W. Dijkstra: „The art of programming is the art of organizing complexity."

Ursprung der Objektorientierung

Edsger W. Dijkstra in seiner Dankesrede zum Turing Award:

> *[The major cause of the software crisis is] that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*

Im Verlauf der Software-Krise erinnerte man sich an die Konzepte der objektorientierten Programmierung und hoffte, damit den Weg aus der Krise zu finden. Heute haben wir aspektorientierte Programmierung, agile Methoden und modellgetriebene Software-Entwicklung und trotzdem werden Budgets überzogen, Termine nicht eingehalten und Software ist immer noch fehlerhaft.

Ursprung der Objektorientierung

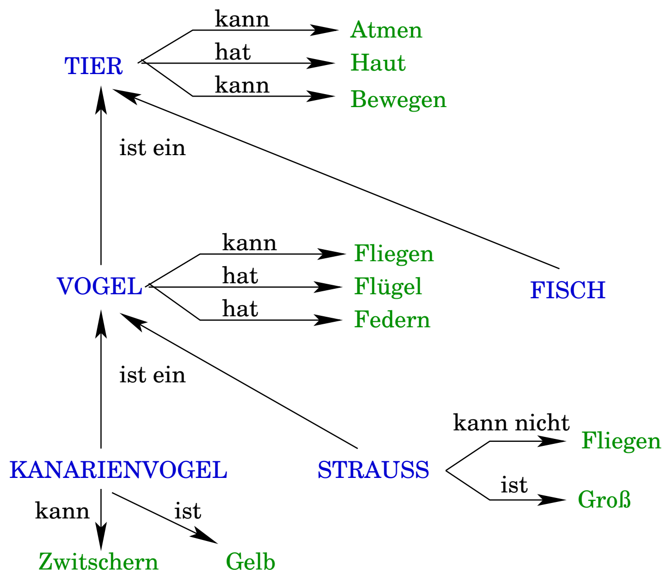
Auch das Gebiet der künstlichen Intelligenz bzw. die Entwicklung wissensbasierter Systeme forcierte die Entwicklung der objektorientierten Sprachen.

Wie wird Wissen im menschlichen Gehirn gespeichert? Können wir in ähnlicher Weise Wissen im Rechner speichern?

Semantische Netze stellen Wissen in einem Graphen dar:

- Knoten entsprechen Fakten bzw. Konzepten und
- Kanten entsprechen Relationen bzw. Assoziationen zwischen Konzepten.
- Sowohl Knoten als auch Kanten sind in der Regel mit Beschriftungen versehen.

Ursprung der Objektorientierung



Semantische Netze entsprechen vielleicht der Speicherung von Informationen beim Menschen. (Harmon, King, 1985)

Satz	Antwortzeit (s)
Ein Kanarienvogel ist ein Kanarienvogel.	1.0
Ein Kanarienvogel ist ein Vogel.	1.18
Ein Kanarienvogel ist ein Tier.	1.26
Ein Kanarienvogel kann zwitschern.	1.32
Ein Kanarienvogel kann fliegen.	1.38
Ein Kanarienvogel hat Haut.	1.48

Am schnellsten konnten die Informationen abgerufen werden, die spezifisch für den jeweiligen Vogel sind.

Auch die Verarbeitung von Ausnahmen scheint auf der spezifischen Ebene zu erfolgen. Auf die Frage, ob Strauße fliegen können, wurde schneller geantwortet als auf die Frage, ob Strauße atmen können.

Übersicht

Objektorientierung

- Motivation
- *Funktionalität einer Klasse erweitern*
- Polymorphie
- Abstrakte Klassen
- Klassen im Zusammenspiel
- Modellierung
- Sequenz-, Aktivitäts- und Zustandsdiagramme
- Standard Template Library

Motivation

Wir wollen unsere Liste um

- eine Suchfunktion `bool contains(T val)` und
- eine Abfrage der Größe `int size()` erweitern und
- den Index-Operator überladen.

Frage: Spricht irgendetwas dagegen, diese Erweiterungen einfach in den bestehenden Code zu schreiben?

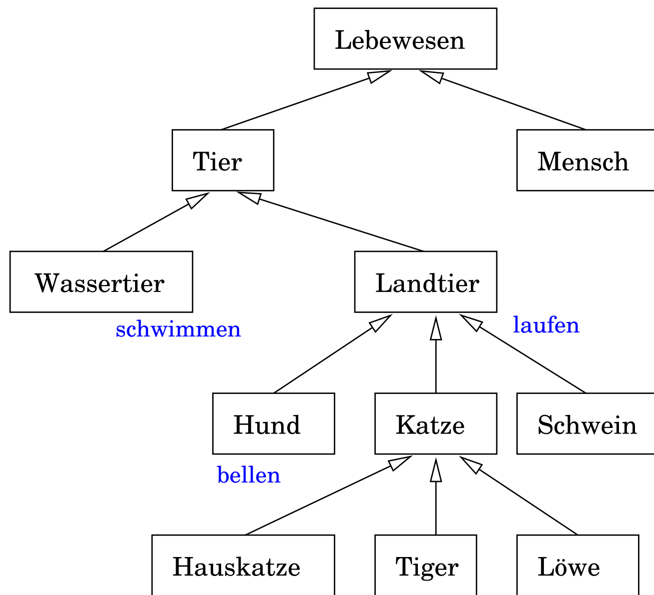
Motivation

Eigentlich nicht, aber: Erweiterungen in den bestehenden Code zu schreiben ist nicht immer eine gute Idee.

- Der Source-Code der Klasse `Liste` wird immer umfangreicher und ist nur noch schwer zu verstehen/warten.
- Beim Ändern/Erweitern von bestehendem Code werden evtl. Fehler eingebaut und alte Software läuft anschließend nicht mehr.
- Der Source-Code steht in der Praxis nicht unbedingt zur Verfügung.

Wir brauchen einen Mechanismus, der uns die alte Funktionalität bereit stellt, aber dennoch Erweiterungen zulässt! Vielleicht entkommen wir so der Softwarekrise.

Hierarchische Anordnung von Klassen



Hierarchische Anordnung von Klassen

In einer solchen Hierarchie gilt:

- Alle Lebewesen einer Klasse verfügen über identische Eigenschaften.
- Die Klassen in einer tieferen Hierarchiestufe sind eine Spezialisierung der direkt übergeordneten Klasse.
- Die von einer Klasse abgeleiteten Unterklassen verfügen immer über alle Eigenschaften der Oberklasse.
- Abgeleitete Unterklassen können weitere Eigenschaften hinzufügen.

Eine solche Hierarchie ist analog zu einem evolutionärem Stammbaum, daher spricht man auch von der Vererbung von Eigenschaften.

Hierarchische Anordnung von Klassen

Obiges Konzept ist auch anwendbar auf Software-Entwicklung!

Begriffe:

- Die Klasse, die Eigenschaften weitervererbt, wird Oberklasse, Basisklasse oder Superklasse genannt.
- Klassen, die etwas erben, heißen Unterklassen, Subklassen oder abgeleitete Klassen.
- Die Oberklasse ist eine Generalisierung der Unterklassen.
- Eine Unterklasse ist eine Spezialisierung der Oberklasse.

Eine abgeleitete Klasse kann wiederum als Basisklasse für weitere Unterklassen dienen.

Typische Anwendungsfälle für Vererbung

- Es wird eine Klasse benötigt, die eine Spezialisierung oder Erweiterung einer bereits vorhandenen Klasse ist.

  hier: Wir wollen der `Liste` die Funktion `size` hinzufügen, ohne den bestehenden Code zu ändern.

- Eine Klasse vereint die Eigenschaften mehrerer vorhandener Klassen.

- Man entwickelt mehrere Klassen parallel, und man erkennt erst nachträglich, dass viel Gemeinsamkeiten vorliegen, für die man eine Basisklasse einführt.

  Wie kann das trotz guter Projektplanung passieren?

  - Software-Entwicklung häufig auf Zuruf und unter Zeitdruck.
  - Klassen entstehen in verschiedenen Projekten.

  → Refactoring

Erweitern der Funktionalität

Unsere Liste soll zusätzlich

- eine Suchfunktion `bool contains(T val)` bereit stellen,
- die Anzahl der gespeicherten Elemente ausgeben `int size()`
- und einen Zugriff auf die Elemente über den Index-Operator ermöglichen.

Und wenn wir schon mal dabei sind: Die Liste soll zusätzlich

- das Minimum ausgeben: `T minimum()`
- den Median ausgeben: `T median()`
- die Werte sortieren: `void sort()`

extliste.h

```
#include "liste.h"
#include "exception.h"

template <typename T>
class ExtListe: public Liste<T> {    // Schutztyp
    bool _sorted;            // bzw. Ableitungsmodus
    bool isEmpty() {
        return _last == 0;
    }
public:
    ExtListe(int size = 8);
    int size();
    bool contains(T val);
    T operator[](int pos);
    T minimum();
    T median();
    void sort();
};
```

Erweitern der Funktionalität

```
template <typename T>
ExtListe<T>::ExtListe(int size): Liste<T>(size){
     // Konstruktor der Basisklasse wird in der
     // Initialisiererliste aufgerufen, nicht im
     // Funktionsrumpf

    _sorted = false;
}

template <typename T>
bool ExtListe<T>::contains(T val) {
    for (int i = 0; i < _last; i++)
        // implizite Annahme: Templatetyp muss
        // den Vergleichsoperator == ueberladen
        if (_values[i] == val)
            return true;
    return false;
}
```

Erweitern der Funktionalität

```
template <typename T>
int ExtListe<T>::size(void) {
    return _last;
}

template <typename T>
T ExtListe<T>::operator[](int pos) {
    return getValueAt(pos);
}
```

Erweitern der Funktionalität

```
template <typename T>
T ExtListe<T>::minimum(void) {
    if (isEmpty())
        throw Exception("empty list");

    if (!_sorted)
        sort();
    return _values[0];
}

template <typename T>
T ExtListe<T>::median(void) {
    // ..... analog zu minimum()
    return _values[_last / 2];
}
```

```
template <typename T>
void ExtListe<T>::sort(void) {
    for (int i = 0; i < _last; i++) {
        for (int j = i + 1; j < _last; j++) {
            if (_values[i] > _values[j]) {
                T tmp = _values[i];

                _values[i] = _values[j];
                _values[j] = tmp;
            }
        }
    }
    _sorted = true;
}
```

hier: Naives Sortierverfahren! In der Vorlesung *Algorithmen und Datenstrukturen* lernen Sie bessere Sortierverfahren kennen.

Erweitern der Funktionalität

Implizite Annahmen über den Templatetyp:

- Die Vergleichsoperatoren `>` und `==` sowie
- der Zuweisungsoperator `=` müssen überladen sein!

Alte Klausurfragen:

- Wofür wird ein Zuweisungsoperator benötigt?
- Was ist der wesentliche Unterschied zwischen dem Zuweisungsoperator und dem Kopierkonstruktor?

Erweitern der Funktionalität

Anmerkung: Bei einigen neueren C++-Compilern wird der obige Code so nicht übersetzt.

- Die Namenssuche erfolgt nicht automatisch in der Basisklasse, wenn Templates ins Spiel kommen.
- Alle Zugriffe auf Variablen und Methoden der Oberklasse müssen explizit mit `Liste<T>::` erfolgen.

```
template <typename T>
int ExtListe<T>::size(void) {
    return Liste<T>::_last;
}

template <typename T>
T ExtListe<T>::operator[](int pos) {
    return Liste<T>::getValueAt(pos);
}
```

Erweitern der Funktionalität

Unsere Implementierung ist falsch:

- `_values` und `_last` sind in der Basisklasse als `private` deklariert und daher in der abgeleiteten Klasse nicht verfügbar.
- Die Logik ist nicht korrekt: Wir müssen die Methode `append` überschreiben und `_sorted` auf `false` setzen.

weitere Anmerkungen:

- Der Konstruktor wird nicht geerbt.
- Der Destruktor wird nicht geerbt.

Eigenschaften abgeleiteter Klassen

- Alle `public`-Elemente der Basisklasse sind verfügbar.

  Übrigens: Die Menge der `public`-Elemente einer Klasse nennt man die Schnittstelle der Klasse.

- Alle `private`-Elemente bleiben auf jeden Fall nach außen verborgen, also auch einer abgeleiteten Klasse verborgen.

- Wollen wir Attribute oder Methoden zwar nach außen verbergen, aber abgeleiteten Klassen zur Verfügung stellen, müssen wir die Elemente mittels `protected` kennzeichnen.

Eigenschaften abgeleiteter Klassen

- Weitere Elemente können der abgeleiteten Klasse hinzugefügt werden.
- Elemente der Basisklasse können überschrieben (also ersetzt), aber nicht gestrichen werden.
  (Das ist so nicht ganz richtig, siehe dazu den Abschnitt Schutztyp.)

→ Die abgeleitete Klasse leistet alles, was die Basisklasse kann und bietet darüberhinaus noch zusätzliche Funktionalitäten an.

Zugriffschutz `protected`

`protected`

- erlaubt den Zugriff auf Variablen und Methoden aus abgeleiteten Klassen, aber nicht von anderen Klassen.
- entspricht also
  - `public` für abgeleitete Klassen und
  - `private` für alle anderen Klassen.

→ Beim Entwurf der Klasse beachten, ob später Erweiterungen möglich sind.

Zugriffschutz

Zugriff möglich durch ...	`private`	`protected`	`public`
eine eigene Memberfunktion	x	x	x
eine beliebige Funktion			x
eine befreundete Funktion	x	x	x
die Memberfunktion einer beliebigen Klasse			x
die Memberfunktion einer direkt abgeleiteten Klasse		x	x
die Memberfunktion einer befreundeten Klasse	x	x	x

Schutztyp – Zugriffsrechte bei Ableitung

- Ableitung `public`: Die Zugriffsrechte der Basisklasse bleiben erhalten.

```
class ExtListe: public Liste
```

- Ableitung `protected`: Die `public`-Elemente der Basisklasse werden zu `protected`-Elementen der abgeleiteten Klasse.

```
class ExtListe: protected Liste
```

- Ableitung `private` (Standard, falls die Angabe fehlt): Die `public`- und die `protected`-Elemente der Basisklasse werden zu `private`-Elementen der abgeleiteten Klasse.

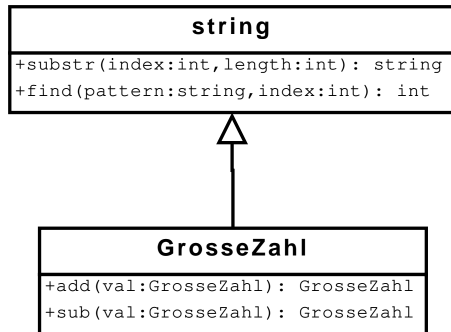
```
class ExtListe: private Liste
```

Wofür braucht man das?

Schutztyp

Anmerkung: Im Normalfall ist nur die `public`-Ableitung sinnvoll!
Aber die `private`-Ableitung wurde als Standard festgelegt.

Es sind `private`-Ableitungen vorstellbar:



- Mittels `GrosseZahl` sind beliebig lange Zahlen darstellbar.
- Durch die `private`-Ableitung verhindern wir, dass der Benutzer mit `string`-Methoden Unsinn machen kann.
- Schlechtes Design?

Schutztyp

Vererbung wird oft als ist-ein-Beziehung charakterisiert:

- Ein Tier ist ein Lebewesen.
- Ein Landtier ist ein Tier.
- Eine Katze ist ein Landtier.
- Ein Tiger ist eine Katze.

Diese Logik gilt aber nur für `public`-Ableitungen. Nur dann verfügen die Unterklassen über die Eigenschaften der Oberklasse.

Eine `GrosseZahl` ist eine Zahl, aber kein `string`, denn die `string`-Funktionalität steht nach der Ableitung nicht mehr zur Verfügung.

Konstruktoren und Destruktoren

Konstruktoren und Destruktoren werden nicht vererbt!

- Der Konstruktor der abgeleiteten Klasse ruft automatisch als erstes den Konstruktor der Basisklasse auf, entweder
  - explizit durch Angabe des Konstruktors in der Initialisiererliste
  - oder implizit durch Aufruf des Standard-Konstruktors.
- Erst wenn die Abarbeitung des Basis-Konstruktors beendet ist, wird der Rumpf des Konstruktors der abgeleiteten Klasse abgearbeitet.

Das hat zur Folge:

- Die Datenelemente der Basisklasse wurden bereits initialisiert, wenn die Abarbeitung der Befehle des Konstruktors der abgeleiteten Klasse beginnt.
- Die Methoden der Basisklasse können im Konstruktor der abgeleiteten Klasse aufgerufen werden.

- Der Aufruf der Destruktoren erfolgt in umgekehrter Reihenfolge: zuerst von der abgeleiteten Klasse, dann von der Basisklasse.
- Werden keine besonderen Maßnahmen getroffen, wird der Standard-Konstruktor der Basisklasse aufgerufen, d.h. dieser muss auch vorhanden sein!

  Soll ein anderer, selbstdefinierter Konstruktor der Basisklasse aufgerufen werden, so erfolgt dies mittels Initialisierungslisten.
- Argumente für den Konstruktor der Basisklasse werden in der Definition des Konstruktors der abgeleiteten Klassen angegeben.

Konstruktoren und Destruktoren

```
class Date {
    int day, month, year;
....
    Date(int d, int m, int y) { .... }
    static Date getCurrentDate();
....
};

class DateTime : public Date {
    int hour, minute, seconds;
....
    DateTime(int d, int m, int y,
             int ho, int mi, int se)
             : Date(d, m, y) { .... }
    static DateTime getCurrentDateTime();
....
};
```

Konstruktoren und Destruktoren

Ausgabe? Was fällt bei der Zuweisung an `_alter` auf?

```
#include <iostream>
using namespace std;

class Vater {
protected:
    int _alter;

public:
    Vater() {
        cout << "Standard-Konstruktor Vater\n";
        _alter = 35;
    }
    ~Vater() {
        cout << "Destruktor Vater" << endl;
    }
};
```

Konstruktoren und Destruktoren

```
class Sohn: public Vater {
public:
    Sohn(int alter) {
        cout << "Konstruktor Sohn" << endl;
        _alter = alter;
    }
    ~Sohn() {
        cout << "Destruktor Sohn" << endl;
    }
};

int main(void) {
    Sohn s(12);
    return 0;
}
```

Ausgabe? Was fällt bei der Zuweisung an `_alter` auf?

Konstruktoren und Destruktoren

```
#include <iostream>
using namespace std;

class Mutter {
protected:
    int _alter;
public:
    Mutter() {
        cout << "Standard-Konstruktor Mutter\n";
        _alter = 30;
    }
    Mutter(int alter) {
        cout << "Konstruktor Mutter\n";
        _alter = alter;
    }
    ~Mutter() {
        cout << "Destruktor Mutter\n";
    }
};
```

Konstruktoren und Destruktoren

```
class Tochter: public Mutter {
public:
    Tochter(int alter): Mutter(alter) {
        cout << "Konstruktor Tochter\n";
    }
    ~Tochter() {
        cout << "Destruktor Tochter\n";
    }
};

int main(void) {
    Tochter t(12);
    return 0;
}
```

Ausgabe? Was fällt bei der Zuweisung an `_alter` auf?

Syntax der Initialisierungslisten

```
Klasse::Klasse(parliste)
        : basisklasse1(parliste1),
          basisklasse2(parliste2),
        var1(parliste3), var2(parliste4), ... {
    ....
}
```

In C++ ist Mehrfachvererbung möglich: Eine Klasse kann von mehreren Basisklassen erben.

- `iostream` ist von `istream` und `ostream` abgeleitet.
- Wir könnten unsere Klasse `DateTime` von einer Klasse `Date` und von einer Klasse `Time` ableiten.

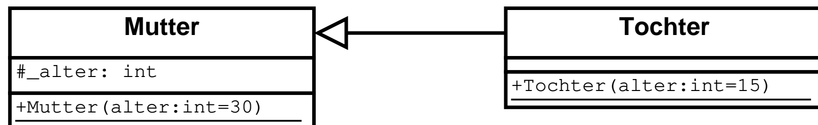
Wird hier nicht weiter betrachtet: Java, C# und andere Sprachen verzichten bewusst auf Mehrfachvererbung.

Konstruktoren und Destruktoren

- Wird für eine Basisklasse kein Konstruktor in der Initialisierungsliste angegeben, wird deren Standard-Konstruktor aufgerufen (der auch vorhanden sein muss!)
- Membervariablen werden in der Reihenfolge ihrer Deklaration in der Klasse initialisiert, denn der Nutzer kennt nur die Header-Datei, aber nicht die Implementierung!
- Anweisungen im Konstruktor werden zuletzt ausgeführt.
- Konstruktoren, Destruktoren, `operator=` sowie Freunde werden nicht vererbt!

Wir können Standardwerte für Parameter definieren:

```
Mutter(int alter = 30) {
    cout << "Konstruktor Mutter\n";
    _alter = alter;
}

Tochter(int alter = 15) :
        Mutter(alter) {
    cout << "Konstruktor Tochter\n";
}
```



Templates vs. Vererbung

In Java: Alle Klassen sind implizit Spezialisierung von der Klasse `Object`, auch selbstdefinierte Typen!

Wäre das in C++ auch so, könnte man unsere Liste auch ohne Templates wie folgt definieren:

```
class Liste {
protected:
    int last, size;
    Object *values;

public:
    void append(Object value);
    Object getValueAt(int pos);
    ....
};
```

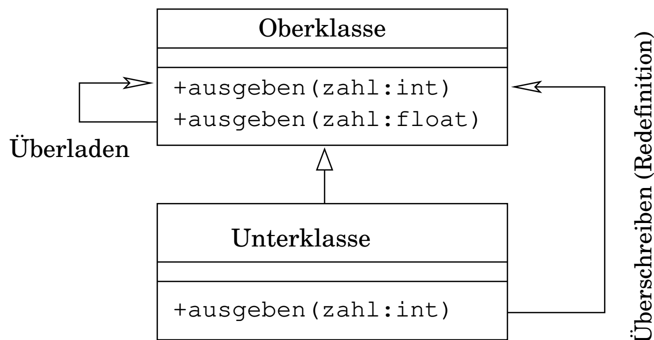
Vorteile? Nachteile?

Übersicht

Objektorientierung

- Motivation
- Funktionalität einer Klasse erweitern
- *Polymorphie*
- Abstrakte Klassen
- Klassen im Zusammenspiel
- Modellierung
- Sequenz-, Aktivitäts- und Zustandsdiagramme
- Standard Template Library

Überschreiben vs. Überladen



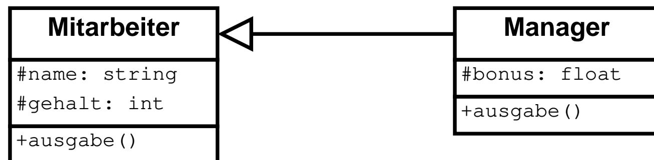
- Beim Überladen wird derselbe Funktionsname innerhalb einer Klasse mit verschiedenen Parameterschnittstellen verwendet.
- Beim Überschreiben wird eine Operation der Oberklasse mit dergleichen Signatur in der Unterklasse neu implementiert. Die *Signatur* besteht aus dem Namen der Funktion sowie der Anzahl, Reihenfolge und Typen ihrer Parameter.

Welche Ausgabe erzeugt das folgende Programm?

```
#include <iostream>
using namespace std;

void ausgabe(double &a) {
    cout << "1\n";
}
void ausgabe(const double &a) {
    cout << "2\n";
}
int main(void) {
    const double pi = 3.141592653589793;
    ausgabe(pi);
}
```

Im folgenden Beispiel haben wir eine Basisklasse `Mitarbeiter` und eine davon abgeleitete Klasse `Manager`.

Beide Klassen haben eine Methode `ausgabe()`, die den Namen und das monatliche Gehalt des Beschäftigten ausgibt. Bei Managern ist ein Bonus zu berücksichtigen.



Wir wollen untersuchen, welche Methode `ausgabe()` wann aufgerufen wird. In C++ ist dies davon abhängig,

- wie die Methoden deklariert sind und
- ob wir mit Variablen, Referenzen oder Zeigern arbeiten.

Wir definieren zunächst die Basisklasse Mitarbeiter:

```
class Mitarbeiter {
protected:
    string _name;
    int _gehalt;

public:
    Mitarbeiter(string name, int gehalt = 2000){
        _name = name;
        _gehalt = gehalt;
    }

    void ausgabe() {
        cout << "Mitarbeiter " << _name;
        cout << " : " << _gehalt << endl;
    }
};
```

Wir leiten dann die Klasse Manager von der Basisklasse ab:

```
class Manager : public Mitarbeiter {
protected:
    float _bonus;

public:
    Manager(string name, int gehalt = 2000,
            float bonus = 1.2)
                : Mitarbeiter(name, gehalt) {
        _bonus = bonus;
    }

    void ausgabe() {
        cout << "Manager " << _name << " : ";
        cout << (_gehalt * _bonus) << endl;
    }
};
```

statische Bindung

```
int main() {
    Mitarbeiter mi("Max Meier", 3000);
    Manager ma("Hans Wurst", 4000, 1.3);

    mi.ausgabe();
    ma.ausgabe();
}
```

Ausgabe:

```
    Mitarbeiter Max Meier : 3000
    Manager Hans Wurst : 5200
```

Standard in C++ ist die statische Bindung:

- Eine Methode wird zur Compile-Zeit an ein Objekt gebunden,
- eine Veränderung kann zur Laufzeit nicht vorgenommen werden.

```
int main() {
    Mitarbeiter *m[2];

    m[0] = new Mitarbeiter("Max Meier", 3000);
    m[1] = new Manager("Hans Wurst", 4000, 1.3);

    for (int i = 0; i < 2; i++)
        m[i]->ausgabe();
}
```

Ausgabe:

```
Mitarbeiter Max Meier : 3000
Mitarbeiter Hans Wurst : 4000
```

Wegen der voreingestellten statischen Bindung wird hier die Methode `Mitarbeiter::ausgabe()` an beide Objekte gebunden, obwohl das zweite Objekt vom Typ `Manager` ist.

dynamische Bindung

Damit immer die zum Objekt passende Methode aufgerufen wird, also

- bei einem `Mitarbeiter`-Objekt die Mitarbeiter-Methode
- und bei einem `Manager`-Objekt die Manager-Methode,

müssen wir die Methode ausgabe als `virtual` kennzeichnen:

```
class Mitarbeiter {
    .....
    virtual void ausgabe() {
        cout << "Mitarbeiter " << _name;
        cout << " : " << _gehalt << endl;
    }
};
```

In der abgeleiteten Klasse `Manager` ist die Methode `ausgabe` dann automatisch auch `virtual`.

Wenn wir jetzt nochmal unser Hauptprogramm ausführen,

```
int main() {
    Mitarbeiter *m[2];

    m[0] = new Mitarbeiter("Max Meier", 3000);
    m[1] = new Manager("Hans Wurst", 4000, 1.3);

    for (int i = 0; i < 2; i++)
        m[i]->ausgabe();
}
```

erhalten wir als Ausgabe:

```
Mitarbeiter Max Meier : 3000
Manager Hans Wurst : 5200
```

Die Methode wird an das tatsächliche Objekt gebunden!

dynamische Bindung

Im UML-Klassendiagramm ist keine besondere Kennzeichnung für virtuelle Methoden vorgesehen. Virtuelle Methoden sind sowohl in UML als auch in Java und anderen Sprachen Standard. Ein Überschreiben muss in Java durch Angabe von `finally` explizit verhindert werden.

Polymorphie tritt immer im Zusammenhang mit Vererbung auf.

- Es existieren verschiedene Methoden mit gleicher Signatur in unterschiedlichen Ebenen einer Vererbungshierarchie.
- Erst zur Laufzeit wird bestimmt, welche der Methoden für ein gegebenes Objekt verwendet wird.

dynamische Bindung

Gleichnamige virtuelle Methoden mit unterschiedlichen Argumenten sind echt verschieden und überschreiben sich nicht.

```
struct B {
    virtual int m(int p1) {         // M1
        cout << "m(int) in B\n";
    }
    int m(int p1, int p2) {         // M2
        cout << "m(int, int) in B\n";
    }
};
struct A : public B {
    int m(int p1) {                 // M3
        cout << "m(int) in A\n";
    }
    int m(char *p1) {               // M4
        cout << "m(char *) in A\n";
    }
};
```

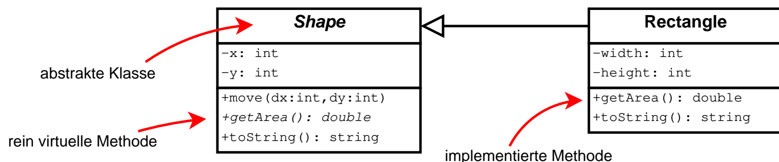
Anmerkungen:

- Der Konstruktor kann nicht als `virtual` definiert werden.
- Manchmal können Methoden in der Basisklasse noch nicht implementiert werden. Trotzdem möchte man sicherstellen, dass alle abgeleiteten Klassen diese Methode bereitstellen. Dann wird die Methode in der Basisklasse zwar definiert, aber nicht implementiert. Solche Methoden heißen rein virtuelle Methoden.

  Syntax: `virtual <type> methodname(<params>) = 0;`
- Rein virtuelle Methoden werden im UML-Diagramm kursiv dargestellt.
- Klassen, die rein virtuelle Methoden enthalten, heißen abstrakte Klassen.
- Man kann keine Objekte von abstrakten Klassen erzeugen.

In der Basisklasse `Shape`, die ein allgemeines geometrisches Objekt darstellt, kann der Flächeninhalt des Objektes nicht berechnet werden.

In der konkreten, abgeleiteten Klasse `Rectangle` kann der Flächeninhalt einfach bestimmt werden.



Die Basisklasse `Shape` erzwingt, dass alle konkreten geometrischen Objekte die Methode `getArea()` zur Verfügung stellen.

```
#include <iostream>
using namespace std;

class Shape {
protected:
    int _x, _y;

public:
    void move(int dx, int dy) {
        _x += dx;
        _y += dy;
    }

    virtual string toString() { ..... } // ???

    // rein virtuelle Methode:
    virtual double getArea() = 0;
};
```

```
class Rectangle: public Shape {
private:
    int _w, _h;

public:
    Rectangle(int x, int y, int w, int h) {
        _x = x;             // vererbt von Shape
        _y = y;             // vererbt von Shape
        _w = w;
        _h = h;
    }

    string toString() { ..... }       // ?????

    double getArea() {    // definiert getArea()
        return _w * _h;
    }
};
```

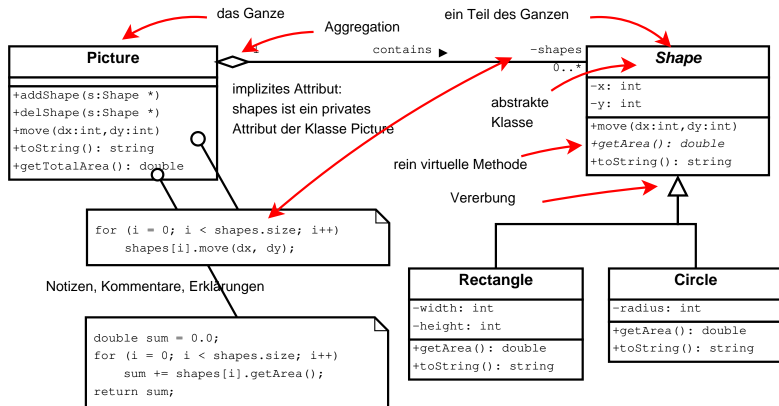
```
class Circle: public Shape {
private:
    int _r;

public:
    Circle(int x, int y, int r) {
        _x = x;              // vererbt von Shape
        _y = y;              // vererbt von Shape
        _r = r;
    }

    string toString() { ..... }        // ?????

    double getArea() {     // definiert getArea()
        return _r * _r * 3.14159265358979 3;
    }
};
```

dynamische Bindung

```
class Picture {
private:
    Liste<Shape *> _shapes;

public:
    void addShape(Shape *s) {
        _shapes.append(s);
    }

    void delShape(Shape *s) {
        _shapes.remove(s);
    }

    void move(int dx, int dy) {
        for (int i = 0; i < _shapes.size(); i++)
            _shapes[i]->move(dx, dy);
    }
    ....
```

```
    string toString() {
        string res;

        for (int i = 0; i < _shapes.size(); i++)
            res += _shapes[i]->toString();
        return res;
    }

    double getTotalArea() {
        double sum = 0.0;

        for (int i = 0; i < _shapes.size(); i++)
            sum += _shapes[i]->getArea();
        return sum;
    }
};
```

```
int main(void) {
    Picture pic;

    pic.addShape(new Rectangle(2, 0, 1, 2));
    pic.addShape(new Rectangle(0, 2, 3, 7));
    pic.addShape(new Circle(0, 1, 3));
    pic.addShape(new Circle(1, 0, 4));

    cout << "Summe Flaecheninhalt: "
         << pic.getTotalArea() << endl;

    cout << " vorher:\n" << pic.toString();

    pic.move(3, 4);
    cout << "nachher:\n" << pic.toString();
}
```



Delegation: Mechanismus, bei dem ein Objekt eine Nachricht nicht (vollständig) selbst interpretiert, sondern an ein anderes Objekt weiterleitet.

Typprüfung zur Laufzeit

Wie funktioniert die Typprüfung zur Laufzeit?

- Der Compiler erweitert den Code um eine Tabelle.
- Die Tabelle enthält für jede Klasse alle virtuellen Methoden.
- Anhand des Typs eines Objekts kann in dieser Tabelle die passende Methode gefunden werden.
- Dieses Nachschlagen in der Tabelle führt zu einer geringfügigen Verlangsamung des Programms.

Redefinition ist kein Polymorphismus!

Typprüfung zur Laufzeit

im Detail:

- Jeder Methodenaufruf wird zunächst anhand einer (laufenden) Nummer beschrieben.
- Für jede Klasse wird eine Virtual Method Table (VMT) aufgestellt, die für jede Methode den Verweis auf die Adresse der Methode enthält.
- Alle Objekte erhalten automatisch einen Verweis auf die VMT ihrer Klasse.
  - Lassen Sie sich zur Kontrolle `sizeof(Rectangle)` ausgeben!
  - Die Differenz zwischen angezeigtem Wert und der eigentlichen Größe sollte genau der Größe eines Zeigers entsprechen.
- Zur Laufzeit wird mit der Methodennummer aus dem Methodenaufruf die aktuelle VMT untersucht und daraus die Adresse der Methode ermittelt.

Typprüfung zur Laufzeit

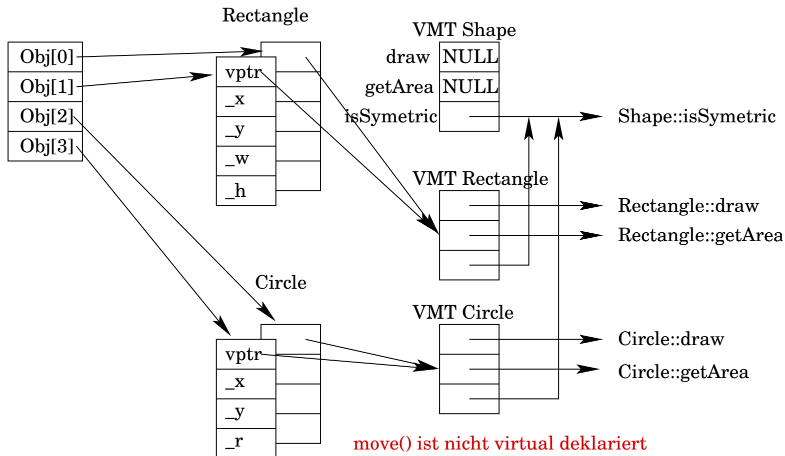
Betrachten wir eine Variante unseres Shape-Beispiels:

```
class Shape {
protected:
    int _x, _y;

public:
    virtual void draw() = 0;
    virtual float getArea() = 0;

    void move(int dx, int dy) {
        _x += dx;
        _y += dy;
    }
    virtual bool isSymetric() {
        return true;
    }
};
```

Typprüfung zur Laufzeit

```
class Rectangle: public Shape {
protected:
    int _w, _h;

public:
    void draw() {          // hier implementieren
        .....
    }
    float getArea() {      // hier implementieren
        return _w * _h;
    }
    // move()              wird von Shape geerbt
    // isSymetric()        wird von Shape geerbt
};

class Circle: public Shape {
    .....   // analog zu Rectangle
};
```

Typprüfung zur Laufzeit



move() ist nicht virtual deklariert

Typprüfung zur Laufzeit

Wenn Sie unter Linux den C++-Compiler `g++` mit der Option `-fdump-class-hierarchy` aufrufen, werden Ihnen die angelegten virtual method tables angezeigt:

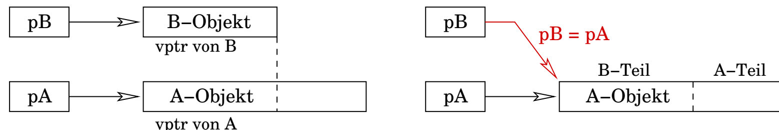
```
Vtable for Shape
Shape::_ZTV5Shape: 5u entries
0     (int (*)(...))0
4     (int (*)(...))(& _ZTI5Shape)
8     __cxa_pure_virtual
12    __cxa_pure_virtual
16    Shape::isSymetric
```

Typprüfung zur Laufzeit

```
Vtable for Rectangle
Rectangle::_ZTV9Rectangle: 5u entries
0     (int (*)(...))0
4     (int (*)(...))(& _ZTI9Rectangle)
8     Rectangle::draw
12    Rectangle::getArea
16    Shape::isSymetric

Vtable for Circle
Circle::_ZTV6Circle: 5u entries
0     (int (*)(...))0
4     (int (*)(...))(& _ZTI6Circle)
8     Circle::draw
12    Circle::getArea
16    Shape::isSymetric
```

Typprüfung zur Laufzeit

Im Folgenden sei `pB` vom Typ „Zeiger auf Basisklasse“ und `pA` vom Typ „Zeiger auf abgeleitete Klasse“.



- Der „Basisteil“ der abgeleiteten Klasse ist ansprechbar.
- Jedes Objekt besitzt einen virtual table pointer (vptr), der auf die VMT seiner Klasse zeigt.
- Nach dem „umbiegen“ von `pB` auf das A-Objekt `pA` arbeitet `pB` mit dem `vptr` vom Objekt A, es wird also immer die passende Methode aufgerufen.
- Mit `pB` kann nicht auf Methoden oder Attribute der abgeleiteten Klasse zugegriffen werden!

Destruktoren bei „virtuellen Klassen“

Der Einsatz virtueller Methoden in einer Klasse verlangt, dass auch der Destruktor dieser Klasse `virtual` ist, damit der richtige Destruktor entsprechend des tatsächlichen Typs des Objekts aufgerufen wird!

```
class B {
public:
    virtual ~B();
    virtual int m();
};
```

```
class A: public B
public:
    ~A();
    int m();
 };
```

```
B *pB;
pB = new A();    // !!!!!
delete pB;       // !!!!!
```

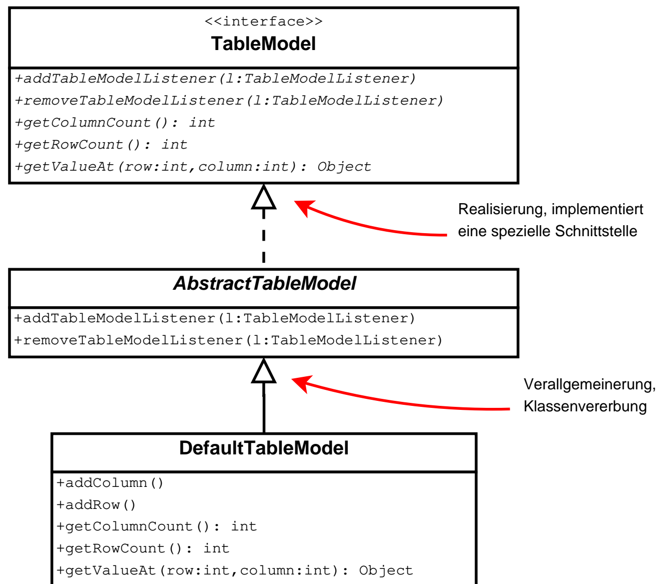
Übersicht

Objektorientierung

- Motivation
- Funktionalität einer Klasse erweitern
- Polymorphie
- *Abstrakte Klassen*
- Klassen im Zusammenspiel
- Modellierung
- Sequenz-, Aktivitäts- und Zustandsdiagramme
- Standard Template Library

- Eine Klasse, die mindestens eine rein virtuelle Methode enthält, nennen wir abstrakte Klasse. In einer abstrakten Klasse sind also nicht alle Methoden implementiert.
- Als Folge der nicht-implementierten Methoden können keine Objekte abstrakter Klassen erzeugt werden.
- Rein virtuelle Methoden nennen wir auch abstrakte Methoden.
- Abstrakte Klassen definieren Schnittstellen, sie legen fest, was eine Klasse können soll. Die Realisierung der Funktionalität muss dann in einer konkreten Unterklasse erfolgen.
- Eine von einer abstrakten Klasse abgeleiteten Klasse muss alle vererbten abstrakten Methoden implementieren, damit die erbende Klasse selbst nicht abstrakt ist.
- In Java: Ein `Interface` ist eine besondere Form einer Klasse, die nur abstrakte Methoden sowie Konstanten enthält.

Abstrakte Klassen



Abstrakte Klassen

- Das Interface `TableModel` legt die Schnittstelle fest, beschreibt also, welche Methoden implementiert werden müssen.
- Die abstrakte Klasse `AbstractTableModel` stellt eine partielle Implementierung bereit: Nicht alle, aber einige Methoden werden implementiert.

  Es werden solche Methoden implementiert, die wahrscheinlich für die meisten Realisierungen vom `TableModel` nutzbar sind, bspw. das An- und Abmelden von Beobachtern (siehe Entwurfsmuster Beobachter).
- Am Ende der Hierarchie steht eine Implementierung, die in vielen Projekten nutzbar ist, das `DefaultTableModel`.

  Diese Klasse kann bei speziellen Bedürfnissen weiter spezialisiert werden.

Übersicht

Objektorientierung

- Motivation
- Funktionalität einer Klasse erweitern
- Polymorphie
- Abstrakte Klassen
- *Klassen im Zusammenspiel*
- Modellierung
- Sequenz-, Aktivitäts- und Zustandsdiagramme
- Standard Template Library

Aufteilung in Klassen

Programme bestehen in der Regel aus mehreren Klassen.

Frage: Welche Klassen benötigen wir, um ein gegebenes Problem zu lösen?

Einfache Grundregel:

- Jedes Substantiv stellt eine potentielle Klasse dar.
- Jedes Verb beschreibt eine mögliche Methode.

Problem: Welche Substantive und Verben in der Aufgabenstellung vorkommen, hängt stark von der Formulierung ab!

Aufteilung in Klassen

Aufgabenstellung 1:

- In einer Bank können Personen ein Konto anlegen.
- Zu einem Konto wird der Kontostand, die Person, eine PIN und das Eröffnungsdatum gespeichert.
- Das Konto ermöglicht Ein- und Auszahlungen sowie Überweisungen.

Damit erhalten wir:

- Substantive: Bank, Person, Konto, Kontostand, PIN, Datum, Ein- und Auszahlung, Überweisung
- Verben: Konto anlegen, Kontostand speichern, Überweisungen ermöglichen

Aufteilung in Klassen

Aufgabenstellung 2:

- Eine Bank verwaltet Konten. Jedes Konto ist einer Person zugeordnet.
- Eine Person ist gekennzeichnet durch Vor- und Zuname, Adresse und Geburtsdatum.
- Eine Person kann Geld auf das Konto einzahlen, vom Konto Geld abheben oder auf ein anderes Konto überweisen.

Damit erhalten wir:

- Substantive: Bank, Person, Konto, Vor- und Zuname, Adresse, Datum, Geld
- Verben: Konto verwalten, einzahlen, abheben, überweisen

Aufteilung in Klassen

Wie wir sehen, hängen die Substantive und Verben sehr stark von der Formulierung der Aufgabenstellung ab.

- Variante 1:
  - Substantive: Bank, Person, Konto, Kontostand, PIN, Datum, Ein- und Auszahlung, Überweisung
  - Verben: Konto anlegen, Kontostand speichern, Überweisungen ermöglichen
- Variante 2:
  - Substantive: Bank, Person, Konto, Vor- und Zuname, Adresse, Datum, Geld
  - Verben: Konto verwalten, einzahlen, abheben, überweisen

Unsere einfache Regel *Substantive sind mögliche Klassen, Verben sind mögliche Methoden* scheint also nicht zu funktionieren, oder?

Aufteilung in Klassen

Doch! Aus verschiedenen Formulierungen können wir mit gesundem Menschenverstand Klassen und Attribute sehr gut erkennen:

- Die Substantive `Bank`, `Konto` und `Person` kommen in beiden Formulierungen vor.
- Die Attribute einer `Person` sind `Vorname`, `Zuname`, `Adresse` und `Geburtsdatum`.
- Unter `Konto` fassen wir die Attribute `Kontostand`, `PIN`, `Eröffnungsdatum` und `Inhaber` zusammen.
- Ein-/Auszahlung und Überweisung sind nur andere Formulierungen für einzahlen, abheben und überweisen und stellen somit Methoden der Klasse `Konto` dar.

Aufteilung in Klassen

- `PIN` und `Kontostand` sind einfache Datentypen: `string` bzw. `double`
- `Bank` ist quasi eine Sammlung von `Konten` und `Personen` und hat zusätzlich das Attribut `BLZ`.
- Operationen der `Bank` sind `kontoAnlegen`, `kontoLoeschen` und `kundeAnlegen`.
- Verschiedene Kontoarten wie `Sparkonto`, `Girokonto` usw. können wir später einfügen.

Aufteilung in Klassen

- In einem ersten Entwurf darf die `Adresse` vom Datentyp `string` sein.

  Falls wir aber bspw. alle Kunden eines bestimmten Ortes oder einer bestimmten Straße auflisten wollen, wäre es sinnvoll, `Adresse` als Struktur mit den Attributen `strasse`, `plz` und `ort`, ggf. mit `postfach` zu vereinbaren.
- Für die Zinsberechnung benötigen wir ggf. eine Klasse `Datum`, um bspw. die Differenz in Tagen zwischen zwei Einzahlungen zu ermitteln.

Aufteilung in Klassen

Klassen und zugehörige Methoden für das Beispiel:

- `Bank:` `addAccount()`, `eraseAccount()`, `addCustomer()`
- `Konto:` `withdraw(), payInto(), transfer(), balance()`
- `Person:` `changeName(), addAccount(), isCreditWorthy()`

Hilfsklassen:

- `Datum:` `isLeapYear(), getDayInYear(), add(days)`
- `Kontobewegung:` Struktur mit Attributen `datum, von, nach, betrag, info`

Wichtig: Eine Klasse – eine Aufgabe!

Aufteilung in Klassen

```
Bank
-kunden: Liste<Person>
-konten: Liste<Konto>
+blz: string
+addAccount(acc:Konto)
+rmAccount(acc:Konto)
+addCustomer(cust:Person)
```

```
Person
-name: string
-vorname: string
-adresse: string
-gebDatum: Datum
+addAccount(k:Konto)
+rmAccount(k:Konto)
+isCreditWorthy(): bool
```

```
Kontobewegung
+datum: Datum
+von: string
+nach: string
+info: string
+betrag: int
```

```
Datum
-day: int
-month: int
-year: int
+isLeapYear(): bool
+getDayInWeek(): int
+getDayInYear(): int
+getWeekInYear(): int
+add(days:int)
+diff(date:Datum): int
```

```
Konto
-eroeffnetAm: Datum
-inhaber: Person
-PIN: string
-nr: string
-stand: int
+withdraw(amount:int)
+payInto(amount:int)
+transfer(to:Konto,amount:int)
+getBalance(): int
```

Welche Beziehungen gibt es zwischen den Klassen? Und wie werden diese Beziehungen dargestellt?

Aufteilung in Klassen

Welche Beziehungen gibt es zwischen den Klassen?

- *Vererbung:* Spezialisierung einer anderen Klasse.
- *Assoziation:* Man kennt und hilft sich, indem gegenseitig Methoden aufgerufen werden.
- *Aggregation und Komposition:* Objekte sind Teile eines anderen Objekts.

Wie werden diese Beziehungen

- in UML dargestellt?
- in C++ implementiert?

Eine Aggregation ist eine Relation zwischen Klassen, deren beteiligte Klassen eine Ganzes-Teile-Hierarchie darstellen.

Die *Kardinalität* gibt an, mit wievielen Objekten der gegenüber liegenden Klasse das Objekt in Beziehung steht:

- 0..1 Ganzes besteht aus * Teilen

Aggregation

Beispiel:

- 1 Unternehmen besteht aus 1..* Abteilungen
- 1 Abteilung besteht aus 1..* Mitarbeitern

Aggregation

- Das Ganze (Aggregat) nimmt stellvertretend für seine Teile Aufgaben wahr:

  Die Aggregatklasse enthält Operationen, die keine unmittelbare Veränderung im Aggregat selbst bewirken, sondern die Nachricht an seine Einzelteile weiterleitet.

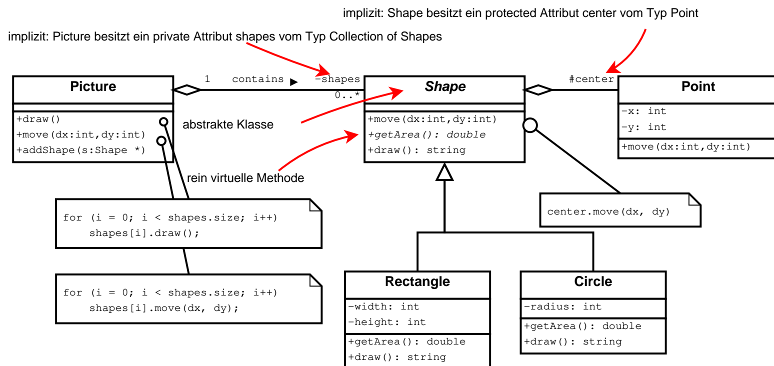
$\Rightarrow$ Die beteiligten Klassen sind nicht gleichberechtigt:

  Das Aggregat übernimmt stellvertretend für die Einzelteile die Verantwortung und Führung.

Was heißt das? Wie wird das in C++ implementiert?

Aggregation

Noch einmal eine Variante unseres Shape-Beispiels, diesmal erweitert um eine Klasse `Point`, die den Mittelpunkt eines geometrischen Objekts darstellen soll:



```
class Point {
    friend class Shape;
    // warum ist das notwendig???

private:
    int _x, _y;

public:
    Point(int x = 0, int y = 0) {
        _x = x;
        _y = y;
    }

    void move(int dx, int dy) {
        _x += dx;
        _y += dy;
    }
};
```

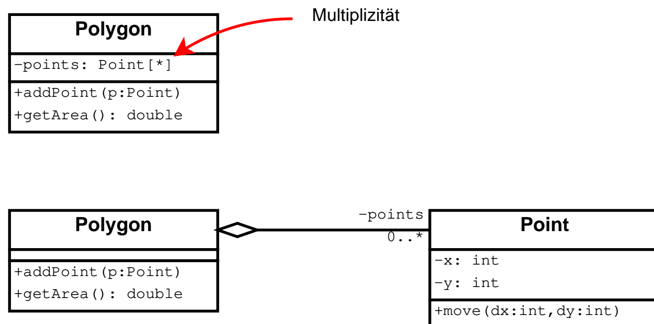
```
class Shape {
protected:
    Point _center;
public:
    Shape(Point p = 0) {
        _center = p;
    }
    virtual string draw() {   // Standard-
        ostringstream os;     // Implementierung
        os << "(" << _center._x << ", ";
        os << _center._y << ")" << endl;
        return os.str();
    }
    void move(int dx, int dy) {     // Delegation
        _center.move(dx, dy);
    }
    virtual double getArea() = 0;
};
```

```
class Rectangle: public Shape {
protected:
    int _w, _h;

public:
    Rectangle(Point p, int w, int h): Shape(p) {
        _w = w;
        _h = h;
    }
    string draw() {     // Basis-Implementierung
        ostringstream os;    // plus Erweiterung
        os << "Rect:" << _w << "," << _h << ",";
        return os.str() + Shape::draw();
    }
    double getArea() {    // definiert getArea()
        return _w * _h;
    }
};
```

```
class Picture {
private:
    Liste<Shape *> shapes;
public:
    void addShape(Shape *s) {
        shapes.append(s);
    }
    string draw() {
        string res;
        for (int i = 0; i < _shapes.size(); i++)
            res += _shapes[i]->draw();
        return res;
    }
    void move(int dx, int dy) {
        for (int i = 0; i < shapes.size(); i++)
            shapes[i]->move(dx, dy);
    }
};
```

Aggregation

Wir können die Klasse `Point` auch für ein `Polygon` nutzen. Die beiden folgenden Darstellungen sind gleichwertig:



Wie die Speicherung der Punkte erfolgt (Liste, Array, ...), wird durch das Klassendiagramm nicht festgelegt. Aber die Art der Speicherung hat Auswirkungen darauf, wie effizient die Methoden der Klasse implementiert werden können.

Aggregation

Beide folgenden Implementierungen entsprechen den obigen Darstellungen:

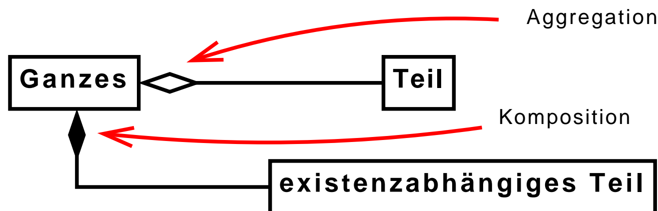
```
class Polygon {
    Liste<Point> points;

public:
    void addPoint(Point p) {
        points.append(p);
    }
    double getArea() {
        .....
    }
};
```

Werden die Punkte in einem Array gespeichert, ist die Implementierung komplizierter:

Aggregation

```
class Polygon {
    Point *points;
    int size, next;
public:
    Polygon() {
        next = 0;
        size = 8;
        points = new Point[size];
    }
    void addPoint(Point p) {
        if (next == size)
            .....  // resize array

        points[next] = p;
        next += 1;
    }
    double getArea();
};
```

Komposition ist eine strengere Form der Aggregation: die Teile sind vom Ganzen existenzabhängig, d.h. die Teile können ohne das Ganze nicht existieren. (nicht eindeutig in der Literatur)



Komposition

Das Ganze ist verantwortlich für das Erzeugen und Löschen der Teile.

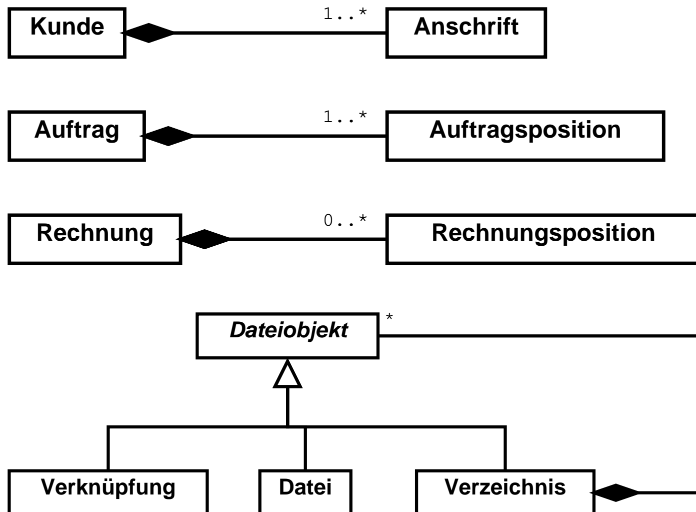
- Beim Erzeugen des Aggregat-Objekts werden auch die einzelnen Teile erzeugt.
- Wird das Ganze gelöscht, werden automatisch auch alle seine Teile gelöscht.

Ein Teil ist jedoch nicht untrennbar mit seinem Ganzen verbunden, sondern darf vorher einem anderen Aggregat-Objekt zugeordnet werden. Beispiel:

> Eine Datei kann in ein anderes Verzeichnis verschoben werden, bevor das Verzeichnis gelöscht wird.

Komposition

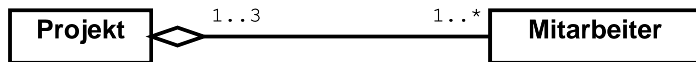
Beispiele:



Aggregation vs. Komposition

Bei einer *Aggregation* ist die Kardinalität auf der Seite des Aggregats oft 1 (Standard, wenn Angabe fehlt).

Das ist aber nicht zwingend so: Ein Teil kann gleichzeitig zu mehreren Aggregationen gehören.



Wird in C++ oft realisiert über Zeiger:

```
class Projekt {
private:
    Liste<Mitarbeiter *> mitarbeiter;
    .....
};
```

Warum realisieren wir das mittels Zeiger?

Aggregation vs. Komposition

Fortsetzung Aggregation:

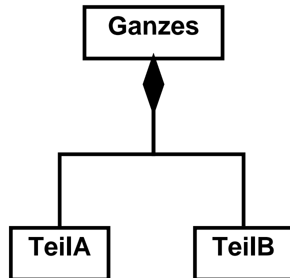
- Ein `Mitarbeiter` kann in bis zu drei Projekten arbeiten.
- Redundanz vermeiden: Wenn viele Kopien eines Mitarbeiters im System existieren würden, gäbe es Probleme z.B. bei einer Adress- oder Namensänderung: Alle Kopien müssten geändert werden.
- Daher arbeitet man anstelle von Kopien nur mit Zeigern auf das Original.
- Endet ein Projekt, darf der Destruktor der Klasse `Projekt` und der Klasse `Liste` nicht den Mitarbeiter löschen, nur der Zeiger wird entfernt.

Aggregation vs. Komposition

Bei einer *Komposition* kann die Kardinalität auf der Seite des Aggregats nur 1 sein. Jedes Teil ist Teil genau eines Kompositionsobjekts, sonst gäbe es einen Widerspruch zur Existenzabhängigkeit!

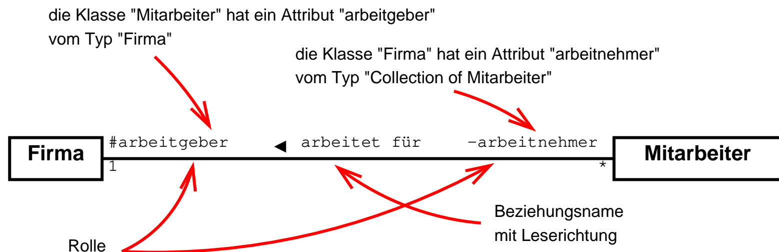
Realisierung in C++:

```
class Ganzes {
    TeilA _tA;
    TeilB *_tB;
public:
    Ganzes() {
        _tB = new TeilB();
    }
    ~Ganzes() {
        delete _tB;
    }
};
```



Die *Assoziation* ist eine schwächere Form der Aggregation: eine gleichrangige Beziehung zwischen Objekten, d.h. die Objekte kennen sich, haben aber keine stärkere Beziehung zueinander.

Beispiel:



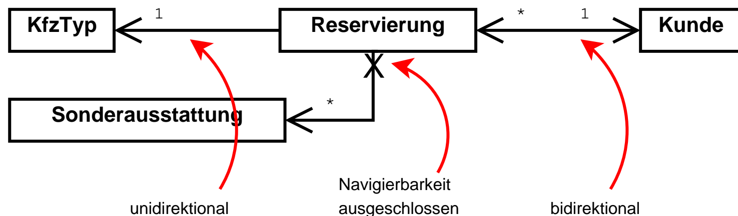
Hatten wir dieses Beispiel nicht schon einmal bei Aggregation?

Assoziation

Nochmal zur *Realisierung:* die beteiligten Klassen erhalten entsprechende Referenzattribute:

- Die Klasse `Mitarbeiter` hat ein Attribut `arbeitgeber` als Referenz auf ein Objekt der Klasse `Firma`.
- Die Klasse `Firma` hat ein Attribut `arbeitnehmer` mit einer Sammlung (engl.: Collection, z.B. Array oder Liste) von `Mitarbeiter`-Objekten.
- Diese Attribute werden nicht explizit in den jeweiligen Attribut-Rubriken der Klassen aufgeführt und die Attribute haben die bekannten Sichtbarkeitsattribute.

gerichtete Assoziation: Man kann von einer beteiligten Rolle zur anderen navigieren, aber nicht umgekehrt.



- Man kann von der `Reservierung` zum `KfzTyp` navigieren, die andere Richtung ist noch undefiniert.
- Man kann sowohl von `Kunde` zur `Reservierung` navigieren, als auch umgekehrt.
- Man kann von `Reservierung` zur `Sonderausstattung` navigieren, aber nicht umgekehrt.

Assoziation

Problem: Oft ist die Abgrenzung zwischen Assoziation und Aggregation schwierig.

- Die Stabilität eines Software-Produkts ist nicht abhängig vom *richtigen* Symbol im Klassendiagramm (Raute oder nicht).
- **Aber:** Analysiere, wie eng die Beziehung zwischen den Objekten der Klassen ist und welche Konsequenzen sich daraus ergeben.

Beispiel: Die Beziehung zwischen `Rechnung` und `Anschrift` ist nur ein Verweis.

- Vorteil: Ist weniger speicherintensiv, als wenn jede `Rechnung` eine Kopie der `Anschrift` aggregiert.
- Nachteil: Eine Änderung der `Anschrift` hat auch Auswirkungen auf jede vergangene `Rechnung` des Kunden.

Lernkontrolle

Was ist der Unterschied zwischen folgender Modellierung?

Übung

Welche Beziehungen zwischen den Klassen unseres Bankbeispiels gibt es?

- Assoziation?
- Aggregation?
- Komposition?

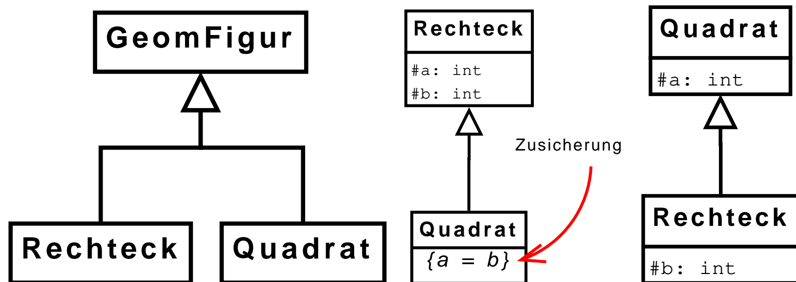
Übersicht

Objektorientierung

- Motivation
- Funktionalität einer Klasse erweitern
- Polymorphie
- Abstrakte Klassen
- Klassen im Zusammenspiel
- *Modellierung*
- Sequenz-, Aktivitäts- und Zustandsdiagramme
- Standard Template Library

Vererbung ist nicht der Weisheit letzter Schluss. Zu oft setzen wir Vererbung ein, obwohl es nicht sinnvoll ist.

Wir werden uns einige Beispiele ansehen, wo Vererbung nicht angebracht ist und durch Delegation ersetzt werden sollte.

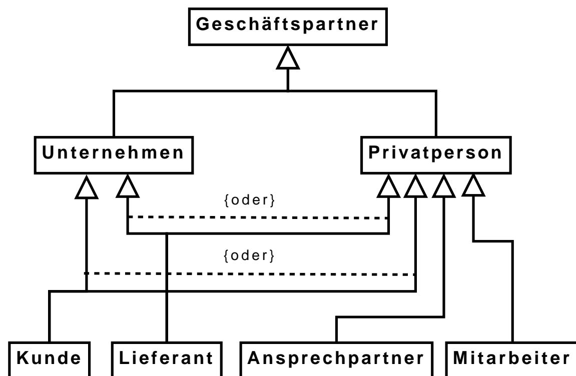


Vererbung kritisch prüfen:

- In diesem Beispiel ist `Geschäftspartner` ein Oberbegriff für `Kunde`, `Lieferant`, `Ansprechpartner` und `Mitarbeiter` und
- die Begriffe `Unternehmen` sowie `Privatperson` existieren nicht eigenständig, sondern nur im Zusammenhang mit einem Geschäftspartner.

  Ein Kunde kann eine Privatperson oder ein Unternehmen sein.

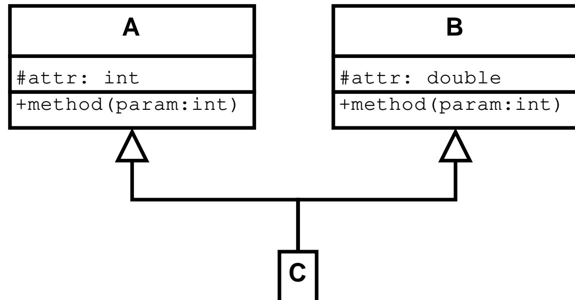
Versuch 1:



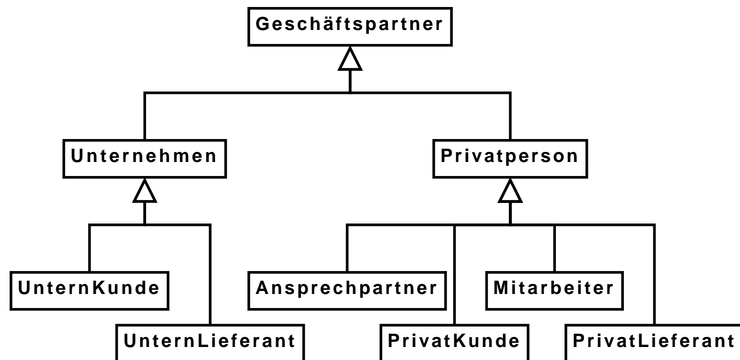
Problem: Wir benötigen Mehrfachvererbung mit exklusivem Oder.

Warum ist Mehrfachvererbung ein Problem?

- Viele objektorientierte Programmiersprachen unterstützen Mehrfachvererbung nicht, z.B. Java, C# und Smalltalk.
- Welche Eigenschaft soll geerbt werden, wenn die Oberklassen gleichnamige Eigenschaften besitzen?

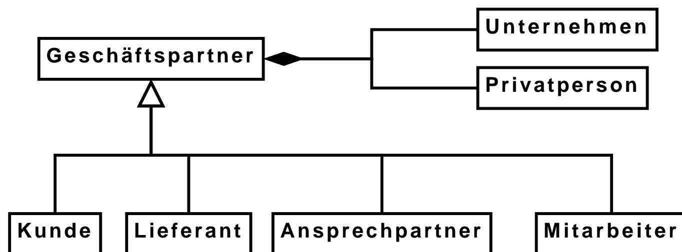


Versuch 2:



Problem: Kombinatorische Explosion von Möglichkeiten, wobei die in `UnternXY` und `PrivatXY` anzusiedelnden Eigenschaften allerdings weitgehend identisch sind.

Versuch 3:



OCL-Ausdrücke
(Object Constraint Language)

Geschäftspartner:

self.Privatperson->notEmpty implies self.Unternehmen->isEmpty
self.Privatperson->isEmpty implies self.Unternehmen->notEmpty
self.Unternehmen->notEmpty implies
    (self.isKindOf(Kunde) or self.isKindOf(Lieferant))

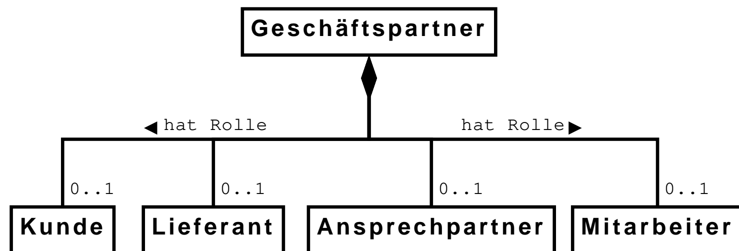
Problem: sehr viele Zusicherungen

Ist Vererbung hier angebracht? Nein!

- Lieferanten und Mitarbeiter können auch Kunden sein!
- Ein Objekt kann seine Klassenzugehörigkeit nicht wechseln, z.B. von Lieferant zu Kunde.

  Das wäre allerdings auch falsch: Der Geschäftspartner ist nicht abwechselnd mal Lieferant und mal Kunde, sondern beides gleichzeitig.

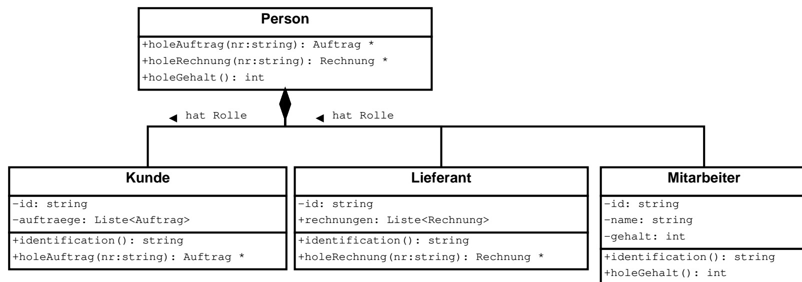
→ Kunde, Lieferant usw. sind keine Spezialisierungen von Geschäftspartner, sondern mögliche Eigenschaften!



Eine Rolle definiert eine spezielle Sichtweise auf ein Objekt: Die Anwender des Systems nehmen die Geschäftspartner in bestimmten Situationen in einer bestimmten Rolle wie Lieferant oder Kunde wahr.

Wie implementiert man das?

Vielleicht so?

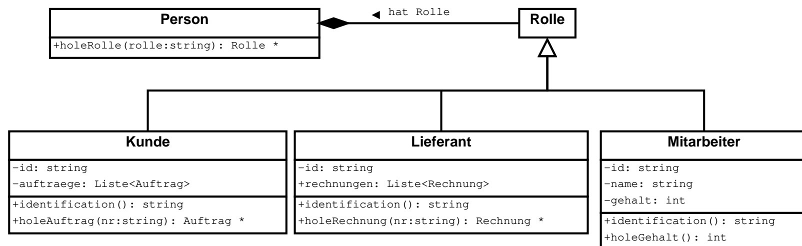



```
Person pers;
.....
Auftrag *auftr = pers.holeAuftrag("4711");
Rechnung *rech = pers.holeRechnung("0815");
```

Das wäre wohl nicht so günstig, denn

- die Klasse `Person` müsste jedesmal erweitert werden, wenn eine neue Rolle hinzu kommt.
- die Modellierung ist falsch: Die Methode `holeGehalt` ist eine Methode der Klasse `Mitarbeiter`, aber nicht der Klasse `Person`.

Wie sollen wir es denn dann implementieren? Wie kann eine Person alle Rollen annehmen, also die Schnittstellen der Klassen `Kunde`, `Mitarbeiter`, `Lieferant` usw. bereitstellen?




```
Person p;
.....
try {
    Kunde *k = (Kunde *) p.holeRolle("Kunde");
    Auftrag *auftr = k->holeAuftrag("4711");
} catch (NoSuchRoleException e) { ..... }
```

Modellierung

```
class Person {
private:
    Kunde *kunde;
    Lieferant *lieferant;
.....
};

Rolle * Person::holeRolle(string rolle) {
    if (rolle == "Kunde")
        return kunde;
    if (rolle == "Lieferant")
        return lieferant;
    .....
    throw NoSuchRoleException(rolle);
}
```

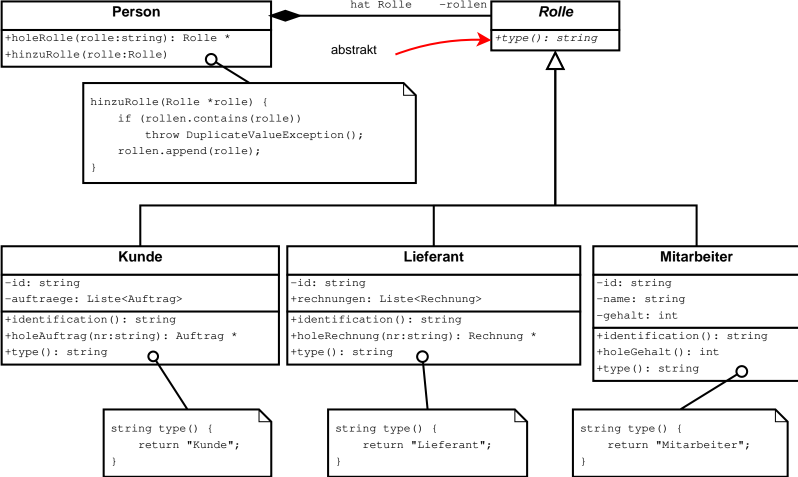
Was halten Sie von dieser Implementierung?

Problem: Fügen wir eine neue Rolle hinzu, müssen wir die Methode `holeRolle()` erweitern.

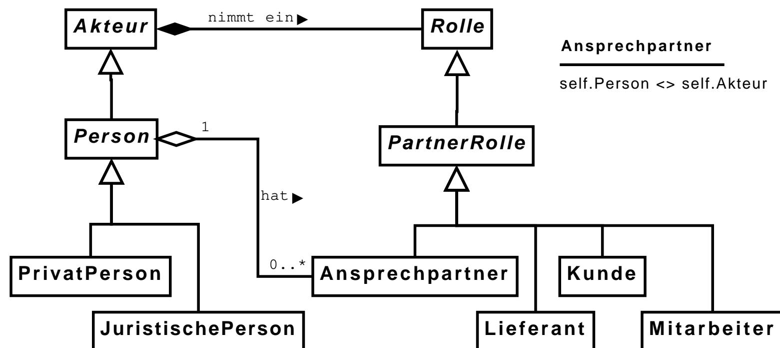
Lösung:

- Die Klasse `Person` speichert eine Liste von `Rollen`, die beliebig erweitert werden kann.
- Jede konkrete Rolle hat eine ID, die mittels `type()` abgefragt werden kann.

```
Rolle * Person::holeRolle(string rolle) {
    for (int i = 0; i < rollen.size(); i++)
        if (rollen[i]->type() == rolle)
            return rollen[i];
    throw NoSuchRoleException(rolle);
}
```


Person
+holeRolle(rolle:string): Rolle *
+hinzuRolle(rolle:Rolle)
hat Rolle
-rollen
Rolle
+type(): string
abstrakt
hinzuRolle(Rolle *rolle) {
if (rollen.contains(rolle))
throw DuplicateValueException();
rollen.append(rolle);
}
Kunde
-id: string
-auftraege: Liste<Auftrag>
+identification(): string
+holeAuftrag(nr:string): Auftrag *
+type(): string
Lieferant
-id: string
+rechnungen: Liste<Rechnung>
+identification(): string
+holeRechnung(nr:string): Rechnung *
+type(): string
Mitarbeiter
-id: string
-name: string
-gehalt: int
+identification(): string
+holeGehalt(): int
+type(): string
string type() {
return "Kunde";
}
string type() {
return "Lieferant";
}
string type() {
return "Mitarbeiter";
}

Akteur-Rolle-Entwurfsmuster



Zusicherung: Eine Person darf sich nicht selbst als Ansprechpartner referenzieren.

Frage: Warum ist die Klasse `Person` von `Akteur` abgeleitet?

Akteur-Rolle-Entwurfsmuster

Um gemeinsame Funktionalität für verschiedene Akteure bereitzustellen:

```
class Akteur {
private:
    Liste<Rolle *> rollen;
    .....
public:
    void hinzuRolle(Rolle *r) {
        if (rollen.contains(r))
            throw DuplicateValueException();
        rollen.append(r);
    }
    .....
};
```

Die Methode `hinzuRolle()` muss in der Klasse `Person` und allen Unterklassen der Basisklasse nicht mehr implementiert werden.

Akteur-Rolle-Entwurfsmuster

Frage: Warum sind die Klassen `Kunde`, `Mitarbeiter` usw. von der Klasse `Rolle` abgeleitet?

Die Methode `hinzuRolle(Rolle *r)` der Klasse `Person` erwartet als Parameter eine *allgemeine Rolle*, da wir die konkreten Rollen zum Zeitpunkt der Definition von `Person` noch gar nicht kennen.

```
Person pers;
.....
pers.hinzuRolle(new Kunde(...));
pers.hinzuRolle(new Mitarbeiter(...));
pers.hinzuRolle(new Lieferant(...));
```

Akteur-Rolle-Entwurfsmuster

Frage: Wie kann eine `Person` gleichzeitig `Mitarbeiter` und `Kunde` sein?

Akteur-Rolle-Entwurfsmuster

Idee wurde bereits bei der vorigen Antwort beschrieben:

```
Liste<Person *> kundenliste;
Liste<Person *> mitarbliste;
...
Adresse adr("Rosenweg", "7b", "01234", "Kaff");
Person *p = new Person("Meier", "Max", adr);
p->hinzuRolle(new Kunde(...));
p->hinzuRolle(new Mitarbeiter(...));
...
kundenliste.append(p);
mitarbliste.append(p);
```

Hinweis: Akteur-Rolle-Entwurfsmuster wird nicht im Buch der *Gang of Four* beschrieben.

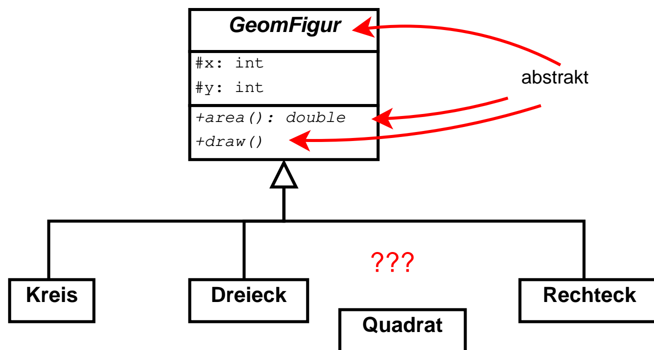
Akteur-Rolle-Entwurfsmuster

Ohne dieses Muster sähe es so aus:

```
Liste<Kunde *> kundenliste;
Liste<Mitarbeiter *> mitarbliste;
...
Adresse adr("Rosenweg", "7b", "01234", "Kaff");
Kunde *k = new Kunde("Meier", "Max", adr);
Mitarbeiter *m = new Mitarbeiter("Meier",
        "Max", adr);
...
kundenliste.append(k);
mitarbliste.append(m);
...
k.changeName("Schulze");
m.changeName("Schulze");     // nicht vergessen!
```

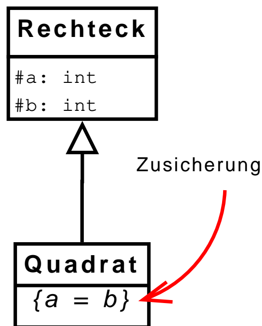
Vererbung kritisch prüfen:

- Rechteck-Quadrat-Problem oder analog
- Ellipse-Kreis-Problem



Ähnliche Frage: Ist eine reelle Zahl eine Komplexe Zahl?

1. Versuch: Ein Quadrat *ist ein* Rechteck.



→ redundante Kanteninformation

Problem:
Wenn wir die Klasse `Rechteck` um eine Methode `resize(int dx, int dy)` erweitern, dann erbt die Klasse `Quadrat` diese Methode und ein Quadrat-Objekt kann zu einem Rechteck verzerrt werden.

Wir müssen alle gefährlichen Methoden von `Rechteck` durch überschreiben ausblenden.

Nach jeder Änderung in `Rechteck` ist zu prüfen, ob auch `Quadrat` zu ändern ist.

trotzdem: Wir können ein `Quadrat` explizit in ein `Rechteck` umwandeln (type cast) und dann die `Rechteck`-Methode `resize` aufrufen.

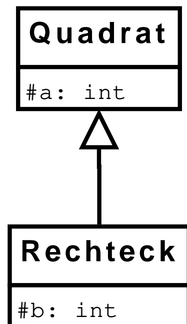
Durch diesen Umweg können wir das `Quadrat` zu einem `Rechteck` verzerren!

Was lernen wir daraus?

Keine Zusicherungen auf Attribute geben, die in der Oberklasse liegen!

Wir müssen bei all unseren Modellierungen das *Liskovsche Substitutionsprinzip* beachten: Objekte der abgeleiteten Klasse können stets an die Stelle von Objekten der Oberklasse treten.

2. Versuch: „technische Vererbung“ oder „Vererbung aus Bequemlichkeit“ (inheritance by convenience)



`Rechteck` ist eine Spezialisierung von `Quadrat`, weil es eine neue Eigenschaft ergänzt: die Länge der anderen Seite.

Problem:

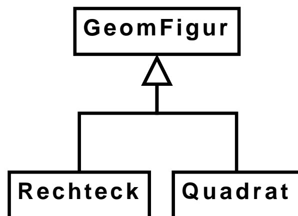
- Modellierung ist nicht intuitiv: Die Klasse verhält sich nicht erwartungskonform.

→ Bei einer Weiterentwicklung sind Fehler vorprogrammiert!

- Geerbte Methoden wie `area()` gelten für `Rechteck` nicht.

→ Der Wartungsaufwand ist groß!

3. Versuch:



Problem: Unsere Klassen enthalten redundanten Code, denn ein `Quadrat` ähnelt einem `Rechteck` sehr.

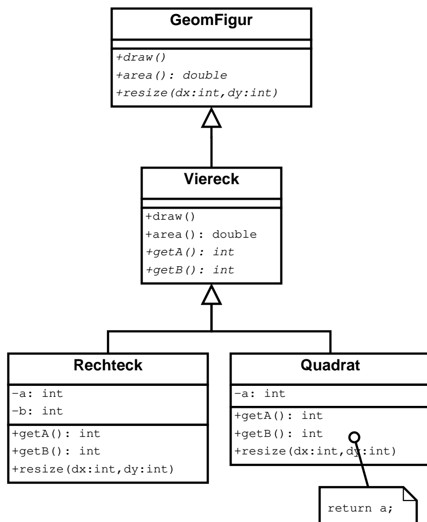
Um den Wartungsaufwand klein zu halten, schreiben wir möglichst wenig redundanten Code, d.h. wir verwenden kein Copy-And-Paste!

Um redundanten Code zu vermeiden, können wir eine spezifische Basisklasse `Viereck` für `Quadrat` und `Rechteck` einführen.

Viereck implementiert die gemeinsamen Methoden draw() und area() über Verwendung der Get-Methoden:

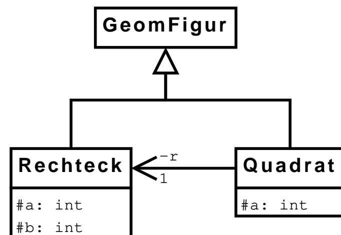
```
double
Viereck::area(void) {
    return getA() *
                getB();
}
```

Damit das funktioniert, muss getB() der Klasse Quadrat den Wert a liefern.



Die Methode resize() muss jeweils in Rechteck und Quadrat implementiert werden.

4. Versuch: Delegation



- `Quadrat` bekommt ein eigenes, privates Attribut `r` vom Typ `Rechteck`
- `Quadrat` ruft alle brauchbaren Methoden von `Rechteck` über `r` auf (simple Einzeiler!)

```
Quadrat::Quadrat(int a) {
    r = Rechteck(a, a);
    this->a = a;
}

double Quadrat::area() {
    return r->area();
}
```

Anmerkungen:

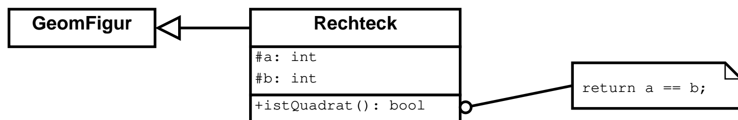
- `Quadrat`-spezifische Methoden werden dort realisiert, wo sie hingehören: innerhalb der Klasse `Quadrat`.
- Gute Lösung, solange die Basisklasse rein abstrakt ist.

Problem:

- Beim Einführen neuer Methoden wie bspw. `move()` in `GeomFigur` müssen wir in `Quadrat` die Methode überschreiben und an `Rechteck` delegieren.

$\rightarrow$ Wartungsaufwand ist groß!

5. Versuch: `Quadrat` als Eigenschaft von `Rechteck`



Problem: Dies führt zu `if/else`-Konstrukten, die wir durch Polymorphie vermeiden wollten!

Beispiel:

```
void Rechteck::resize(int dx, int dy) {
    if (istQuadrat() && dx != dy)
        throw "can not resize";
    ....
}
```

Tut mir leid: Ich habe keine Lösung für das Problem!

Übersicht

Objektorientierung

- Motivation
- Funktionalität einer Klasse erweitern
- Polymorphie
- Abstrakte Klassen
- Klassen im Zusammenspiel
- Modellierung
- *Sequenz-, Aktivitäts- und Zustandsdiagramme*
- Standard Template Library

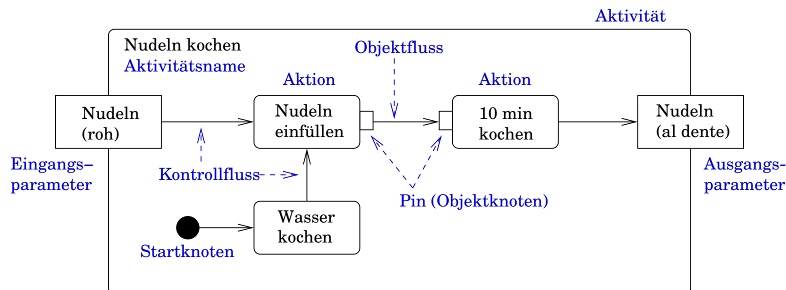
Übersicht

Bisher kennen wir nur das Klassendiagramm, das die Beziehungen zwischen den Klassen in einer Art Bauplan darstellt.

- Ein Aktivitätsdiagramm stellt komplexe Abläufe z.B. eines Anwendungsfalles grafisch dar. Dabei werden Ausnahmen, Varianten, Sprünge und Wiederholungen übersichtlich und verständlich dargestellt.
- Ein Sequenzdiagramm zeigt die einzelnen Botschaften der Objekte, die nötig sind, um einen ausgewählten Ablauf zu realisieren. Hierbei wird in der Regel nicht ein vollständiger Anwendungsfall, sondern nur ein Szenario dargestellt, also ein einzelner Weg innerhalb eines Systemablaufs.
- Ein Zustandsdiagramm zeigt eine Folge von Zuständen, die ein einzelnes Objekt im Laufe seines Lebens annehmen kann und aufgrund welcher Stimuli eine Zustandsänderung eintritt.

Aktivitätsdiagramme

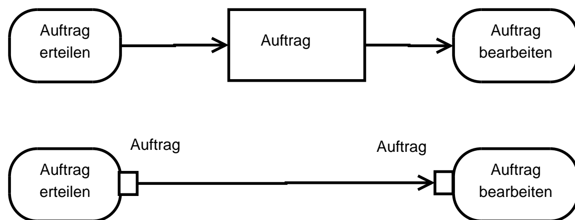
- beschreiben die Abfolgen von Aktionen in einem System
- sind den prozeduralen Flussdiagrammen sehr ähnlich
- haben eine ähnliche Semantik wie Petri-Netze



Eine *Aktivität* besteht aus einer Folge von Aktionen und beschreibt oft einen Anwendungsfall. Eine *Aktion* ist ein einzelner Schritt. Damit eine Aktion startet, müssen alle eingehenden Kontrollflüsse vorliegen.

Aktivitätsdiagramme

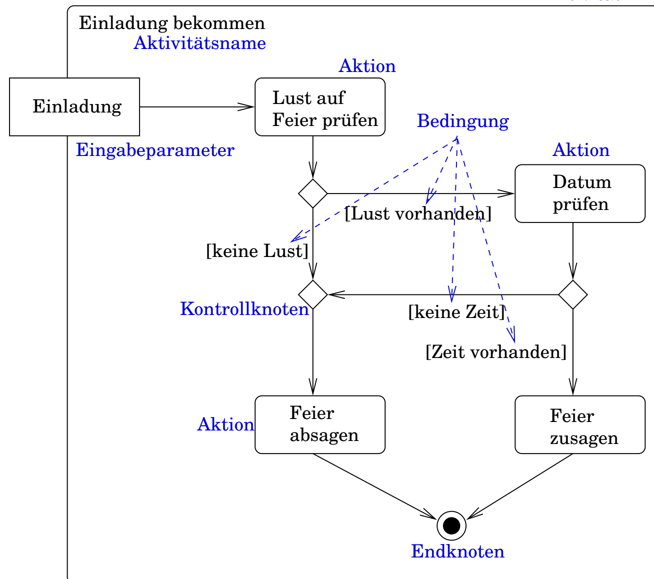
Unterscheide Aktionsknoten und Objektknoten.



Weiterhin unterscheiden wir drei Arten von Kontrollknoten:

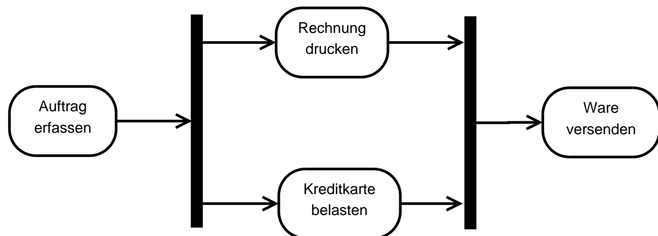
- Entscheidung und Zusammenführung
- Start- und Endknoten
- Splitting und Synchronisation (bei parallelen Abläufen)

Aktivitätsdiagramme

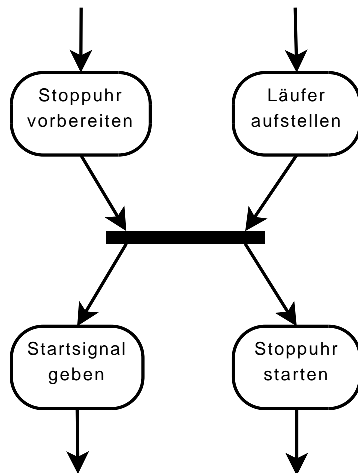


Aktivitätsdiagramme

Parallele Bearbeitung wird mittels Splitting und Synchronisation umgesetzt (Fork/Wait-Symbole). Für parallel laufende Aktivitäten ist es unerheblich, in welcher Reihenfolge sie ablaufen. Sie können zum gleichen Zeitpunkt aber auch nacheinander ausgeführt werden.



Aktivitätsdiagramme



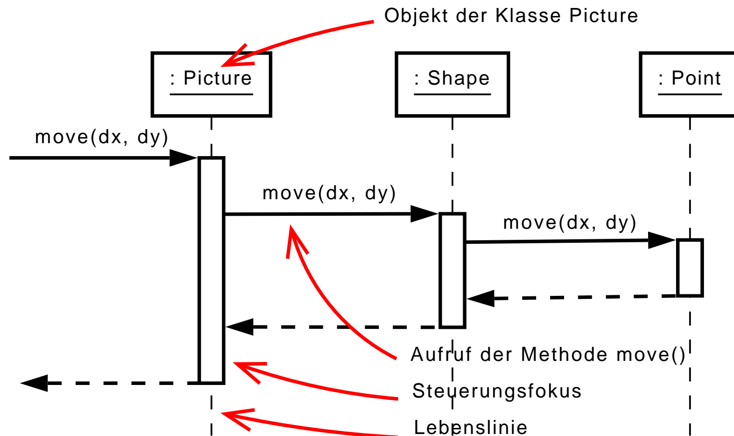
Zunächst müssen die Läufer aufgestellt und die Stoppuhr vorbereitet sein.

Ist beides geschehen, wird zeitgleich das Startsignal gegeben und die Stoppuhr gestartet.

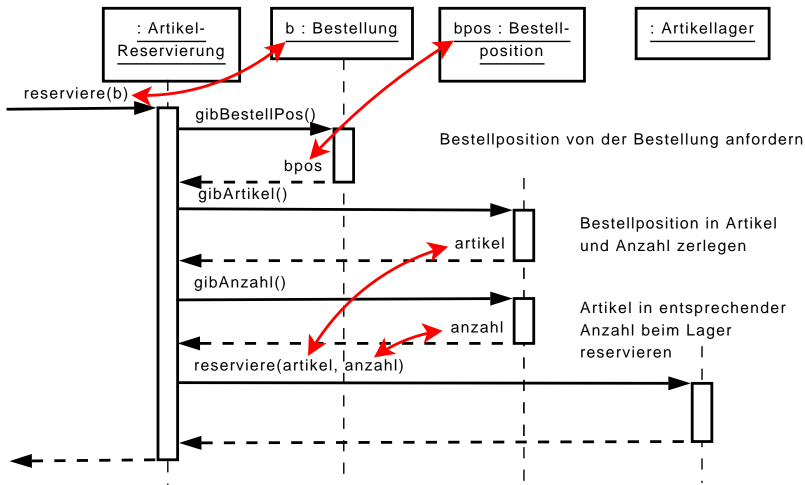
Abläufe werden am Synchronisationspunkt implizit durch AND verbunden.

Andere Verknüpfungen müssen explizit angegeben werden.

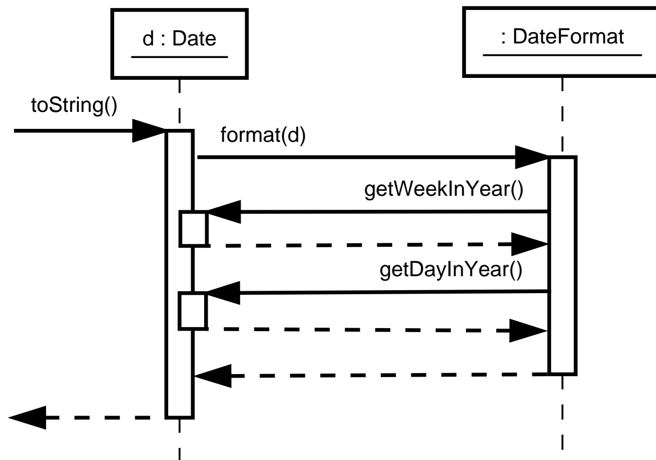
Ein *Sequenzdiagramm* zeigt eine Reihe von Nachrichten, die eine ausgewählte Menge von Objekten in einer zeitlich begrenzten Situation austauscht, wobei der zeitliche Ablauf betont wird.



Sequenzdiagramme



Sequenzdiagramme

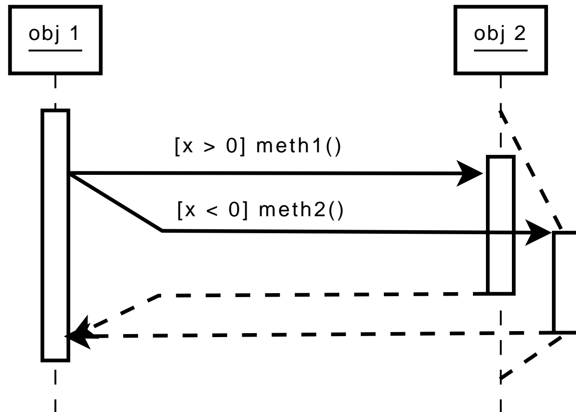


Realisierung des letzten Beispiels:

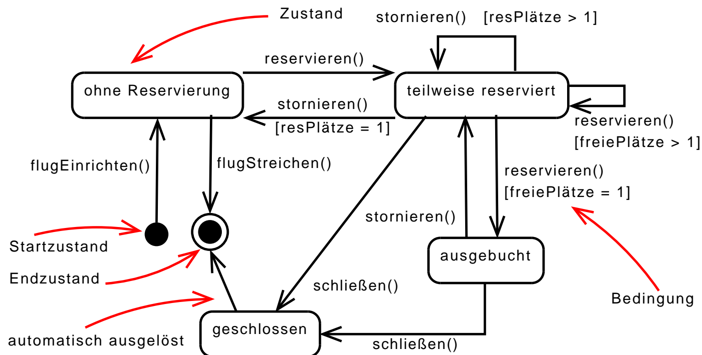
```
class Date {
private:
    int _day, _month, _year;
    DateFormat *_df;

public:
    string toString() {
        return _df->format(this);
    }
    int getWeekInYear();
    int getDayInYear();
    .....
    int getDay();
    int getMonth();
    int getYear();
};
```

```
class DateFormat {
public:
    virtual string format(Date *d) {
        ostringstream os;

        os << d->getDay() << ".";
        os << d->getMonth() << ".";
        os << d->getYear() << endl;

        os << "Kalenderwoche: "
           << d->getWeekInYear() << endl;
        os << "Tag im Jahr: "
           << d->getDayInYear() << endl;

        return os.str();
    }
    .....
};
```

Nachrichten können mit Bedingungen versehen werden.



Ein *Zustandsdiagramm* zeigt eine Folge von Zuständen, die ein einzelnes Objekt im Laufe seines Lebens einnehmen kann und aufgrund welcher Stimuli Zustandsänderungen stattfinden.



Übersicht

Objektorientierung

- Motivation
- Funktionalität einer Klasse erweitern
- Polymorphie
- Abstrakte Klassen
- Klassen im Zusammenspiel
- Modellierung
- Sequenz-, Aktivitäts- und Zustandsdiagramme
- *Standard Template Library*

Motivation

Häufig genutzte Container und Algorithmen werden in der STL (Standard Template Library) bereitgestellt und müssen von uns nicht mehr implementiert werden.

Problem: Algorithmen hängen von der Art und Weise ab, wie die Daten gespeichert werden:

- binäre Suche im Array (sortiert)
- lineare Suche in verketteter Liste

Um unsere Algorithmen unabhängig von der Art der Speicherung zu machen, lernen wir das Iterator-Konzept kennen.

Container:

- sind eine Sammlung von Objekten (geordnet oder ungeordnet)
- verfügen über Methoden zum
  - Einfügen von Objekten: `insert`, `push_back`, `push_front`
  - Löschen von Objekten: `pop_back`, `pop_front`, `erase` und `clear`
  - Suchen von Objekten: `find`
  - Iterieren über die enthaltenen Objekte: `back`, `front`, `begin`, `end`, `rbegin` und `rend`
- oft noch weitere Funktionen wie Sortieren, Vergleichen usw.

Beispiele für

- sequentielle Container: `list`, `vector`, `deque`
- assoziative Container: `set`, `multiset`, `map`, `multimap`
- Algorithmen: `sort`, `binary_search`, `for_each`, `count`, `nth_element` und weitere

Eine gute Übersicht zur Standard Template Library finden Sie bspw. unter:

http://www.cppreference.com
http://www.cplusplus.com

Kanonische Klassen: Die Klassen der Container-Elemente sollen Methoden vorgeben, damit die Container auf den Elementen arbeiten können:

- Konstruktor und Destruktor
- Copy-Konstruktor
- Zuweisungsoperator `operator=`
- Gleichheitsoperator `operator==`
- Kleineroperator `operator<`

In der STL werden die anderen Vergleichsoperatoren aus den gegebenen in `<utility>` definiert.

Beispiel: `>` ist äquivalent zu „nicht kleiner und nicht gleich"

Container für unterschiedliche Arten von Objekten:

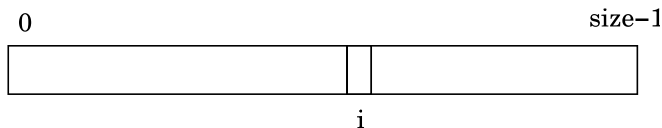
1. Basisklasse für den Container:
    - Für jede neue Objektart, die gespeichert werden soll, wird eine Klasse von der Container-Basisklasse abgeleitet.
    - Nachteil: Hoher Programmieraufwand und sehr ähnliche Lösungen für jede neue Objektart.
2. Basisklasse für die zu speichernden Objekte:
    - Die Container werden für die Basisobjekte formuliert.
    - Neue Objektarten werden durch Vererbung gebildet.
    - Durch dynamische Bindung können auch Objekte des abgeleiteten Typs im Container gespeichert werden.
    - Nachteil: eingeschränkte Typfreiheit

3. Generische Programmierung (Templates):
   - Die Container werden allgemein formuliert.
   - Es werden konkrete Ausprägungen für einzelne Typen von Objekten gebildet.
   - Nachteil: Für jede Ausprägung generiert der Compiler zusätzlichen Objekt-Code und wir können nur Objekte eines Typs und deren Untertypen speichern.

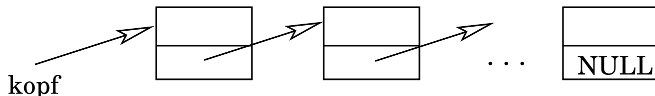
Die STL verwendet Templates für die Container und sehr effiziente Implementierungen der Algorithmen!

Wir können einen Container, in dem Objekte unsortiert gespeichert werden, auf mehrere Arten implementieren: Die Objekte werden

- in einem Array gespeichert



- oder in einer verketteten Liste.



Container mittels unsortiertem Array implementiert:

```
template <typename T>
int MyVector<T>::suchen(const T& x) const {
    for (int i = 0; i < size; i++)
        if (values[i] == x)
            return i;
    return -1;
}
...

MyVector<int> v(100);
...
int i = v.suchen(42);
if (i != -1)
    // gefunden
else // nicht gefunden
```

Container mittels verketteter Liste implementiert:

```
template <typename T>
T * MyList<T>::suchen(const T& x) const {
    for (Elem<T> *p = kopf; p != 0; p = p->next)
        if (p->value == x)
          return &p->value;  // Kopie liefern?
    return 0;
}
...

MyList<int> l(100);
...
int *p = l.suchen(42);
if (p != 0)
    // gefunden
else // nicht gefunden
```

Vektor vs. Liste

Problem:

- Beide Klassen besitzen eine eigene Elementfunktion zum Suchen, die speziell auf die Klasse zugeschnitten ist.
- Bei n Klassen und m Elementfunktionen müssen wir also $n \cdot m$ Elementfunktionen definieren.

Lösung: Standardisierung der Suchfunktion.

- Wie wird der Suchparameter an die Funktion übergeben? (Referenz, Zeiger, Objekt)
- Welchen Rückgabetyp soll die Funktion haben?
- Was wird zurückgegeben, wenn das Gesuchte nicht gefunden wird?

Vektor vs. Liste

Angleichen der Suchfunktionen des Vektors und der Liste:

```
T * MyVector<T>::suchen(const T& x) const {
    T *p = values;          // start
    while (p != values + size   // nicht am Ende
            && *p != x)         // nicht gefunden
        p++;                // naechstes Element
    return p;
}

T * MyList<T>::suchen(const T& x) const {
    Elem<T> *p = kopf;  // start
    while (p != 0               // nicht am Ende
            && p->value != x)   // nicht gefunden
        p = p->next;        // naechstes Element
    return ????;
}
```

Vektor vs. Liste

Die `while`-Schleife bricht ab, sobald das Element gefunden wurde oder das „Ende des Containers“ erreicht wurde.

Nichtgefunden bedeutet: Rückgabeposition ist „Ende des Containers“.

Parametrisierte Funktion zum Suchen in `MyVector`:

```
template <typename T>
T * suchen(T *start, T *ende, T x) {
    T *p = start;

    while (p != ende && *p != x)
        p++;

    return p;
}
```

Vektor vs. Liste

Aufruf der Suchfunktion im Vector-Beispiel:

```
MyVector<int> v(100);
...
int *z = suchen(v.start(), v.ende(), 42);
if (z != v.ende())
    // gefunden
else // nicht gefunden
```

Die Klasse `MyVector` muss die Funktionen `start()` und `ende()` bereitstellen!

Frage: Warum können wir diese parametrisierte Suchfunktion nicht für unsere Listenklasse nutzen?

Vektor vs. Liste

Antwort:

- Der Inhaltsoperator funktioniert nicht: In der verketteten Liste müsste es `p->value != x` heißen anstatt `*p != x`.
- Die lokale Variable `start` ist nicht vom Typ `T *` sondern vom Typ `Elem<T> *`.
- Ebenso ist `ende` nicht vom Typ `T *` sondern vom Typ `Elem<T> *`.
- Das nachfolgende Element in einer verketteten Liste ergibt sich nicht aus einem Zeiger auf ein Feldelement +1, sondern der Zeiger auf das nachfolgende Element ist in dem Feldelement selbst enthalten.

Identifiziere die Unterschiede und kapsele diese!

```
template <typename Iter, typename T>
Iter suchen(Iter start, Iter ende, const T x) {
    while (start != ende && *start != x)
        start++;

    return start;
}
```

Welche Operationen werden beim Iterator benötigt?

Implizite Annahmen über den Iterator:

- Die Parameter `start` und `ende` vom Typ `Iterator` werden als Wert an die Suchfunktion übergeben, es muss also ein Copy-Konstruktor implementiert sein.
- In der Abbruchbedingung der `while`-Schleife
  - werden zwei Iteratoren auf Ungleichheit geprüft und
  - der Iterator `start` wird dereferenziert, um das aktuelle Element mit dem gesuchten Element `x` zu vergleichen.
- Im Rumpf der Schleife wird der Iterator `start` inkrementiert.
- Iterator `start` wird per Copy-Konstruktor zurückgegeben.

Iterator für einen einfachen Vektor-Typ

```
template <typename T>
class MyVector {
    int _last, _size;
    T *_values;

public:
    MyVector(int size = 10) {
        _last = 0;
        _size = size;
        _values = new T[size];
    }
    void insert(T val) {
        if (_last == _size)
            throw "no space left";
        _values[_last] = val;
        _last += 1;
    }
    .....
```

Iterator für einen einfachen Vektor-Typ

```
class Iter {
private:
    T *_cursor;
public:
    Iter(T *cursor = 0) {  // Copy-Konstr.?
        _cursor = cursor;
    }
    bool operator!=(Iter iter) const {
        return _cursor != iter._cursor;
    }
    T& operator*() const {
        return *_cursor;
    }
    Iter operator++(int) {  // postfix
        Iter it = *this;
        _cursor++;
        return it;
    }
}; // Ende des Iterators
```

```
    // weiter mit MyVector

    Iter start() {
        return _values;  // Rueckgabe-Typ ok?
    }

    Iter ende() {
        return _values + _last;
    }

    Iter suchen(Iter start, Iter ende, T x) {
        while (start != ende && *start != x)
            start++;

        return start;
    }
}; // Ende der Klasse MyVector
```

Iterator für einen einfachen Vektor-Typ

```
int main(void) {
    MyVector<int> v;

    for (int i = 1; i < 10; i++)
        v.insert(i);

    MyVector<int>::Iter iter;

    iter = v.suchen(v.start(), v.ende(), 6);
    if (iter != v.ende())
        cout << "6 gefunden\n";
    else cout << "6 nicht gefunden\n";
}
```

Iterator für einen einfachen Listen-Typ

```
template <typename T>
class MyList {
private:
    struct Elem {
        T value;
        Elem *next;

        Elem(const T& val) {
            value = val;
            next = 0;
        }
    } *_head, *_tail;

public:
    MyList() {
        _head = 0;
        _tail = 0;
    }
```

Iterator für einen einfachen Listen-Typ

```
void append(T val) {
    Elem *e = new Elem(val);

    if (_tail == 0)
        _head = e;
    else _tail->next = e;
    _tail = e;
}
```

Iterator für einen einfachen Listen-Typ

```
class Iter {
private:
    Elem *_cursor;
public:
    Iter(Elem *cursor = 0) { // Copy-Konst.?
        _cursor = cursor;
    }
    bool operator!=(Iter iter) const {
        return _cursor != iter._cursor;
    }
    T& operator*() const {
        return _cursor->value;
    }
    Iter operator++(int) {  // postfix
        Iter it = *this;
        _cursor = _cursor->next;
        return it;
    }
}; // Ende des Iterators
```

Iterator für einen einfachen Listen-Typ

```
    // weiter mit der Klasse MyList

    Iter start() {
        return _head;    // Rueckgabe-Typ ok?
    }

    Iter ende() {
        return 0;
    }

    Iter suchen(Iter start, Iter ende, T x) {
        while (start != ende && *start != x)
            start++;

        return start;
    }
}; // Ende der Klasse MyList
```

Iterator für einen einfachen Listen-Typ

```
int main(void) {
    MyList<int> l;

    for (int j = 6; j < 26; j++)
        l.append(j);

    MyList<int>::Iter iter;

    iter = l.suchen(l.start(), l.ende(), 12);
    if (iter != l.ende())
        cout << "12 gefunden\n";
    else cout << "12 nicht gefunden\n";
}
```

Auf den folgenden Folien finden Sie einige Programme, die exemplarisch die Anwendung von Containern und den Umgang mit Iteratoren zeigen:

- `set` und `multiset`
- `vector`
- `map` und `multimap`
- `iterator` und `reverse_iterator`

Set

```
#include <iostream>
#include <set>
#include <cstdlib>
using namespace std;

#define MAX 30

int main(void) {
    set<char> s;
    multiset<char> ms;

    for (int i = 0; i < MAX; i++) {
        char r = 'a' + rand() % 26;
        cout << r;
        s.insert(r);
        ms.insert(r);
    }
    cout << endl;
```

```
    set<char>::iterator iter;
    for (iter = s.begin(); iter != s.end();
            iter++)
        cout << *iter;
    cout << endl;

    set<char>::reverse_iterator riter;
    for (riter = s.rbegin(); riter != s.rend();
            riter++)
        cout << *riter;
    cout << endl;

    multiset<char>::iterator iter2;
    for (iter2 = ms.begin(); iter2 != ms.end();
            iter2++)
        cout << *iter2;
    cout << endl;
}
```

Ausgabe:

```
nwlrbbmqbhcdarzowkkyhiddqscdxr
abcdhiklmnoqrswxyz
zyxwsrqonmlkihdcba
abbbccddddhhikklmnoqqrrrswwxyz
```

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
using namespace std;

int main(void) {
    vector<int> v;

    for (int i = 1; i < 20; i++)
        v.push_back(i);

    vector<int>::iterator it;
    for (it = v.begin(); it != v.end(); it++) {
        if (it != v.begin())
            cout << ",";
        cout << *it;
    }
    cout << endl;
```

```
    vector<int>::reverse_iterator rit;
    for (rit = v.rbegin(); rit != v.rend();
            rit++) {
        if (rit != v.rbegin())
            cout << ",";
        cout << *rit;
    }
    cout << endl;

    it = find(v.begin(), v.end(), 10);
    v.erase(it);
    for (it = v.begin(); it != v.end(); it++) {
        if (it != v.begin())
            cout << ",";
        cout << *it;
    }
    cout << endl;
}
```

Ausgabe:

```
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19
19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1
1,2,3,4,5,6,7,8,9,11,12,13,14,15,16,17,18,19
```

Multimap

```
#include <iostream>
#include <string>
#include <map>
#include <cstdlib>
using namespace std;

string randomString() {
    string res;

    for (int i = 0; i < 4; i++)
        res += rand() % 26 + 'a';

    return res;
}
```

Multimap

```
int main(void) {
    map<int, string> m1;
    multimap<int, string> m2;

    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 3; j++) {
            string s = randomString();
            m1.insert(pair<int, string>(i, s));
            m2.insert(pair<int, string>(i, s));
        }

    map<int, string>::iterator it1;
    for (it1 = m1.begin(); it1 != m1.end();
            it1++)
        cout << (*it1).first << ": "
             << (*it1).second << endl;
    ...
```

```
    multimap<int, string>::iterator it2;
    for (it2 = m2.begin(); it2 != m2.end();
            it2++)
        cout << (*it2).first << ": "
            << (*it2).second << endl;

    return 0;
}
```

Ausgabe:

```
0: nwlr
1: arzo
2: qscd
3: xsjy
```

Ausgabe:

```
0: nwlr
0: bbmq
0: bhcd
1: arzo
1: wkky
1: hidd
2: qscd
2: xrjm
2: owfr
3: xsjy
3: bldb
3: efsa
```

Functor

In C haben wir beim Aufruf von Funktionen wie `bsearch` oder `qsort` Zeiger auf Funktionen übergeben. Diese Funktionen implementieren eine Vergleichsoperation für zwei Feldelemente.

Der Aufruf der Vergleichsfunktion erfolgt aus `bsearch` bzw. aus `qsort` heraus *zurück* in das Anwendungsprogramm, deshalb wird diese Technik als *Callback* bezeichnet.

In C++ tritt an die Stelle von Zeigern auf Funktionen das allgemeinere Konzept des Funktionsobjekts, welches auch als Functor bezeichnet wird:

> Ein Functor ist ein Objekt, welches `operator()` definiert.

Functor

```
struct Limiter {
    int lowerBound;
    int upperBound;

    Limiter(int l, int u) {
        lowerBound = l;
        upperBound = u;
    }

    void operator()(int& value) {
        if (value < lowerBound)
            value = lowerBound;
        else if (value > upperBound)
            value = upperBound;
    }
};
```

Beispiel aus `http://www.cpp-tutor.de/default.html`

Functor

```
int main(void) {
    vector<int> v;
    vector<int>::iterator it;
    int lower, upper;

    cout << "untere Grenze: ";
    cin >> lower;
    cout << "obere Grenze: ";
    cin >> upper;

    for (int i = -5; i <= 5; i++)
        v.push_back(i);

    for (it = v.begin(); it != v.end(); it++)
        cout << *it << endl;
    cout << endl;
    ...
```

```
    for_each(v.begin(), v.end(),
             Limiter(lower, upper));

    for (it = v.begin(); it != v.end(); it++)
        cout << *it << endl;

    return 0;
}
```

Ausgabe für `lower = -2` und `upper = 2`:

```
vorher:
    -5 -4 -3 -2 -1  0  1  2  3  4  5

nachher:
    -2 -2 -2 -2 -1  0  1  2  2  2  2
```

Functor

Anmerkungen:

- Der Functor erhält das Element per Referenz und kann deshalb dessen Wert ändern.
- Innerhalb der `for_each`-Parameterklammer wird das Funktionsobjekt definiert, indem der Konstruktor `Limiter()` mit der unteren Grenze `lower` und der oberen Grenze `upper` aufgerufen wird.
- Der überladene Operator `()` wird intern von `for_each` aufgerufen.
- Das obige Beispiel wäre mit einer einfachen Callback-Funktion nicht realisierbar, da die Werte `lower` und `upper` in der Funktion fest kodiert sein müssten! Oder?

Functor

Doch. Das obige Beispiel ließe sich auch mit einer „normalen“ Callback-Funktion realisieren, aber nicht so elegant:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int lowerBound;      // globale
int upperBound;      // Variablen

void limiter(int& val) {
    if (val < lowerBound)
        val = lowerBound;
    else if (val > upperBound)
        val = upperBound;
}
...
```

Functor

```
int main(void) {
    vector<int> v;
    vector<int>::iterator it;

    cout << "untere Grenze: ";
    cin >> lowerBound; // schreibt in glob. Var.
    cout << "obere Grenze: ";
    cin >> upperBound; // schreibt in glob. Var.

    for (int i = -5; i <= 5; i++)
        v.push_back(i);

    for_each(v.begin(), v.end(), limiter);

    for (it = v.begin(); it != v.end(); it++)
        cout << *it << endl;
}
```

Functor

Auch wenn in einem Container keine Objekte sondern Zeiger auf Objekte gespeichert werden, sind Functoren sinnvoll einsetzbar.

Die Algorithmen `sort`, `min_element` oder `max_element` benötigen Vergleichsoperatoren, ebenso der Container `priority_queue`. Da die Vergleichsoperatoren aber auf Objekten definiert sind, würden nur Zeigerwerte, aber nicht die Inhalte verglichen. Siehe dazu das folgende Beispiel.

Beispiel

```
#include <algorithm>
#include <vector>
...
int main(void) {
    vector<Rational *> v;

    v.push_back(new Rational(1, 2));
    v.push_back(new Rational(1, 3));
    v.push_back(new Rational(1, 4));
    v.push_back(new Rational(1, 5));
    v.push_back(new Rational(1, 6));

    Rational *min =
            *min_element(v.begin(), v.end());
    cout << "Minimum: " << *min << endl;
```

Überladen von Operatoren

```
    Rational *max =
            *max_element(v.begin(), v.end());
    cout << "Maximum: " << *max << endl;

    vector<Rational *>::iterator it;

    cout << " vor dem Sortieren:\n";
    for (it = v.begin(); it != v.end(); it++)
        cout << **it << endl;

    sort(v.begin(), v.end());

    cout << "nach dem Sortieren:\n";
    for (it = v.begin(); it != v.end(); it++)
        cout << **it << endl;
}
```

Ausgabe:

```
Minimum: (1/2)
Maximum: (1/6)

 vor dem Sortieren:        nach dem Sortieren:
 (1/2)                     (1/2)
 (1/3)                     (1/4)
 (1/4)                     (1/3)
 (1/5)                     (1/5)
 (1/6)                     (1/6)
```

Sieht nicht ganz richtig aus, oder? Hier wurden Zeigerwerte, aber nicht deren Inhalt verglichen!

Korrekt wäre:

```
class RationalComparer {
public:
    bool operator()(Rational *a, Rational *b) {
        return *a < *b;
    }
} rcmp;

int main(void) {
    vector<Rational *> v;

    v.push_back(new Rational(1, 2));
    v.push_back(new Rational(1, 3));
    v.push_back(new Rational(1, 4));
    v.push_back(new Rational(1, 5));
    v.push_back(new Rational(1, 6));
```

Überladen von Operatoren

```
    Rational *min =
        *min_element(v.begin(), v.end(), rcmp);
    cout << "Minimum: " << *min << endl;

    Rational *max =
        *max_element(v.begin(), v.end(), rcmp);
    cout << "Maximum: " << *max << endl;

    vector<Rational *>::iterator it;
    cout << " vor dem Sortieren:\n";
    for (it = v.begin(); it != v.end(); it++)
        cout << **it << endl;
    sort(v.begin(), v.end(), rcmp);

    cout << "nach dem Sortieren:\n";
    for (it = v.begin(); it != v.end(); it++)
        cout << **it << endl;
}
```

Predicates

Predicates sind Funktionen oder Functors die einen bool-Wert zurückliefern.

- Predicates steuern in Abhängigkeit von einer Bedingung eine Aktion in den Algorithmen der STL.
- Beispiel: `remove_if()` entfernt Elemente, die einer bestimmten Bedingung genügen, aus dem Container.

  Die Bedingung ist als Predicate zu definieren. Liefert das Predicate `true` zurück, so wird das an das Predicate übergebene Element aus dem Container entfernt.

Übersicht

- Strategie (Strategy)
- Abstrakte Fabrik (Abstract Factory)
- Einzelstück (Singleton)
- Dekorierer (Decorator)
- Zustand (State)
- Beobachter (Observer)
- Kompositum (Composite)
- Model-View-Controller

Motivation

Der Entwurf objektorientierter Software ist schwer!

Aber noch viel schwerer ist der Entwurf wiederverwendbarer objektorientierter Software, denn

- er muss spezifischen Anforderungen genügen,
- er muss allgemein genug sein, um zukünftigen Problemen und Anforderungen zu begegnen, und
- wir wollen eine Revision von Entwürfen vermeiden bzw. minimieren.

$\rightarrow$ Vor der Fertigstellung eines Entwurfs versuchen wir, ihn wiederzuverwenden; eventuell ist dazu eine Änderung des Entwurfs nötig.

Motivation

Erfahrene Entwickler erstellen gute Entwürfe.

Was wissen die, was unerfahrene nicht wissen?

- Experten vermeiden es, jedes Problem von Grund auf neu anzugehen.
- Sie verwenden Lösungen wieder, die bereits zuvor erfolgreich eingesetzt wurden: *Entwurfsmuster*.

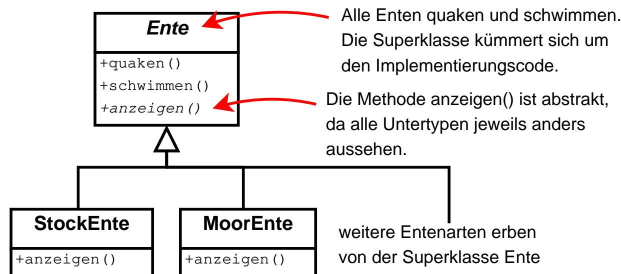
Eine gute, verständliche Einführung in das Thema gibt das Buch von Eric Freeman und Elisabeth Freeman: Entwurfsmuster von Kopf bis Fuß. O'Reilly-Verlag.

Der Abschnitt über das Entwurfsmuster Strategy ist diesem Buch entnommen.

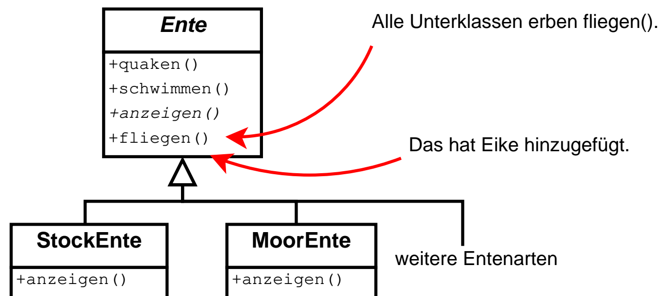
Übersicht

- *Strategy*
- Abstract Factory
- Singleton
- Decorator
- State
- Observer
- Composite
- Model-View-Controller

- Eike arbeitet für ein Unternehmen, dass das höchst erfolgreiche Ententeich-Simulationsspiel `SimEnte` erstellt.
- Das Spiel kann zeigen, wie eine Vielzahl von Entenarten umherschwimmt und Quak-Geräusche von sich gibt.
- Die Spiele-Entwickler haben klassische OO-Techniken verwendet:



- Im letzten Quartal geriet das Unternehmen unter erheblichen Druck durch Konkurrenten.
- Die Unternehmensleitung beschließt beim Golfen, dass sie etwas richtig Beeindruckendes brauchen, was sie auf der Aktionärsversammlung nächste Woche präsentieren können.
- Eikes Manager sagt ihnen, dass fliegende Enten für Eike kein Problem sind, denn er ist ja OO-Programmierer, und so schwer kann das doch nicht sein.
- Eike überlegt: „Ich füge der Superklasse `Ente` einfach eine `fliegen()`-Methode hinzu, die dann von allen Entenarten geerbt wird.“

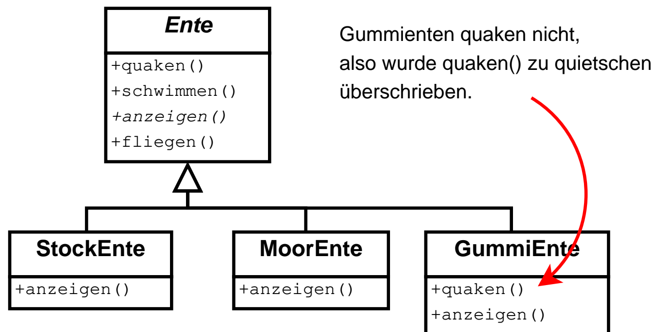


- Ein Anruf von der Aktionärsversammlung: „Eike, ich habe gerade eine Demo vorgeführt, auf der Gummienten über den Bildschirm flogen! Soll das ein Witz sein?“
- Was ist passiert? Was ging schief?

- Eike hat nicht daran gedacht, dass nicht alle Unterklassen von `Ente` fliegen dürfen.

→ Eine lokale Änderung des Codes führte zu einem globalen Nebeneffekt! (fliegende Gummiente)

- Eike denkt: „Na, gut. Mein Entwurf hat einen kleinen Haken. Warum nennen sie es nicht einfach Feature? Sieht doch süß aus.“



Eike denkt über Vererbung nach:

- „Ich könnte die Methode `fliegen()` überschreiben, so wie ich das mit der Methode `quaken()` gemacht habe.“
- „Aber was passiert, wenn wir hölzerne Lockenten hinzufügen, die auch nicht quaken oder fliegen?“

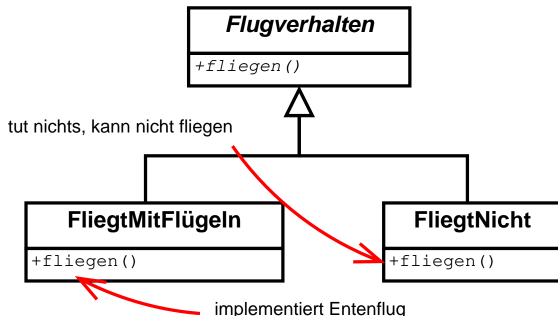
- Die Unternehmensleitung verkündet: „Das Produkt soll jetzt alle 6 Monate aktualisiert werden“. Genaueres weiß man noch nicht.
- Eike weiß jetzt, dass er jedes Mal, wenn dem Programm eine neue Entenklasse hinzugefügt wird, die Methoden `fliegen()` und `quaken()` unter die Lupe nehmen und ggf. überschreiben muss.
- Eine Konstante bei der Software-Entwicklung: Veränderung!

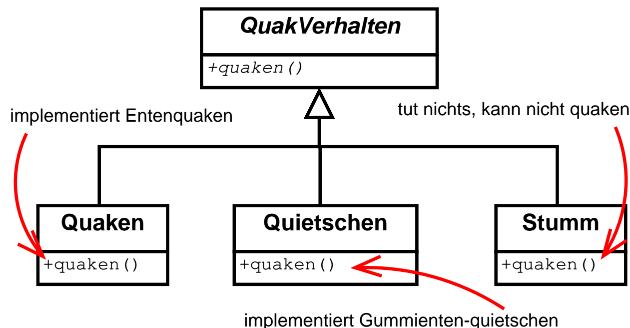
  Egal, wie gut Sie Ihre Anwendung entworfen haben, sie muss im Laufe der Zeit wachsen und sich wandeln, sonst ist sie tot.

Entwurfsentscheidung:

> Identifizieren Sie die Aspekte Ihrer Anwendung, die sich ändern können, und trennen Sie sie von denen, die konstant bleiben.

`fliegen()` und `quaken()` sind die Teile der Klasse Ente, die über die Enten hinweg variieren.





Bei diesem Entwurf können andere Ententypen die Flug- und Quakverhalten wiederverwenden.

Wir können weiteres Verhalten hinzufügen, ohne auch nur eine unserer bestehenden Verhaltensklassen zu ändern.

Implementierung: mittels Delegation!

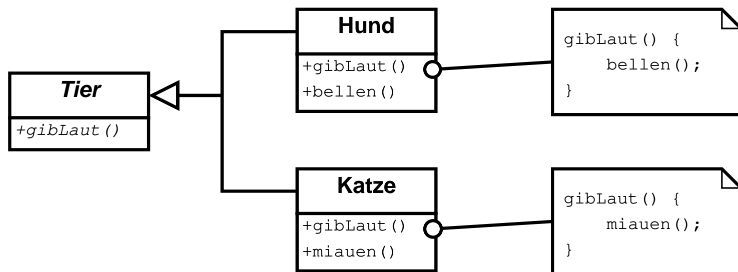
Programmieren Sie auf eine Schnittstelle, nicht auf eine Implementierung.

Auf eine Implementierung programmieren wäre:

```
Hund h = new Hund();
h->bellen();
```

Auf eine Schnittstelle/Supertyp programmieren wäre:

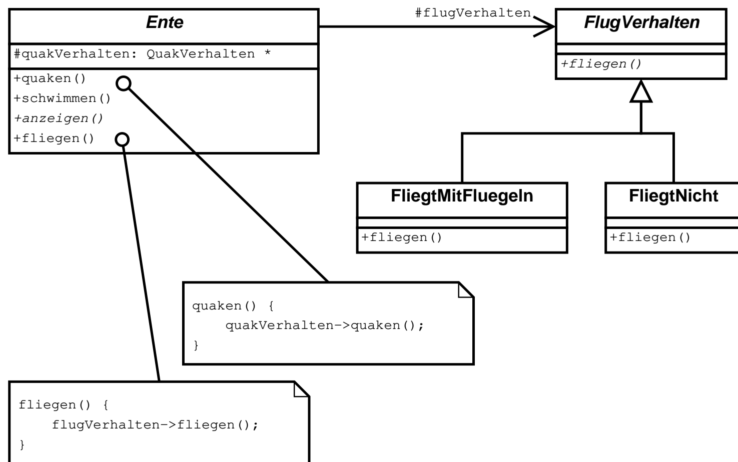
```
Tier t = new Hund();
t->gibLaut();
```



Noch besser wäre es, die Erzeugung des Untertyps nicht im Code festzuschreiben: (siehe abstrakte Fabrik)

```
Tier t = holeTier();
t->gibLaut();
```

Strategy



```
class Ente {
protected:
    QuakVerhalten *quakVerhalten;
    ...
public:
    void quaken(void) {
        quakVerhalten->quaken();
    }
    ...
};
class StockEnte: public Ente {
public:
    StockEnte() {
        quakVerhalten = new Quaken();
        flugVerhalten = new FliegtMitFluegeln();
    }
    ...
};
```

Was lernen wir daraus?

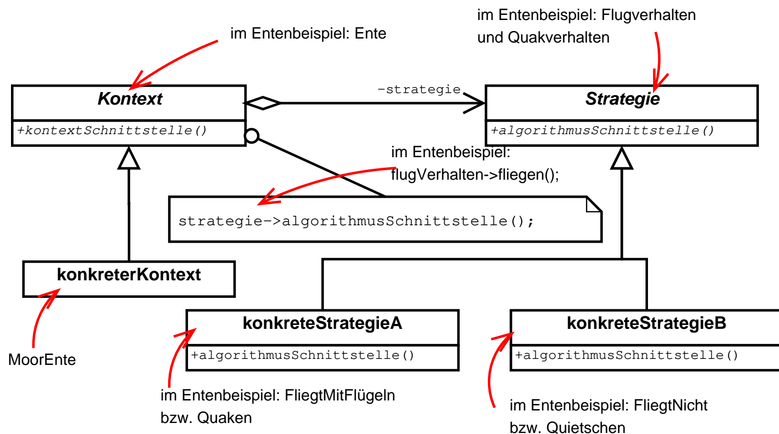
Ziehen Sie Komposition der Vererbung vor.

Herzlichen Glückwunsch zu Ihrem ersten Muster!

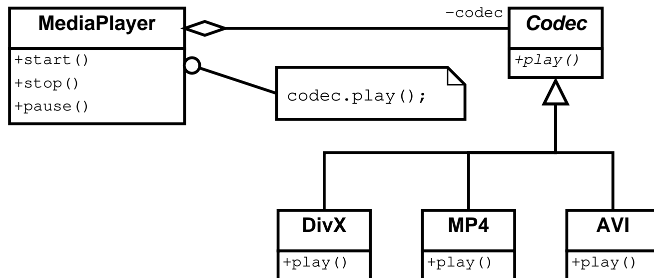
Strategy: Definiere eine Familie von Algorithmen, kapsele jeden einzelnen und mache sie austauschbar.

Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren.

Strategy



Hier eine Anwendung des Strategie-Musters für diejenigen, denen das Entenbeispiel nicht praktisch genug ist.



Übersicht

- Strategy
- *Abstract Factory*
- Singleton
- Decorator
- State
- Observer
- Composite
- Model-View-Controller

Abstract Factory

Motivation: Wir haben ein Software-Produkt erstellt, das an mehreren Stellen an die Bedürfnisse unserer einzelnen Kunden angepasst werden muss:

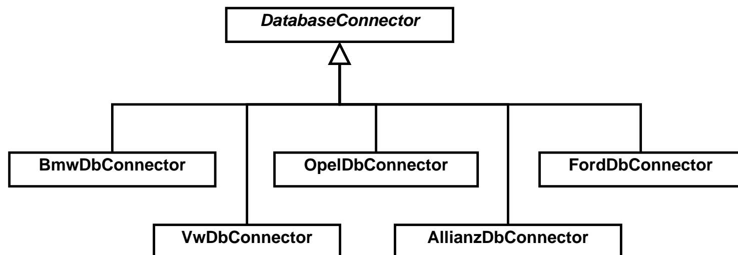
- Verbindung zur Datenbank → `DatabaseConnector`
- Format des Loggings → `Logger`
- Login-Prozess → `Authenticator`
- Verfahren zur Lastbalancierung → `LoadBalancer`

Obige Klassen sind Supertypen, die die Schnittstelle festlegen.

Für unsere Kunden BMW, Ford, Opel, VW, Deutsche Bank, Allianz, Provinzial und Ergo haben wir jeweils spezielle Subklassen implementiert.

Abstract Factory

- Erzeuge für jeden Kunden eigene Subklassen obiger abstrakter Supertypen und
- plaziere möglichst viel gemeinsamen Code in den Basisklassen.



- Große Programme werden bspw. durch Konfigurationsdateien gesteuert: ini-Datei, properties-Datei, ...
- Dort kann der Kunde hinterlegt werden: `Kunde=Ergo`
- In den einzelnen Klassen unserer Software könnte dann stehen:

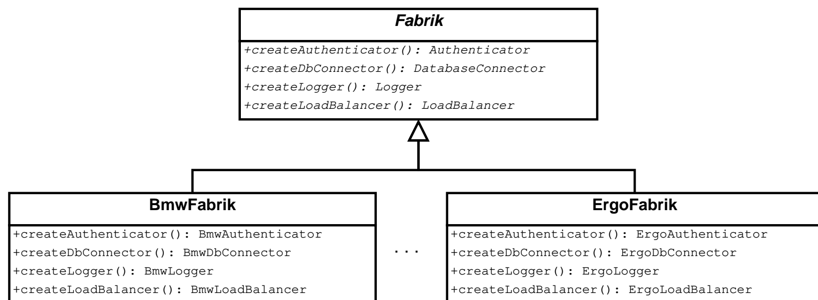
```
DatabaseConnector *dbCon;

if (kunde == "BMW")
    dbCon = new BmwDbConnector();
else if (kunde == "Ford")
    dbCon = new FordDbConnector();
.....
else if (kunde == "Provinzial")
    dbCon = new ProvinzialDbConnector();
else dbCon = new DefaultDbConnector();

dbCon->connect();
```

- Guter Entwurf?

Abstract Factory

Problem: Kommt ein neuer Kunde hinzu, muss an vielen Stellen im Programm der Code ergänzt werden!

- Verlagere die Erzeugung der Objekte in eine eigene Klasse, in die Fabrik.
- Erstelle für jeden Kunden eine eigene Fabrik, die alle benötigten Objekte erzeugen kann.



Abstract Factory

Vorteile: Kommt ein neuer Kunde hinzu, muss nur

- eine neue Fabrik erstellt werden.
- für jede Basisklasse (`DatabaseConnector`, `Authenticator`, `Logger` usw.) eine abgeleitete Klasse erstellt oder die Default-Klasse verwendet werden.
- eine einzige Programmstelle geändert werden (bei Java mit Reflection-API nicht einmal das).

Durch die abstrakte Fabrik wird Konsistenz sichergestellt: An allen Stellen des Programms werden die Komponenten desselben Kunden verwendet.

Abstract Factory

erstellen der Fabrik:

```
Fabrik *fabrik;          // globale Variable!
if (kunde == "BMW")
    fabrik = new BmwFabrik(...);
else if (kunde == "Opel")
    fabrik = new OpelFabrik(...);
...
else if (kunde == "Allianz")
    fabrik = new AllianzFabrik(...);
else fabrik = new DefaultFabrik(...);
```

aufrufen einer `create`-Methode:

```
DatabaseConnector *dbCon;
dbCon = fabrik->createDbConnector(...);
dbCon->connect(...);
```

Abstract Factory

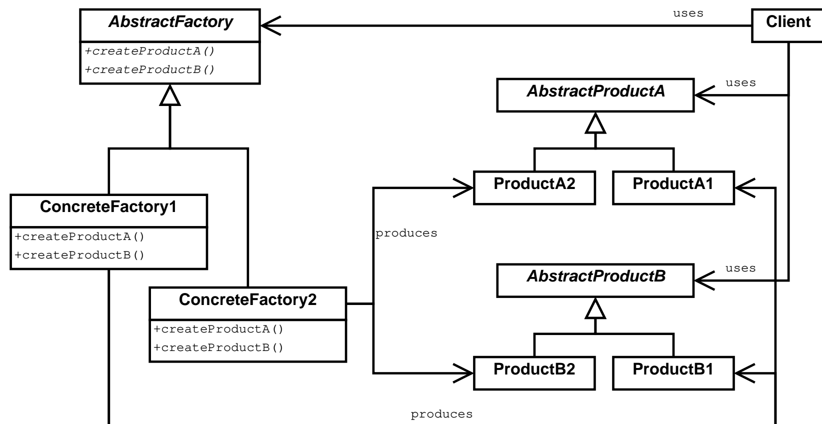
erstellen der Fabrik in Java mit Reflection-API:

```
Fabrik fabrik;
try {
    fabrik = Class.forName(
        System.getProperty("Fabrikklasse"));
} catch (ClassNotFoundException e) {
    e.printStackTrace();
    fabrik = new DefaultFabrik();
}
```

in Properties-Datei:

```
...
Fabrikklasse = de.mycompany.fabrik.BmwFabrik
...
```

Abstract Factory



Abstract Factory

Nachteil:

- Die Fabrik muss an vielen Stellen, also in vielen Klassen bekanntgegeben werden.

  In C++ können wir mit globalen Variablen arbeiten, in anderen Programmiersprachen gibt es keine globalen Variablen.

Frage: Wie können wir sicherstellen, dass es

- nur eine Fabrik gibt?!
- nur einen `DatabaseConnector` gibt?!

→ Entwurfsmuster Singleton

Übersicht

- Strategy
- Abstract Factory
- *Singleton*
- Decorator
- State
- Observer
- Composite
- Model-View-Controller

Singleton

Stelle sicher, dass zu einer Klasse nur genau ein Objekt angelegt wird!

Anwendungen:

- Es gibt in einem System möglicherweise viele
  - Drucker, aber nur einen Druckerspooler.
  - Threads, aber nur einen Thread-Pool.
- Es gibt im System nur
  - eine Datenbankverbindung.
  - ein Objekt für das Logging.
  - einen Cache.
  - ein Objekt `DatumFormatDE`.

Singleton

Idee:

- Damit wir keine Objekte einer Klasse erzeugen können, muss der Konstruktor der Klasse `private` sein.

  In C++ müssen wir zusätzlich den Zuweisungsoperator und den Copy-Konstruktor ausschalten.
- Die statische Methode `exemplar()` liefert das einzige Objekt der Klasse.

Implementierung:

```
class BmwFabrik: public Fabrik {
private:
    BmwFabrik();          // Konstruktor !!!
    static BmwFabrik *fabrik;
    ...

    // Zuweisungsoperator ausschalten
    BmwFabrik & operator=(BmwFabrik &);

    // Copy-Konstruktor ausschalten
    BmwFabrik(const BmwFabrik &);

public:
    static BmwFabrik *exemplar();   // !!!
    ...
};
```

Singleton

```
BmwFabrik * BmwFabrik::fabrik = 0;

BmwFabrik * BmwFabrik::exemplar() {
    if (BmwFabrik::fabrik == 0)
        BmwFabrik::fabrik = new BmwFabrik(...);

    return BmwFabrik::fabrik;
}
```

Dadurch ist zwar sichergestellt, dass es nur eine einzige `BmwFabrik` gibt, aber unser Ausgangsproblem ist so noch nicht gelöst: Es kann immer noch verschiedene Fabriken geben.

- eine Stelle: `fabrik = OpelFabrik::exemplar();`
- andere Stelle: `fabrik = VwFabrik::exemplar();`

→ keine Konsistenzsicherung!

Die globale Variable `fabrik` kann beliebig überschrieben werden!

- Verlagere die globale Variable `fabrik` als Attribut in die Oberklasse `Fabrik`.
- Wandle die Oberklasse `Fabrik` in ein Singleton.

```
class Fabrik {
private:
   static Fabrik *fabrik;

protected:
   Fabrik();                   // warum nicht private?
   Fabrik & operator=(Fabrik &);   // Zuw.op. aus
   Fabrik(const Fabrik &);     // Copy-Konstr. aus

public:
   static Fabrik *exemplar();
};
```

```
Fabrik * Fabrik::fabrik = 0;

Fabrik * Fabrik::exemplar() {
   if (Fabrik::fabrik == 0) {
      if (kunde == "BMW")
         Fabrik::fabrik = new BmwFabrik();
      else if (kunde == "Ford")
         Fabrik::fabrik = new FordFabrik();
      ...
      else Fabrik::fabrik = new DefaultFabrik();
   }

   return Fabrik::fabrik;
}
```

Problem: Für jeden neuen Kunden muss die bestehende Klasse geändert werden! Die Oberklasse muss alle Unterklassen kennen, auch die, die erst in Zukunft hinzugefügt werden!

Lösung:

- Jede konkrete Fabrik registriert sich bei der `Fabrik`.
- Dazu erweitern wir die `Fabrik` um eine Liste von Fabriken.
- Jede Fabrik hat eine Funktion `type()`, die einen String zur Identifizierung (z.B. den Kundennamen) liefert.

```
class Fabrik {
private:
   static Fabrik *fabrik;
   static list<Fabrik *> fabriken;      // neu!
   ...
protected:
   Fabrik();
```

```
public:
    static Fabrik *exemplar();
    static void registerIt(Fabrik *);    // neu!
    virtual string type();               // neu!

    // eigentliche Schnittstelle definieren
    virtual Authenticator
            *createAuthenticator() = 0;

    virtual DatabaseConnector
            *createDbConnector() = 0;

    virtual LoadBalancer
            *createLoadBalancer() = 0;

    virtual Logger *createLogger() = 0;
};
```

```
// statische Variablen initialisieren
Fabrik * Fabrik::fabrik = 0;
list<Fabrik *> Fabrik::fabriken;          // neu!

string Fabrik::type() {                   // neu!
    return "Fabrik";
}

void Fabrik::registerIt(Fabrik *f) {      // neu!
    Fabrik::fabriken.push_back(f);
    cout << f->type() << " registered\n";
}
```

```
Fabrik * Fabrik::exemplar() {
    if (Fabrik::fabrik == 0) {
        list<Fabrik *>::iterator iter;

        iter = Fabrik::fabriken.begin();
        while (iter != Fabrik::fabriken.end()
                && kunde != (*iter)->type()) {
            iter++;
        }

        if (iter == Fabrik::fabriken.end())
            Fabrik::fabrik =
                    DefaultFabrik::exemplar();
        else Fabrik::fabrik = *iter;
    }

    return Fabrik::fabrik;
}
```

```
class VwFabrik : public Fabrik {
private:
    static VwFabrik *fabrik;
    string type() {                         // neu!
        return "VW";
    }
...
};

VwFabrik * VwFabrik::exemplar() {
    if (VwFabrik::fabrik == 0) {
        VwFabrik::fabrik = new VwFabrik();

        // neu:
        Fabrik::registerIt(VwFabrik::fabrik);
    }
    return VwFabrik::fabrik;
}
```

Nachteil: Alle Fabriken müssen erzeugt werden und sich registrieren, obwohl nur eine Fabrik benötigt wird!

Frage: Wer veranlasst eigentlich die Erzeugung der konkreten Fabriken, damit sie sich registrieren können?

Die Oberklasse `Fabrik` kann das nicht, da es die konkreten Klassen nicht kennt - und auch nicht kennen soll!

Antwort: Wir müssen direkt ein statisches Objekt anlegen.

```
class VwFabrik : public Fabrik {
private:
    static VwFabrik fabrik;    // kein Zeiger!
    VwFabrik();
...
};

VwFabrik VwFabrik::fabrik;    // Objekt anlegen!

VwFabrik::VwFabrik() {
    Fabrik::registerIt(this); // neu im Konstr.!
}

// die Methode "exemplar()" gibt es nicht mehr

...
```

Übersicht

- Strategy
- Abstract Factory
- Singleton
- *Decorator*
- State
- Observer
- Composite
- Model-View-Controller

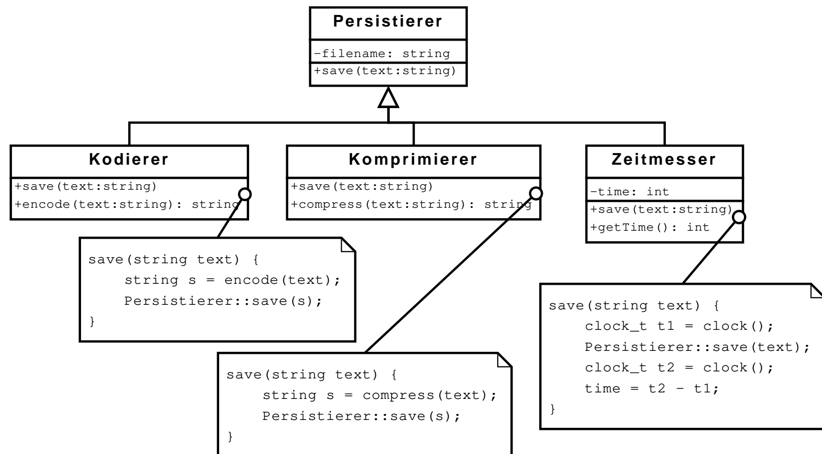
Motivation: Unsere Software hat eine Klasse `Persistierer`, die dazu dient, einen beliebigen Text in einer Datei abzuspeichern.

Persistierer
-filename: string
+save(text:string)

Im Laufe der Zeit kommen weitere Anforderungen an den Persistierer hinzu:

- Der Text soll kodiert abgelegt werden.
- Der Text soll komprimiert werden.
- Zu Testzwecken soll die zum Abspeichern benötigte Zeit gemessen werden.

mögliche Lösung:



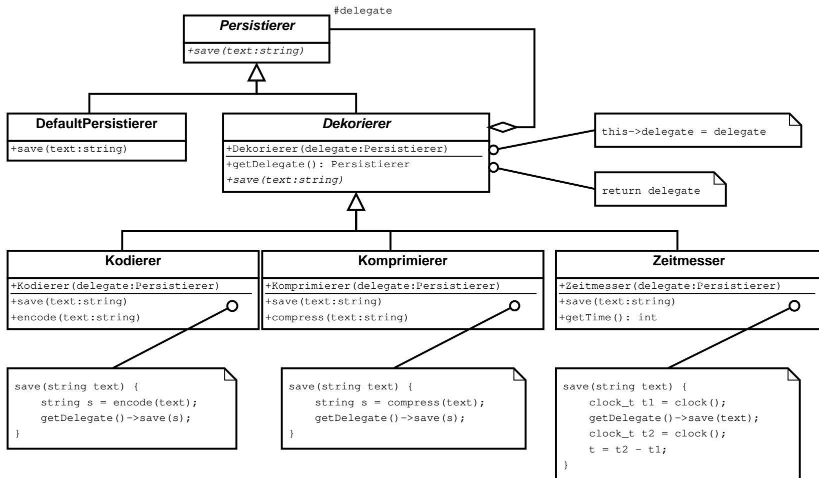
Decorator

Problem: Wie kombiniert man die Unterklassen?

- Wir wollen einen Text zunächst kodieren und dann komprimieren.
- Während der Testphase wollen wir wissen, wie lange das Komprimieren dauert.

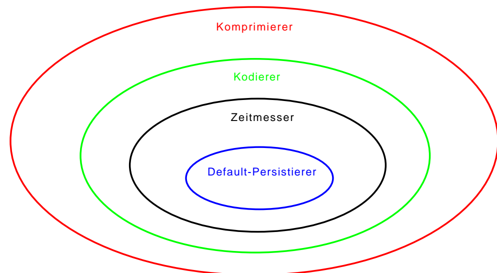
Lösung: Verwende Delegation anstelle von Vererbung!

Decorator

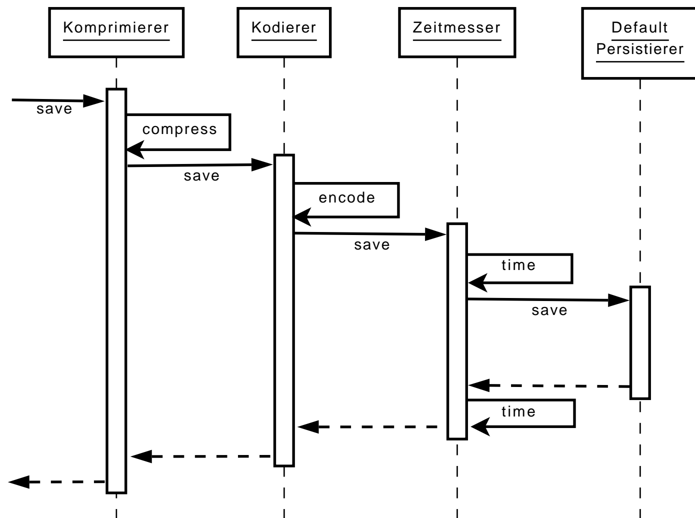


Wie wird es verwendet?

```
string text("Dies ist nur ein kleiner Test");
Persistierer *p = new Komprimierer(
    new Kodierer(
        new Zeitmesser(
            new DefaultPersistierer("test.txt"))));
p->save(text);
```



Decorator



```java
import java.io.*;
import java.util.zip.*;

public class StreamTest {
  public static void main(String[] args) {
    try {
      String str = new String(".....");

      // Ausgabe in eine Datei
      ObjectOutputStream out =
        new ObjectOutputStream(
          new GZIPOutputStream(
            new FileOutputStream("abc.zip")));
      out.writeObject(str);
      out.flush();
      out.close();
```

```
        // Einlesen aus einer Datei
        ObjectInputStream in =
          new ObjectInputStream(
            new GZIPInputStream(
               new FileInputStream("abc.zip")));
        System.out.println(in.readObject());
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
```

Übersicht

- Strategy
- Abstract Factory
- Singleton
- Decorator
- *State*
- Observer
- Composite
- Model-View-Controller

State

Motivation: Wir haben eine Methode innerhalb einer Klasse, die sich je nach Zustand anders verhalten muss.

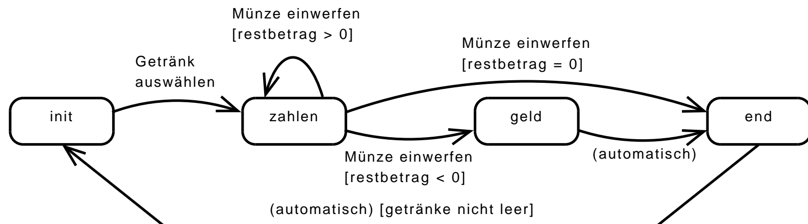
Beispiel: Der Menü-Eintrag bearbeite in einem Zeichenprogramm ist abhängig vom ausgewählten Objekt.

- Punkt: `verschiebe`
- Linie: `verschiebe`, `rechts rotieren`, `Startpfeil einfügen`
- Rechteck: `verschiebe`, `rechts rotieren`, `ausfüllen`

State

Beispiel: Anzeige eines Getränkeautomaten

- Bitte Getränk auswählen (init)
- Noch zu zahlen: 62 Cent (zahlen)
- Bitte Rückgeld entnehmen (geld)
- Bitte Getränk entnehmen (end)



Was wir nicht wollen:

```
string anzeigen() {
    if (status == INIT) {
        return "Bitte Getraenk auswaehlen";
    } else if (status == ZAHLEN) {
        ostringstream os;
        os << "Noch zu zahlen: " << restbetrag;
        return os.str();
    } else if (status == GELD) {
        return "Bitte Rueckgeld entnehmen";
    } else {
        return "Bitte Getraenk entnehmen";
    }
}
```

```
void eingeben() {
    if (status == INIT) {
        cin >> auswahl;
        if (auswahl == 0)
            exit(0);
        restbetrag = holePreis(auswahl);
        status = ZAHLEN;
    } else if (status == ZAHLEN) {
        cin >> einwurf;
        if (muenzeOk(einwurf))
            restbetrag -= einwurf;
            if (restbetrag == 0)
                status = END;
            if (restbetrag < 0)
                status = GELD;
    } else if (status == GELD) {
    .....
}
```

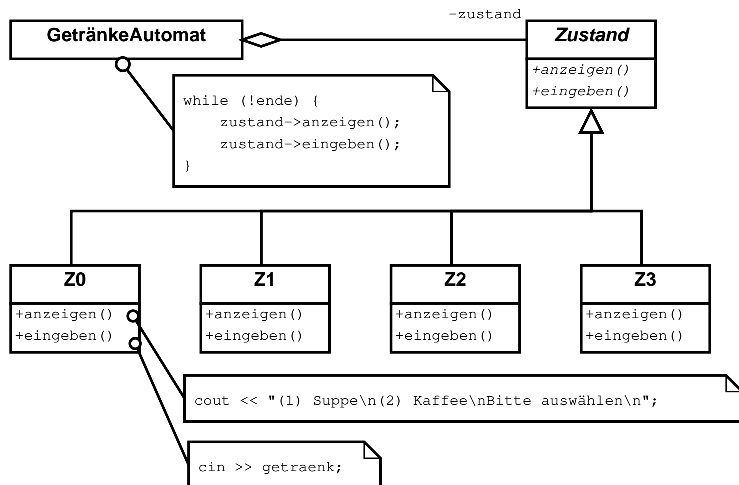
Warum wollen wir das nicht?

- Unübersichtlich, falls viele Zustände vorhanden.
- Zustandsübergänge sind nicht explizit; sie sind irgendwo in einem Haufen von Bedingungsanweisungen begraben.
- Wir haben nicht das gekapselt, was variiert!
- Wenn Code hinzugefügt wird, verursacht das mit großer Wahrscheinlichkeit Fehler.

Idee:

- Einführen einer abstrakten Klasse `Zustand`.
- Jede konkrete Unterklasse implementiert die Methode `anzeigen()` und `eingeben()`.

State

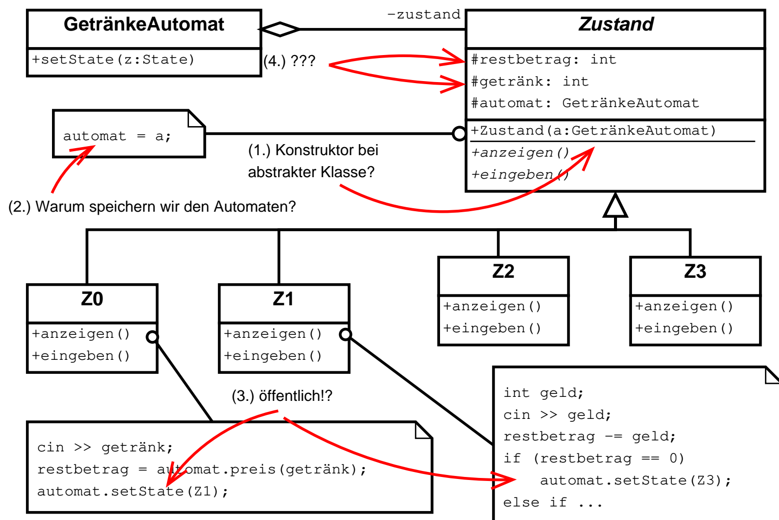


Problem: Wie wird der Zustandswechsel durchgeführt?

→ Jeder Zustand kennt seine Nachfolgezustände!

Fragen zur nächsten Folie:

- Kann eine abstrakte Klasse einen Konstruktor haben?
- Warum merken wir uns den Getränkeautomaten in dem Attribut `automat`?
- Durch die öffentliche Methode `setState()` der Klasse `Automat` kann jeder von außen den inneren Ablauf des Automaten stören! Wie können wir das vermeiden?
- Wie können die Werte der Attribute `restbetrag` und `getraenk` beim Zustandswechsel erhalten bleiben?


GetränkeAutomat
+setState(z:State)
-zustand
Zustand
#restbetrag: int
#getränk: int
#automat: GetränkeAutomat
+Zustand(a:GetränkeAutomat)
+anzeigen()
+eingeben()
(4.) ???
automat = a;
(1.) Konstruktor bei abstrakter Klasse?
(2.) Warum speichern wir den Automaten?
Z0
+anzeigen()
+eingeben()
Z1
+anzeigen()
+eingeben()
Z2
+anzeigen()
+eingeben()
Z3
+anzeigen()
+eingeben()
(3.) öffentlich!?
cin >> getränk;
restbetrag = automat.preis(getränk);
automat.setState(Z1);
int geld;
cin >> geld;
restbetrag -= geld;
if (restbetrag == 0)
automat.setState(Z3);
else if ...

State

- Kann eine abstrakte Klasse einen Konstruktor haben?

→ Ja! Nur weil ein Konstruktor definiert ist, heißt das noch lange nicht, das auch ein Objekt erzeugt werden kann.

Der Konstruktur initialisiert nur die Attribute.

State

- Warum merken wir uns den Getränkeautomaten in dem Attribut `automat`?

→ Damit wir z.B. die Preise der Getränke erfragen können, denn dafür ist der Automat verantwortlich.

State

- Wie können wir eine öffentliche Methode `setState()` in der Klasse `Automat` vermeiden?

→ Wir definieren `setState()` als `private` und erklären die Klasse `Zustand` zum Freund der Klasse `Automat`.

Funktioniert das?

Nicht ohne weiteres!

Da Freunde nicht vererbt werden, können die Sub-Klassen `Z0`, `Z1` usw. die Methode `setState()` nicht aufrufen. Es sei denn, wir definieren auch sie als Freunde von `Automat`.

Damit wir nicht alle Unterzustände als Freunde von `Automat` definieren müssen, definieren wir eine Methode

```
void Zustand::setState(Zustand *z);
```

in der Basisklasse `Zustand`. Die Methode `setState` der Basisklasse nimmt den Zustandswechsel beim Automaten vor.

Da alle abgeleiteten Zustände diese Methode erben, kann auf diesem Umweg ein Zustandswechsel durchgeführt werden.

State

- Wie können die Werte der Attribute `restbetrag` und `getraenk` beim Zustandswechsel erhalten bleiben?

$\rightarrow$ Wir definieren die Attribute als Zeiger:
  - Im Konstruktor der Basisklasse wird Speicherplatz reserviert.
  - In den Konstruktoren der abgeleiteten Klassen werden die Zeiger von der Basisklasse übernommen.

```
class Zustand {
protected:
    int *getraenk, *restbetrag;
    Automat *automat;
    Zustand();   // warum noch ein Konstruktor?
    void setState(Zustand *z) {
        automat->setState(z);
    }
public:
    Zustand(Automat *a) {
        restbetrag = new int(0);
        getraenk = new int(0);
        automat = a;
    }
    virtual ~Zustand();
    virtual void anzeigen() = 0;
    virtual void eingeben() = 0;
};
```

```
class ZBereit: public Zustand {
public:
    ZBereit(Automat *a,
            int *restbetrag, int *getraenk) {
        automat = a;
        this->restbetrag = restbetrag;
        this->getraenk = getraenk;
    }

    void anzeigen();
    void eingeben();
};
```

```
void ZBereit::eingeben() {
    int auswahl;

    cin >> auswahl;
    if (auswahl == 0)
        setState(NULL);
    else setState(new ZBetragAnzeige(...));
}
```

Was halten Sie von dieser Implementierung?

Probleme:

- Es werden sehr viele Zustandsobjekte erzeugt. Wer ist für das Freigeben dieser Objekte verantwortlich?
- Ständig werden Zustandsobjekte erzeugt und zerstört. Gibt es dadurch Laufzeitprobleme?

Idee: Wir setzen das Singleton-Entwurfsmuster ein. Dazu müssen wir aber Attribute aus der Klasse entfernen:

- Die Attribute `restbetrag` und `getraenk` können in die Klasse `Automat` ausgelagert werden.
- Das Attribut `automat` können wir nicht auslagern, daher übergeben wir diese Information an die Methoden `anzeigen()` und `eingeben()`.

```
class Zustand {     // Freund der Klasse Automat
protected:
    Zustand();

    void setState(Automat *a, Zustand *z) {
        a->setState(z);
    }


public:
    virtual ~Zustand();
    virtual void anzeigen(Automat *) = 0;
    virtual void eingeben(Automat *) = 0;

    .....
```

```
    int holeRestbetrag(Automat *a) {
        return a->restgeld;
    }

    void setzeRestbetrag(Automat *a, int b) {
        a->restgeld = b;
    }

    int holeGetraenk(Automat *a) {
        return a->getraenk;
    }

    void setzeGetraenk(Automat *a, int g) {
        a->getraenk = getr;
    }
};
```

```
class ZBereit: public Zustand {
private:
    ZBereit();
    static ZBereit *_exemplar;

public:
    void anzeigen(Automat *);
    void eingeben(Automat *);
    static ZBereit * exemplar();
};
```

```
// hier die Implementierung als Singleton
ZBereit * ZBereit::_exemplar = 0;

ZBereit::ZBereit() {}

ZBereit * ZBereit::exemplar() {
    if (ZBereit::_exemplar == 0)
        ZBereit::_exemplar = new ZBereit();

    return ZBereit::_exemplar;
}
```

```
// hier die Implementierung der Methoden
void ZBereit::anzeigen(Automat *a) {
    string *auswahl = a->holeAuswahl();
    int *preise = a->holePreise();
    int n = a->holeAnzahl();

    cout << "\nZustand: Bereit\n\n";
    cout << " (0) " << auswahl[0] << endl;
    for (int i = 1; i < n; i++) {
        cout << " (" << i << ") " << auswahl[i];
        cout << "(" << preise[i] << ")" << endl;
    }
    cout << "-------------------------------\n";
    cout << "  Ihre Auswahl? ";
}
```

```
void ZBereit::eingeben(Automat *a) {
    int auswahl;

    cin >> auswahl;
    if (auswahl == 0) {
        setState(a, NULL);
    } else {
        int betrag = a->holePreis(auswahl);

        setzeGetraenk(a, auswahl);
        setzeRestbetrag(a, betrag);
        setState(a, ZBetragAnzeige::exemplar());
    }
}
```

Ist das übersichtlicher als unsere ursprüngliche `if/else`- bzw. `switch/case`-Anweisung?

Ja! Es ist übersichtlicher, weil

- die Zustandsübergänge explizit sind,
- das Verhalten jedes Zustands in seiner eigenen Klasse lokalisiert ist und
- die Klassenstruktur sich viel enger an das Zustandsdiagramm anlehnt, wodurch der Code besser lesbar und verständlich ist.

Dadurch, dass wir die Zustandsübergänge in den Zustandsklassen untergebracht haben, erzeugen wir Abhängigkeiten zwischen den Zustandsklassen.

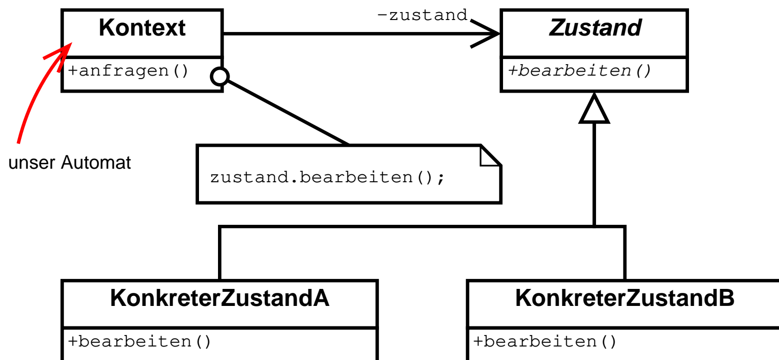
$\Rightarrow$ Kommt ein neuer Zustand hinzu, muss der bestehende Code geändert werden!

Ja, so ist das!

- Aber dadurch, dass dieser Entwurf besser lesbar und verständlich ist, sind diese Änderungen einfacher zu realisieren.
- Wir können auch den Automaten über den Fluss der Zustandsübergänge entscheiden lassen (siehe nächste Folie).

```
// Zustandswechsel durch den Automaten !!!
while (zust != NULL) {
    zust->anzeigen();
    zust->eingeben();

    string ident = typeid(*zust).name();
    if (ident == "ZBereit") {
        if (getraenk == 0)
            zust = NULL;
        else zust = ZBetragAnz::exemplar();
    } else if (ident == "ZBetragAnz") {
        if (restbetrag == 0)
            zust = ZAusgabe::exemplar();
        else if (restbetrag < 0)
            zust = ZRueckgeld::exemplar();
    } else if ...
}
```

Das State-Muster ermöglicht einem Objekt, sein Verhalten zu ändern, wenn sich sein innerer Zustand ändert.

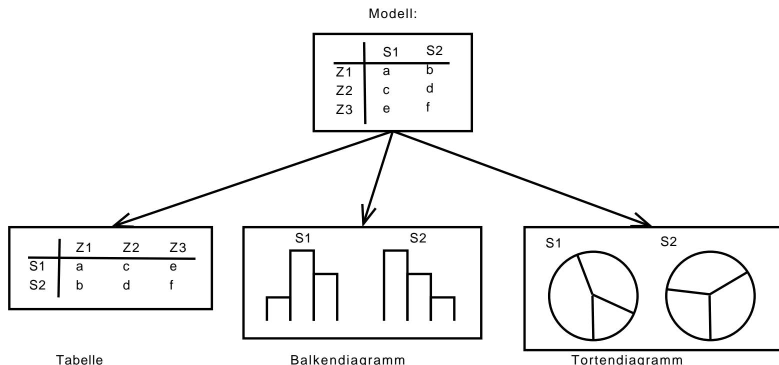


Kommt Ihnen dieses Muster nicht sehr bekannt vor?

Übersicht

- Strategy
- Abstract Factory
- Singleton
- Decorator
- State
- *Observer*
- Composite
- Model-View-Controller

Motivation: Mehrere Komponenten stellen den Zustand eines einzigen Objekts dar, zum Beispiel als Tabelle, als Balken- und als Tortendiagramm.



Ändert sich der Zustand des Objekts, müssen alle Komponenten darüber informiert werden.

Im Beispiel: Ändert sich das Tabellen-Modell, z.B. Spalte hinzu, Zeile löschen oder Eintrag ändern, dann müssen alle Diagramme und Views über diese Änderung informiert werden.

Die Diagramme und Views können sich dann neu zeichnen und die Änderungen in der neuen Darstellung berücksichtigen.

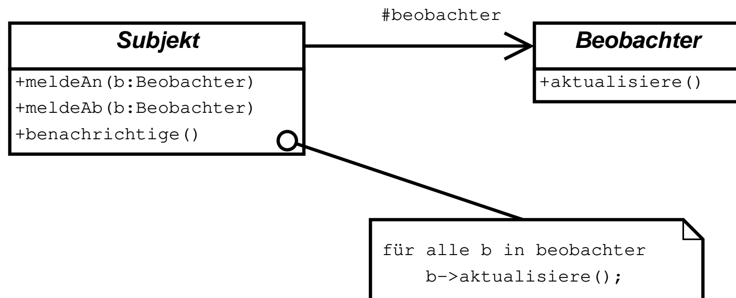
Andererseits soll das Objekt von den Komponenten unabhängig sein, also die Schnittstelle nicht kennen. Warum? Damit weitere Komponenten einfach hinzugefügt werden können.

- Im Beispiel: Wir fügen einen Protokollierer hinzu, der alle Änderungen an dem Tabellen-Modell in einer Log-Datei speichert.

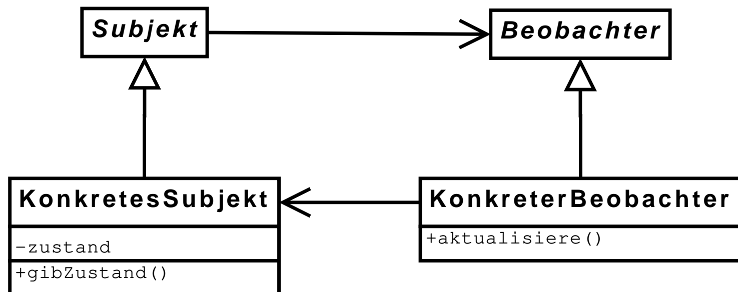
Observer

Das beobachtete Objekt, also das Subjekt, bietet einen Mechanismus, um Beobachter an- und abzumelden und diese über Änderungen zu informieren.

Das Subjekt kennt nur die Schnittstelle `Beobachter`.



Die Beobachter implementieren eine spezifische Methode, um auf die Änderung zu reagieren.



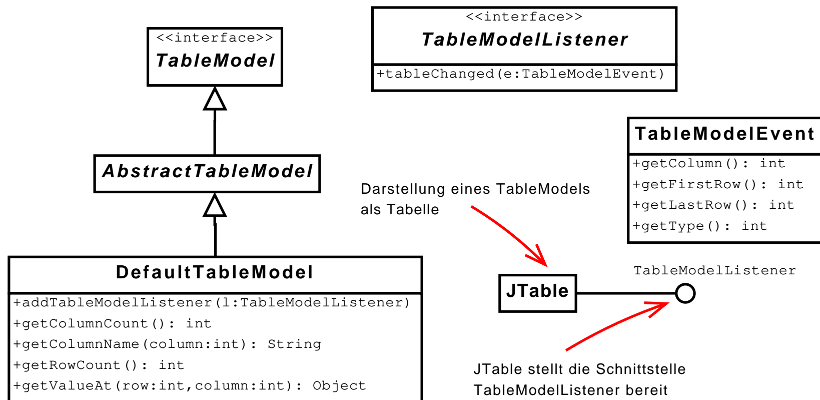
In der Regel werden die für eine Komponente relevanten Teile des Zustands abgefragt. Warum ist das schlecht?

Warum ist es schlecht, alle relevanten Teile des Zustands abfragen zu müssen?

- Die Schnittstelle von `konkretesSubjekt` wird groß, es entsteht eine starke Kopplung zwischen Beobachter und Subjekt.
- Der Beobachter muss herausfinden, was sich geändert hat. Dies kann einige Zeit in Anspruch nehmen.

Anwendung in Java:

- TableModel als Subjekt
- JTable als Darstellungskomponente, also als Beobachter



In Java wird bei `tableChanged` eine zusätzliche Information über die Änderung mitgeteilt.

Der Umfang dieser Information kann im Allgemeinen stark variieren.

- *Push-Modell:* Das Subjekt schickt detailierte Informationen über die Änderung mit, egal, ob sie interessant sind oder nicht.
- *Pull-Modell:* Es werden nur minimale Informationen verschickt, sodass der Beobachter nachfragen muss, um Details zu erfahren.

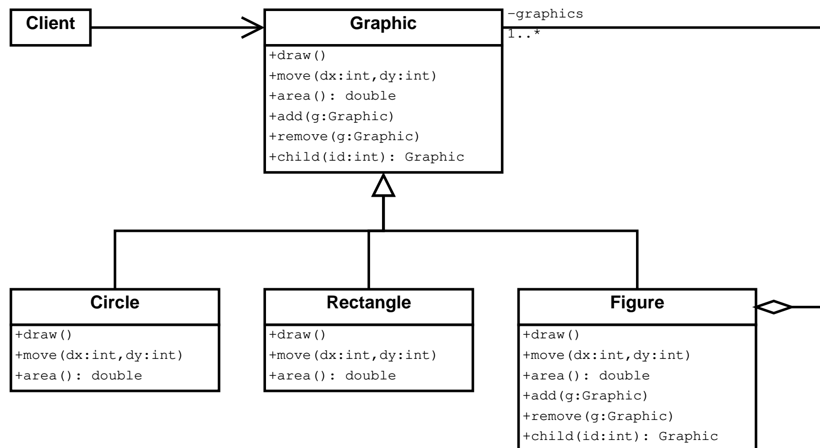
Übersicht

- Strategy
- Abstract Factory
- Singleton
- Decorator
- State
- Observer
- *Composite*
- Model-View-Controller

Motivation

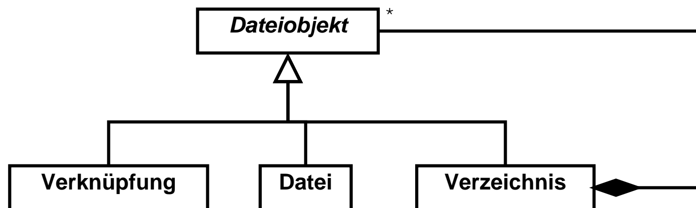
Oft ist es in grafischen Anwendungen wie Zeicheneditoren möglich, komplexe Diagramme aus einfachen Komponenten aufzubauen.

- Als Primitive stehen oft Komponenten wie Rechteck, Kreis, Dreieck oder allgemein geschlossene Polygonzüge zur Verfügung.
- Der Benutzer kann Komponenten zu größeren Komponenten zusammenfassen, die wiederum zu noch größeren Komponenten zusammengefasst werden können usw.
- Alle diese Komponenten können in vielen Fällen gleich behandelt werden. Alle haben eine
  - `move()`-Methode, mit der sich das Objekt verschieben lässt.
  - `area()`-Methode, die den Flächeninhalt des Objekts liefert.
  - `draw()`-Methode, die das Objekt zeichnet.
- Wie können wir erreichen, dass der Klient sowohl primitive als auch zusammengesetzte Objekte einheitlich behandeln kann?



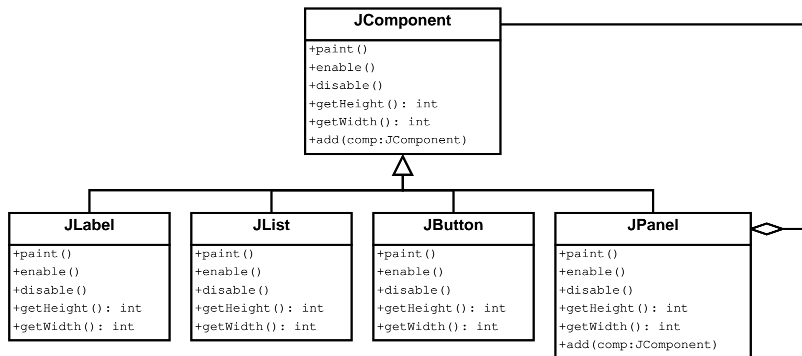
Die Klasse `Figure` definiert ein Aggregat, also eine Zusammenfassung von `Graphic`-Objekten. Die Methode `draw()` ruft einfach die draw-Methode der Kindobjekte auf.

Ein ähnliches Diagramm hatten wir schon einmal im Abschnitt „Komposition“. Nur wussten wir damals noch nicht, dass dies ein Entwurfsmuster ist.



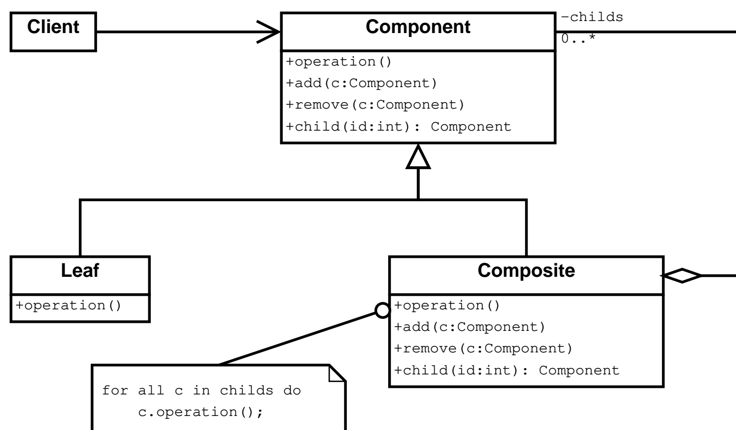
Ein Verzeichnis ist eine Zusammenfassung von Dateiobjekten. Primitive Dateiobjekte sind echte Dateien und Verweise auf Dateien. Wird ein Verzeichnis gelöscht, werden auch alle darin enthaltenen Dateiobjekte gelöscht.

Benutzeroberflächen bauen Fenster aus primitiven Objekten wie Schaltflächen (Button), Listen oder Beschriftungen (Label) und zusammengesetzten Objekten wie Feldern (Panel) auf.



Alle Komponenten haben eine Methode zum Zeichnen des Objekts, Methoden zum Aktivieren und Deaktivieren des Objekts usw.

Grundlegende Struktur des Kompositum-Musters:



Anmerkungen:

- Die allgemeine Basisklasse `Component` bietet eine Standard-Implementierung, die ggf. von den Blatt- und Kompositions-Knoten überschrieben wird.

  Im Fall des Zeicheneditors überschreiben wir in den Blatt-Klassen `Rectangle` und `Circle` die Methoden wie folgt:
  - `child(int id)` liefert `NULL`.
  - `add(Graphic)` wirft eine Exception.
  - `remove(Graphic)` wirft eine Exception.
- Werden explizit Referenzen auf Eltern-Objekte gespeichert, kann dies die Traversierung der Kompositionsstruktur vereinfachen.

  Sinnvoll ist es, diese Referenz in Component zu speichern, da alle abgeleiteten Klassen dieses Attribut erben.

Anmerkungen: (Fortsetzung)

- Der Behälter zur Speicherung der enthaltenen Kinder wie bspw. eine Liste wird im Composite abgelegt, nicht in Component. Ansonsten würde z.B. ein Rechteck einen Behälter zur Speicherung von Kindern bereit stellen, obwohl gar keine Kinder gespeichert werden. $\rightarrow$ Verschwendung von Speicherplatz.

  Die Art des Behälters, also bspw. Liste, Dictionary oder Hash-Map beeinflusst – wie immer – die Laufzeit der Operationen.

- Das Löschen eines Kompositums hat in der Regel zur Folge, dass auch die enthaltenen Kindobjekte gelöscht werden. Diese Aufgabe sollte im Destruktor des Kompositums erledigt werden.

Übersicht

- Strategy
- Abstract Factory
- Singleton
- Decorator
- State
- Observer
- Composite
- *Model-View-Controller*

Motivation

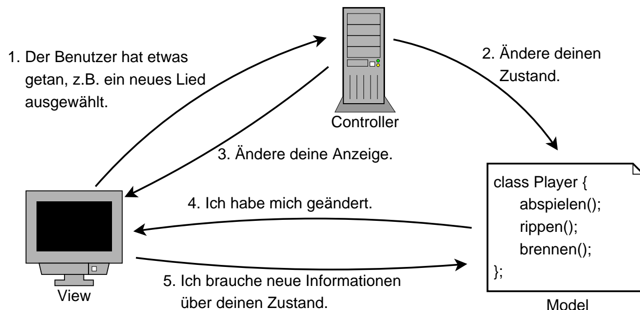
Wie funktioniert Ihr Lieblings-MP3-Player?

- Sie können über die Benutzeroberfläche neue Songs hinzufügen, Playlists verwalten und Titel umbenennen.
- Die Datenbank des Players enthält Ihre Songs mit Titel, Interpret, Abspieldauer, Genre und weiteren Informationen.
- Während des Abspielens der Stücke wird laufend die Benutzeroberfläche aktualisiert: Songtitel, Spieldauer usw.

Das Model-View-Controller-Entwurfsmuster ist kein neues Muster, sondern ein zusammengesetztes Muster. Die Trennung von Modell und Darstellung haben wir bereits beim *Beobachter* kennengelernt.

Das Beispiel ist dem Buch „Entwurfsmuster von Kopf bis Fuß" von Eric Freeman, Elisabeth Freeman, Kathy Sierra und Bert Bates entnommen, erschienen im O'Reilly-Verlag. Ein wirklich sehr empfehlenswertes Buch.

Motivation



- *View:* Präsentation des Modells. Erhält benötigte Daten i.Allg. direkt vom Modell oder fragt die Daten beim Modell ab.
- *Controller:* Stellt fest, was die Eingaben des Benutzers für das Modell bedeuten und ruft entsprechende Methoden auf.
- *Modell:* Enthält die gesamte Anwendungslogik, aber weiß nichts über View und Controller. Sein Zustand ist änder- und abrufbar. Es informiert Beobachter über Zustandsänderungen.

Motivation

Der Benutzer wählt in der Anzeige ein neues Lied aus.

1. *View an Controller:* „Benutzer hat neues Lied X ausgewählt."

   Der Controller muss nun herausfinden, was das Klicken eines Eintrags der Playlist bedeutet und welche Methoden vom Model aufzurufen sind.
2. *Controller an Model:* „Spiele neues Lied X."
3. *Controller an View:* „Pause-Button aktivieren." und „Menüpunkt Foo deaktivieren."
4. *Model an View:* „Mein Zustand hat sich geändert: Ich spiele neues Lied X."
5. *View an Model:* „Gib mir Titel und Abspielzeit vom neuen Lied X."

   Der View kann das Model auch dann nach dessen Zustand fragen, wenn er vom Controller aufgefordet wurde, die Ansicht zu verändern.

Model-View-Controller

Das MVC-Muster setzt sich aus drei Mustern zusammen:

- *Beobachter (Observer):* Das Modell ist unabhängig von der Anzeige und der Steuerung. Für das gleiche Modell können verschiedene Anzeigen verwendet werden. Ändert sich das Modell, werden alle Anzeigen benachrichtigt, die sich dann neu zeichnen können.
- *Kompositum (Composite):* Die Darstellung besteht i.Allg. aus ineinander verschachtelten Komponenten. Jede Komponente ist entweder ein Kompositum wie ein Panel, oder es ist ein Blatt wie ein Label, Button oder TextField.

  Die Steuerung bewirkt eine Aktualisierung der Anzeige, indem die oberste Komponente informiert wird. Das Composite-Muster besorgt dann den Rest.

Anzeige = View; Steuerung = Controller; Modell = Model

Fortsetzung:

- *Strategie (Strategy):* Anzeige und Steuerung implementieren das Strategie-Muster. Die Anzeige kümmert sich nur um die Darstellung des Modells und delegiert die Verarbeitung der Benutzeraktionen an die Steuerung.

  Die Steuerung kümmert sich darum, dass die Eingaben des Benutzers in Aktionen auf dem Modell übersetzt werden. Die Steuerung ist also die Strategie für die Anzeige, die Steuerung ist das Objekt, das weiß, wie man mit den Aktionen des Benutzers umgeht.

  Wir können für die Anzeige ein anderes Verhalten wählen, indem wir die Steuerung austauschen.

Noch einige Worte zum Abschluss:

- Halten Sie Ihren Entwurf so einfach wie möglich!

  Ihr Ziel sollte Einfachheit sein und nicht: “Wie kann ich ein Muster auf dieses Problem anwenden?“

- Muster sind keine Wunderwaffe!

  Sie können sie nicht einfach reinstöpseln, das Programm kompilieren und dann früh zum Mittagessen gehen. Sie müssen auch über die Folgen für den Rest Ihres Entwurfs nachdenken.

  Kann es sein, dass Entwurfsmuster auch Nachteile haben?

- Wenn Sie es im Moment nicht brauchen, dann lassen Sie es!

  Entwickler lieben es, schöne Architekturen zu erstellen, die man von allen Seiten verändern kann.

  Wenn eine praktische Notwendigkeit dafür besteht, dass Ihr Entwurf Veränderungen unterstützt, dann nutzen Sie ein entsprechendes Muster. Ist der Grund rein hypothetisch, dann lassen Sie das Muster weg.

- Refactoring-Zeit ist Musterzeit!

  Refactoring ist der Prozess, bei dem man den Code verändert, um ihn besser zu organisieren. Das Ziel ist es, seine Struktur zu verändern, nicht sein Verhalten.

  Warum das alles? Um Änderungen einfacher in das Programm einbauen zu können.

Übersicht

Software-Reengineering

- *Motivation und Begriffe*
- Software-Qualität und -Metriken
- Refactoring am Beispiel
- Katalog von Refactoring-Mustern

Motivation

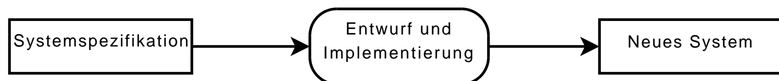
- 1990 wurde Quellcode auf 120 Milliarden Zeilen geschätzt.[1]
- Die Mehrheit dieser Systeme ist in COBOL oder FORTRAN geschrieben und läuft auf Großrechnern.
- Ein vollständiger Ersatz oder eine radikale Umstrukturierung kommt für die meisten Systeme finanziell nicht in Frage.

  Oft wird diesen Programmen nur eine hübsche grafische Benutzeroberfläche spendiert, aber im Hintergrund wird noch auf die alten Daten und Programme zugegriffen.
- Der Wartungsaufwand bei alten Systemen nimmt zu, da die Komplexität nach jeder Änderung steigt.
- Eine Umstrukturierung der Software lohnt sich, wenn das System einen hohen Geschäftswert aufweist, aber aufwändig zu warten ist.

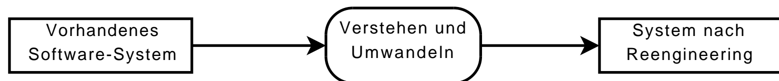
[1] W.M. Ulrich. The evolutionary growth of software reengineering and the decade ahead. American Programmer, 3(10), 14-20.

Motivation

An der Hochschule: Forward Engineering



In der Praxis: Reengineering



Literatur:

- E.J. Chikofsky, J.H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, 7(1), 13-17.
- Ian Sommerville. Software Engineering. Pearson Studium.
- Martin Fowler. Refactoring: Wie Sie das Design vorhandener Software verbessern. Addison-Wesley. Sehr empfehlenswert!

Motivation

Warum Reengineering?

- Bei einer Neuentwicklung hätte man ein höheres Risiko. (?)
  - in der bestehenden Software sind die meisten Fehler bekannt und es gibt Work-Arounds
  - in der Systemspezifikation können Fehler gemacht werden
  - es können Entwicklungsprobleme auftreten
- Geringere Kosten als bei einer Neuentwicklung. (?)
  - große Teile des Programms können evtl. wiederverwendet werden, bspw. die GUI oder die Datenzugriffsmethoden
  - es gibt bereits definierte Testfälle, Testtreiber usw.
  - große Teile der Dokumentation sind evtl. weiterhin nutzbar
  - viele Entwickler der bestehenden Software sind noch verfügbar und kennen das System wie ihre Westentasche

Es gibt aber auch Fälle, wo so viel umgeschrieben werden muss, dass eine Neuentwicklung sinnvoller ist. Bspw. wenn eine andere Programmiersprache eingesetzt wird oder sich die Architektur grundlegend ändert.

Motivation

Reengineering umfasst:

- Übersetzen des Quellcodes in eine andere Sprache.
- Analysieren des Programms und das Aktualisieren der Dokumentation.
- Verbessern der Programmstruktur, um es leichter lesbar und besser verständlich zu machen.
- Modularisieren: Verwandte Programmteile werden in Gruppen zusammengefasst und Redundanzen werden entfernt.
- Umstrukturieren und Ändern von Daten.

Begriffe

Reengineering: Alle Aktivitäten, die nach Inbetriebnahme eines Programmsystems

- das Verständnis der Software erhöhen oder
- die Wartbarkeit, Wiederverwendbarkeit oder die Weiterentwickelbarkeit verbessern oder erst ermöglichen.
- Reengineering = Reverse Engineering + Restrukturierung

Reverse Engineering: Die Extraktion und Repräsentation von Informationen aus einem Software-System in einer anderen Form oder auf höherem Abstraktionsniveau. → Programmverstehen

- erzeuge Struktogramme, Ablauf- oder UML-Diagramme, ...
- lege Querverweis-Tabellen an
- visualisiere die Software
  - imagix
  - jinsight, jdepend, jconsole, ...

Warum Reverse Engineering?

- Oft sind die Entwickler der ursprünglichen Software nicht mehr verfügbar:
  - Viele Programme sind älter als 30 Jahre und die Entwickler genießen längst ihren wohlverdienten Ruhestand.
  - Andere haben vielleicht die Firma gewechselt, weil es in der neuen Firma mehr Geld oder bessere Sozialleistungen gab.
- Die Dokumentation der Programme wird bei jeder Änderung vernachlässigt und ist nach einigen Wartungszyklen nicht mehr brauchbar.

→ Wir müssen zunächst verstehen, was die alte Software macht und wie sie es macht, wie die Daten strukturiert sind und in welcher Reihenfolge auf die Daten zugegriffen wird, welche Berechnungen erfolgen und wie sie erfolgen usw.

Erst dann können wir die Software „besser“ machen.

Begriffe

Restrukturierung Transformation zwischen Repräsentationsformen ohne Änderung der Funktionalität, die Semantik des Programms bleibt also erhalten.

- *Refactoring:* Umbau von objektorientierten Systemen, um den Code zu bereinigen und um eine bessere interne Struktur zu erhalten.
- *Wrapping:* Verpacken eines Altsystems, oder Teile daraus, in eine objektorientierte Schnittstelle.

  Beispiel: Die Methoden `time`, `mktime`, `localtime`, `gmtime`, `strftime` usw. aus C werden in eine Klasse `Time`, `Date` oder auch `DateTime` in C++ verpackt.

  Häufig bei Migration: Erst verpacken, dann nach und nach durch neue Implementierung ersetzen.

In dieser Veranstaltung: Refactoring

Übersicht

Software-Reengineering

- Motivation und Begriffe
- *Software-Qualität und -Metriken*
- Refactoring am Beispiel
- Katalog von Refactoring-Mustern

Software-Qualität

Nach dem Umstrukturieren einer Software,

- soll die innere Struktur „besser“ sein als vorher,
- sollen Änderungen „einfacher“ möglich sein als vorher,
- soll das Weiterentwickeln „leichter“ möglich sein usw.

Wie beurteilt man „besser“, „einfacher“ oder „leichter“? Wann ist eine Software „gut“, wann ist sie „einfach erweiterbar“? Gibt es eine Möglichkeit, die Güte zu messen?

Software-Qualität ist ein schwer fassbarer Begriff, daher werden Qualitätsmodelle wie bspw. die ISO/IEC 9126 eingeführt.

Software-Qualität

Die Norm ISO/IEC 9126 (in ISO 25000 aufgegangen) bezieht sich nur auf die Produktqualität, nicht auf die Prozessqualität.

- *Funktionalität:* Besitzt die Software die geforderten Funktionen?
- *Zuverlässigkeit:* Kann das Leistungsniveau unter definierten Bedingungen über einen vorgegebenen Zeitraum aufrechterhalten werden?
- *Benutzbarkeit:* Wie wird der Einsatz der Software von den Benutzern beurteilt?
- *Effizienz:* Verhältnis zwischen Leistungsniveau der Software und eingesetzten Betriebsmitteln?
- *Änderbarkeit:* Wie groß ist der Aufwand, um vorgegebene Änderungen an der Software durchzuführen?

Software-Qualität

Für jeden der genannten Punkte gibt es weitere Unterpunkte:

- *Funktionalität:* Angemessenheit, Richtigkeit, Interoperabilität, Sicherheit und Ordnungsmäßigkeit
- *Zuverlässigkeit:* Reife, Fehlertoleranz, Robustheit, Wiederherstellbarkeit und Konformität
- *Benutzbarkeit:* Verständlichkeit, Erlernbarkeit, Bedienbarkeit, Attraktivität und Konformität
- *Effizienz:* Zeitverhalten, Verbrauchsverhalten und Konformität
- *Änderbarkeit:* Analysierbarkeit, Modifizierbarkeit, Stabilität und Testbarkeit
- *Übertragbarkeit:* Anpassbarkeit, Installierbarkeit, Koexistenz, Austauschbarkeit und Konformität

Wichtige Normen für Software-Entwicklung

ISO 9241-110: Richlinien zur Dialoggestaltung

- Aufgabenangemessenheit
- Selbstbeschreibungsfähigkeit
- Lernförderlichkeit
- Steuerbarkeit
- Erwartungskonformität
- Individualisierbarkeit
- Fehlertoleranz

ISO 29119: Software Testing

- Konzepte und Definitionen
- Testprozesse
- Testdokumentation
- Testtechniken

Andere Modelle wie bspw. V-Modell, Rational Unified Process oder Agile Methoden wie Scrum konzentrieren sich nicht auf die Produktqualität, sondern auf den Prozess der Produkterstellung, also der Prozessqualität.

Solche Modelle gehen davon aus, dass ein hochwertiger Prozess der Produkterstellung die Entstehung von guten Produkten begünstigt.

Egal, ob ein Modell auf die Produkt- oder auf die Prozessqualität abzielt, bei den Qualitätsindikatoren sollte es sich um beobachtbare oder messbare Sachverhalte handeln.

Hier kommen *Softwaremetriken* zum Einsatz. Eine Softwaremetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit. (IEEE Standard 1061, 1992)

Messgrößen im Software-Prozess:

- Ressourcenaufwand (Mitarbeiter, Zeit, Geld)
- Fehler
- Kommunikationsaufwand

Messgrößen im Software-Produkt:

- Umfang (LOC, %Wiederverwendung, #Prozeduren)
- Komplexität
- Lesbarkeit (Stil)
- Entwurfsqualität (Modularität, Kopplung, Bindung)
- Produktqualität (Testergebnisse, Testabdeckung)

Welche Gütekriterien muss eine Software-Metrik erfüllen?

- *Objektivität:* Keine subjektiven Einflüsse des Messenden auf die Messung sind möglich.
- *Zuverlässigkeit:* (Messgenauigkeit) Bei einer Wiederholung der Messung werden dieselben Ergebnisse erzielt.
- *Validität:* (Gültigkeit) Die Messergebnisse ermöglichen einen eindeutigen, unmittelbaren Rückschluss auf die Kenngröße; die Metrik misst tatsächlich (nur) die zu messende Eigenschaft.
- *Normierung:* Es existiert eine Skala, auf der Messergebnisse eingetragen werden und verglichen werden können.
- *Vergleichbarkeit:* Ein Maß ist vergleichbar, wenn es mit anderen Maßen in Relation steht.
- *Ökonomie:* Die Messung hat geringe Kosten, abhängig vom Automatisierungsgrad, der Anzahl der Berechnungsschritte, ...
- *Nützlichkeit:* Die Messung erfüllt praktische Bedürfnisse.

Messen der Verständlichkeit:

- Gutachter lesen Programme und beurteilen sie anhand von Fragen:
  - Ist das Programm an veränderte Rahmenbedingungen anpassbar?
  - Sind die Bezeichner aussagekräftig und konsistent?
- Beurteilung erfolgt z.B. in Form von Schulnoten.

Der Ansatz ist wenig objektiv, wenig zuverlässig, halbwegs normiert, halbwegs vergleichbar, nicht ökonomisch, aber nützlich und valide.

Gibt es maschinell prüfbare Indikatoren für Verständlichkeit?

Umfangs-Metriken

- *Lines Of Code (LOC):* Zählt die Anzahl der Zeilen im Code.
- *Non-Commented Source Statements (NCSS):* Wie LOC, aber Leer- und Kommentarzeilen werden ignoriert.

positiv:

- Sehr einfach zu messen und zu berechnen.
- Anwendbar auf alle Arten von Programmen.

negativ:

- Was soll gezählt werden?
- Umfang abhängig von der Sprache (Perl vs. C++)

Halstead-Metriken

- Eingeführt zur Messung der textuellen Komplexität.
- Einteilen des Programms in Operatoren und Operanden.

Beispiel:

```
int ggt(int a, int b) {
    assert(a >= 0);
    assert(b >= 0);
    while (b > 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

Operatoren:

- kennzeichnen Aktionen
- typisch: Sprachelemente des Programms
- hier: `int`, `()`, `,`, `{}`, `%`, `>`, `>=`, `assert`

Operanden:

- kennzeichnen Daten
- typisch: Bezeichner und Literale
- hier: `ggt`, `a`, `b`, `r`, `0`

Halstead-Metriken

Basisgrößen:

- n_1: Anzahl unterschiedlicher Operatoren, hier $n_1 = 14$
- n_2: Anzahl unterschiedlicher Operanden, hier $n_2 = 5$
- N_1: Anzahl verwendeter Operatoren, hier $N_1 = 32$
- N_2: Anzahl verwendeter Operanden, hier $N_2 = 17$
- $n = n_1 + n_2$: Größe des Vokabulars
- $N = N_1 + N_2$: Länge der Implementierung

abgeleitete Größen:

- difficulty $D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$
  Schwierigkeit, ein Programm zu verstehen
- volume $V = N \cdot \log(n)$
  Umfang des Programms
- effort $E = D \cdot V$
  Aufwand, das Programm zu verstehen

Halstead-Metriken

positiv:

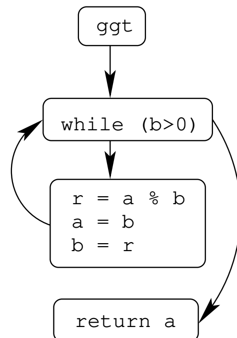
- Einfach zu ermitteln und zu berechnen.
- Für alle Programmiersprachen einsetzbar.
- Experimente zeigen: gutes Maß für Komplexität.

negativ:

- Berücksichtigt nur textuelle Komplexität.
- Moderne Programmierkonzepte wie Sichtbarkeit oder Namensräume werden nicht berücksichtigt.
- Aufteilung Operatoren/Operanden sprachabhängig.

McCabe-Metriken

- Wir wollen auch die strukturelle Komplexität berücksichtigen.
- Quelle: Kontrollflussgraph G
- Ziel: zyklomatische Komplexität $V(G)$



$V(G) = e - n + 2p$

- e: Anzahl Kanten (hier 4)
- n: Anzahl Knoten (hier 4)
- p: Anzahl Komponenten (hier 1)

$\Rightarrow$ hier: $V(G) = 2$

für $p = 1$ gilt: $V(G) = \pi + 1$
π: Anzahl der Bedingungen

McCabe-Metriken

Faustregeln:

$V(G)$	Risiko
1-10	einfaches Programm, geringes Risiko
11-20	komplexeres Programm, erträgliches Risiko
21-50	komplexes Programm, hohes Risiko
> 50	untestbares Programm, extrem hohes Risiko

Beginne bei einer Umstrukturierung mit der Komponente, die die höchste zyklomatische Komplexität hat.

McCabe-Metriken

positiv:

- Einfach zu berechnen.
- Integration mit Testplanung (Bedingungsüberdeckung).
- Studien zeigen: gute Korrelation zwischen zyklomatischer Zahl und Verständlichkeit einer Komponente.

negativ:

- Metrik berücksichtigt nur den Kontrollfluss, die Komplexität des Datenflusses wird nicht berücksichtigt.
- Der Kontrollfluss zwischen den Komponenten ist evtl. sehr komplex, wird aber nur unzureichend berücksichtigt.
- Die Komplexität einzelner Anweisungen, der Einfluss langer Sequenzen oder die Schachtelungstiefe von Schleifen und Auswahlanweisungen wird nicht berücksichtigt.
- Ungeeignet für objektorientierte Programme, die oft viele triviale Funktionen enthalten.

hybride Metriken

Idee: Kombiniere verschiedene Metriken.

Beispiel: Wartbarkeits-Index MI bei Hewlett-Packard.

$$171 - 5.2\ln(\overline{V}) - 0.23\overline{V(G)} - 16.2\ln(\overline{L}) + 50\sin\left(\sqrt{2.4\overline{C}}\right)$$

Bedeutung der Symbole:

- $\overline{V}$: durchschnittliches Halstead-Volumen pro Modul
- $\overline{V(G)}$: durchschnittliche zyklomatische Zahl pro Modul
- $\overline{L}$: durchschnittliche LOC pro Modul
- $\overline{C}$: durchschnittlicher Prozentsatz an Kommentarzeilen

Unterschreitet ein Modul einen vorgegebenen Wert, dann wird das Modul restrukturiert.

Die McCabe-Metrik versagt bei objektorientierten Programmen, da die Kontrollflusskomplexität der meisten Methoden gering ist, oft gilt $V(G) = 1$.

Metriken für objektorientierte Programme müssen daher das Zusammenspiel der Klassen betrachten! Man unterscheidet:

- *Metriken auf Methodenebene* messen die Eigenschaften der einzelnen Methoden.
- *Metriken auf Klassenebene* messen die Strukturmerkmale einer Klasse.
- *Metriken auf Vererbungshierarchien* beachten Strukturen, die durch Vererbung und Abstraktion entstehen.
- *Metriken auf Aggregationshierarchien* betrachten die Verknüpfungen der Klassen untereinander.

Metriken für OO-Komponenten

- *DIT (Depth of Inheritance Tree)*
  - Anzahl Oberklassen einer Klasse.
  - DIT größer $\Rightarrow$ Fehlerwahrscheinlichkeit größer
  - Je mehr Oberklassen zu einer Klasse existieren, desto komplexer ist die Klasse und die Hierarchie: Bei Änderungen einer Oberklasse müssen evtl. auch abgeleitete Klassen verändert werden. Die Wartbarkeit der Klassen wird erschwert.
- *NOC (Number Of Children of a class)*
  - Anzahl direkter Unterklassen.
  - NOC größer $\Rightarrow$ Fehlerwahrscheinlichkeit geringer
  - Je größer die Anzahl der direkten Nachfolger ist, umso größer ist die Wiederverwendung durch Vererbung. Gäbe es einen Fehler in der Klasse, wäre der Fehler bestimmt bei einem der Nachfolger aufgefallen.

Metriken für OO-Komponenten

- *RFC (Response For a Class)*
  - Anzahl Methoden, die ausgeführt werden können, wenn eine Klassenmethode aufgerufen wird. Beinhaltet auch solche Funktionen, die nicht in der Klasse selbst liegen, sondern durch Kopplung mit anderen Klassen erreichbar sind.
  - RFC größer $\Rightarrow$ Fehlerwahrscheinlichkeit größer
  - Je mehr Methoden aufgerufen werden können, um so komplexer ist die Klasse, und um so größer ist der Testaufwand und die Fehlerwahrscheinlichkeit.
- *WMC (Weighted Methods per Class)*
  - Anzahl Klassenmethoden, optional gewichtet nach Größe oder Komplexität.
  - WMC größer $\Rightarrow$ Fehlerwahrscheinlichkeit größer
  - Ist ein Maß für die Wiederverwendbarkeit einer Klasse. Je größer die Anzahl von Methoden in einer Klasse ist, desto spezifischer und komplexer ist die Klasse, und um so größer ist die Fehlerwahrscheinlichkeit, aber um so geringer ist die Wiederverwendbarkeit.

Metriken für OO-Komponenten

- *CBO (Coupling Between Object classes)*
  - Anzahl Klassen, auf deren Dienste die Klasse zugreift oder für die die Klasse Dienste bereit stellt.
  - CBO größer $\Rightarrow$ Fehlerwahrscheinlichkeit größer
  - Eine starke Kopplung wirkt sich immer nachteilig auf die Wiederverwendung und Testbarkeit der Klassen aus. Je mehr Klassen einer Software gekoppelt sind, desto anfälliger sind die Klassen für Änderungen.
- *LCOM (Lack of COhesion in Methods)*
  - Anzahl Methodenpaare ohne gemeinsame Klassenvariablen minus Anzahl Methodenpaare mit gemeinsamen Klassenvariablen.
  - LCOM hoch $\Rightarrow$ Fehlerwahrscheinlichkeit hoch
  - Hoher LCOM-Wert bedeutet fehlender oder geringer Zusammenhang, was eine hohe Komplexität der Klasse bedeutet. Als Folge sollte die Klasse aufgeteilt werden.

Metriken für OO-Komponenten

Kombination von DIT, NOC, RFC usw. mit McCabe- oder Halstead-Metrik möglich.

In der Literatur:

- Autoren schlagen neue Kombinationsmetrik vor
- und beschreiben, welche Werte die Metrik berechnet.
- Aber oft fehlt die Validierung.

Kritik an Metriken:

- Berechnen von Metriken ist kein Ersatz für Gegenlesen, Test oder Verifikation.
- Was ist mit neuronalen Netzen, Data Mining, ...?

Metriken für dynamische Aspekte, die bspw. auf Sequenz-, Aktivitäts- oder Zustandsdiagrammen basieren: bisher Fehlanzeige!

Übersicht

Software-Reengineering

- Motivation und Begriffe
- Software-Qualität und -Metriken
- *Refactoring am Beispiel*
- Katalog von Refactoring-Mustern

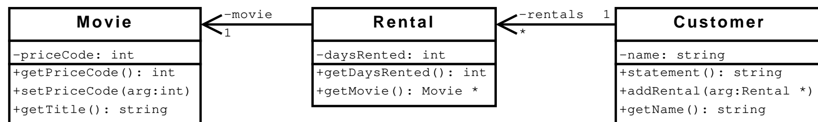
- Ändern eines Software-Systems ohne das externe Verhalten zu beeinflussen, aber mit dem Ziel, den Code zu bereinigen, um eine bessere interne Struktur zu erhalten.
- Refactoring beschreibt kein allgemeines Vorgehen. Aber es gibt einen Katalog von Refactoring-Methoden, ähnlich der Entwurfsmuster.
- Jeder Refactoring-Schritt ist einfach/primitiv.
- Refactoring erfordert automatisierte Tests.

Beispiel

Eine sehr gute, verständliche Einführung in das Thema gibt das Buch von Martin Fowler: Refactoring. Addison-Wesley. Der folgende Abschnitt ist diesem Buch entnommen.

Rechnungen für Kunden einer Videothek erstellen:

- Die zu zahlende Miete ist abhängig von der Leihdauer und der Art des ausgeliehenen Films.
- Es gibt drei Arten von Filmen: reguläre Filme, Kinderfilme und Neuerscheinungen.
- Für gute Kunden gibt es Bonuspunkte bei Neuerscheinungen.



Beispiel: Movie

```
class Movie {
private:
    string _title;
    int _priceCode;

public:
    const static int CHILDRENS = 2;
    const static int NEW_RELEASE = 1;
    const static int REGULAR = 0;

    Movie(string title, int code) {
        _title = title;
        _priceCode = code;
    }
```

```
    int getPriceCode() {
        return _priceCode;
    }

    void setPriceCode(int arg) {
        _priceCode = arg;
    }

    string getTitle() {
        return _title;
    }
};
```

Beispiel: Rental

```
class Rental {
private:
    Movie *_movie;
    int _daysRented;

public:
    Rental(Movie *movie, int daysRented) {
        _movie = movie;
        _daysRented = daysRented;
    }
    int getDaysRented() {
        return _daysRented;
    }
    Movie *getMovie() {
        return _movie;
    }
};
```

Beispiel: Customer

```
class Customer {
private:
    string _name;
    vector<Rental *> _rentals;

public:
    Customer(string name) {
        _name = name;
    }
    void addRental(Rental *arg) {
        _rentals.push_back(arg);
    }
    string getName() {
        return _name;
    }
    string statement();
};
```

Beispiel: Customer

```
string Customer::statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    vector<Rental *>::iterator it;
    ostringstream res;

    res << "Rental Record for ";
    res << getName() << "\n";

    for (it = _rentals.begin();
              it != _rentals.end(); it++) {
        double amount = 0;
        int daysRented = (*it)->getDaysRented();
        int priceCode =
              (*it)->getMovie()->getPriceCode();
```

Beispiel: Customer

```
switch (priceCode) {
case Movie::REGULAR:
    amount += 2;
    if (daysRented > 2)
        amount += (daysRented-2) * 1.5;
    break;
case Movie::NEW_RELEASE:
    amount += daysRented * 3;
    break;
case Movie::CHILDRENS:
    amount += 1.5;
    if (daysRented > 3)
        amount += (daysRented-3) * 1.5;
    break;
}
```

Beispiel: Customer

```
        frequentRenterPoints += 1;
        if ((priceCode == Movie::NEW_RELEASE)
                && daysRented > 1)
            frequentRenterPoints += 1;

        res << "\t"
            << (*it)->getMovie()->getTitle();
        res << "\t" << amount << "\n";
        totalAmount += amount;
    } // end for

    res << "Amount owed is " << totalAmount
        << "\nYou earned "
        << frequentRenterPoints
        << " frequent renter points\n";
    return res.str();
}
```

Guter Software-Entwurf?

Anmerkungen

- Nicht gut gestaltet, nicht objektorientiert: keine Vererbung, keine Delegation, kein Polymorphismus, usw.
  - → Quick and dirty ist o.k. für kleine Programme, ist aber katastrophal für komplexe Programme.
- Die Methode `statement` in der Klasse `Customer` tut entschieden zu viel: Eine Methode, eine Aufgabe.
- Aber: Das Programm funktioniert!
  - → Never change a running system ?????
- Den Compiler stört es nicht, wenn Code hässlich und unsauber ist. Aber: Bei Systemänderungen sind Menschen beteiligt!

  Ein schlecht gestaltetes System ist schwer zu ändern, denn es muss zunächst festgestellt werden, wo und welche Änderungen notwendig sind.
  - → Programmierer irrt sich und baut Fehler ein.

Veränderung der Anforderungen

Rechnung soll auch in HTML ausgegeben werden:

- Aus der Methode `statement` kann nichts wiederverwendet werden.

→ Copy-and-Paste: Methode kopieren und alles Notwendige ändern???

Was tun, wenn sich dann Abrechnungsregeln ändern?

- `statement` und `htmlStatement` ändern und sicherstellen, dass die Änderungen konsistent sind???

→ Sehr fehleranfälliges Vorgehen!

Kopieren und Einfügen ist ein Fluch, wenn sich Programme ändern!

Wir können unsere Änderung so nicht einbauen, deshalb:

- Zuerst die Struktur des alten Programms umbauen,
- dann Änderungen im umstrukturierten Programm einbauen.

Der erste Schritt: Baue eine solide Menge von Testfällen für den zu ändernden Code-Abschnitt auf.

- Menschen machen Fehler. Deshalb sind Tests unbedingt notwendig!
- Die Tests müssen selbstüberprüfend sein, andernfalls wird Zeit damit verschwendet, Ergebnisse zu kontrollieren.

Wichtig: Das Programm muss in kleinen Schritten geändert werden, damit ein Fehler leicht zu finden ist. Der Fehler kann dann nur bei der letzten Änderung aufgetreten sein.

Hier: Zerlegen und umverteilen der Methode `statement` der Klasse `Customer`:

- Kleine Stücke Code sind einfacher zu verstehen, leichter zu verschieben und besser wiederverwendbar.
- Finde ein logisch zusammenhängendes Stück Code und wende *Methode extrahieren* an.

  Hier: `switch`-Befehl
- Unsere Testsuite soll sicher stellen, dass wir durch die Änderung keine Fehler ins Programm eingebaut haben.

Version 2: Customer

```
class Customer {
private:
    string _name;
    vector<Rental *> _rentals;
    double amountFor(Rental *);    // neu!

public:
    Customer(string name) {
        _name = name;
    }
    void addRental(Rental *arg) {
        _rentals.push_back(arg);
    }
    string getName() {
        return _name;
    }
    string statement();
};
```

Version 2: Customer

```
string Customer::statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    vector<Rental *>::iterator it;
    ostringstream res;

    res << "Rental Record for ";
    res << getName() << "\n";

    for (it = _rentals.begin();
            it != _rentals.end(); it++) {
        double amount = 0;
        int priceCode =
            (*it)->getMovie()->getPriceCode();

        amount = amountFor(*it);    // neu!
```

```
            frequentRenterPoints += 1;
            if ((priceCode == Movie::NEW_RELEASE)
                    && (*it)->getDaysRented() > 1)
                frequentRenterPoints += 1;

            res << "\t"
                << (*it)->getMovie()->getTitle();
            res << "\t" << amount << "\n";
            totalAmount += amount;
        } // end for

        res << "Amount owed is " << totalAmount
            << "\nYou earned "
            << frequentRenterPoints
            << " frequent renter points\n";
        return res.str();
    }
```

Version 2: Customer

```
double Customer::amountFor(Rental *arg) {
    double res = 0;
    switch (arg->getMovie()->getPriceCode()) {
    case Movie::REGULAR:
        res += 2;
        if (arg->getDaysRented() > 2)
            res += (arg->getDaysRented() -2) * 1.5;
        break;
    case Movie::NEW_RELEASE:
        res += arg->getDaysRented() * 3;
        break;
    case Movie::CHILDRENS:
        res += 1.5;
        if (arg->getDaysRented() > 3)
            res += (arg->getDaysRented() -3) * 1.5;
    }
    return res;
}
```

Version 2

Anmerkungen:

- Wir sollten einige Variablen und Funktionen umbenennen.
  - → Guter Code drückt klar aus, was er tut! Gute Variablen- und Funktionsnamen sind ein Schlüssel zu klarem Code.
- In unserem Beispiel:
  - `amountFor()` ⟶ `getCharge()`
  - `arg` ⟶ `aRental`

> Jeder Dummkopf kann Code schreiben, den ein Computer versteht. Gute Programmierer schreiben Code, den Menschen verstehen!

Verschieben der Betragsrechnung

Wir stellen fest: Die Methode `getCharge()` verwendet nur Informationen der Klasse `Rental`, nicht der Klasse `Customer`.

Deshalb: Wir verschieben die Methode in die Klasse `Rental`.

```
class Rental {
private:
    Movie *_movie;
    int _daysRented;

public:
    Rental(Movie *movie, int daysRented);
    int getDaysRented();
    Movie *getMovie();
    double getCharge();    // neu!
};
```

Verschieben der Betragsrechnung

```
double Rental::getCharge() {
    double res = 0;
    switch (getMovie()->getPriceCode()) {
    case Movie::REGULAR:
        res += 2;
        if (getDaysRented() > 2)
            res += (getDaysRented() - 2) * 1.5;
        break;
    case Movie::NEW_RELEASE:
        res += getDaysRented() * 3;
        break;
    case Movie::CHILDRENS:
        res += 1.5;
        if (getDaysRented() > 3)
            res += (getDaysRented() - 3) * 1.5;
    }
    return res;
}
```

Wir müssen nun alle Verwendungen der alten Methode finden und durch einen Aufruf der neuen Methode ersetzen. Moderne Entwicklungsumgebungen unterstützen uns bei dieser Arbeit!

```
string Customer::statement() {
    .....
    for (it = _rentals.begin();
            it != _rentals.end(); it++) {
        double amount = 0;

        amount = (*it)->getCharge();  // neu!

        frequentRenterPoints += 1;
        .....
    } // end for
    .....
    return res.str();
}
```

Extrahieren der Bonuspunkte

Auf die gleiche Weise können wir auch die Berechnung der Bonuspunkte extrahieren und verschieben.

```
int Rental::getFrequentRenterPoints() {
    int res = 1;

    if ((getMovie()->getPriceCode()
                == Movie::NEW_RELEASE)
            && getDaysRented() > 1)
        res += 1;
    return res;
}
```

Extrahieren der Bonuspunkte

```
string Customer::statement() {
    .....
    for (it = _rentals.begin();
            it != _rentals.end(); it++) {
        double amount = 0;

        amount = (*it)->getCharge();
        frequentRenterPoints +=
            (*it)->getFrequentRenterPoints();
        .....
    } // end for

    res << "Amount owed is " << totalAmount
        << "\nYou earned "
        << frequentRenterPoints
        << " frequent renter points";
    return res.str();
}
```

Entfernen temporärer Variablen

Temporäre Variablen können ein Problem sein:

- Sie sind nur innerhalb der Routine sinnvoll und führen zu langen, komplexen Routinen.
- Wende *Temporäre Variable durch Abfrage ersetzen* an:
  - `amount` ⟶ `getCharge()`
  - `totalAmount` ⟶ `getTotalCharge()`
  - `frequentRenterPoints` ⟶ `getTotalFreqRenterPoints()`

  Durch diese Änderungen können wir die nun in Methoden bereit gestellten Berechnungen in `htmlStatement` wieder verwenden!

Entfernen temporärer Variablen

```
class Customer {
private:
    string _name;
    vector<Rental *> _rentals;
    double getTotalCharge();                    // neu!
    int getTotalFreqRenterPoints();             // neu!

public:
    Customer(string name);
    void addRental(Rental *arg);
    string getName();
    string statement();
};
```

Entfernen temporärer Variablen

```
double Customer::getTotalCharge() {      // neu!
    double res = 0;
    vector<Rental *>::iterator it;

    for (it = _rentals.begin();
            it != _rentals.end(); it++)
        res += (*it)->getCharge();
    return res;
}
int Customer::getTotalFreqRenterPoints(){ // neu
    int res = 0;
    vector<Rental *>::iterator it;

    for (it = _rentals.begin();
            it != _rentals.end(); it++)
        res += (*it)->getFrequentRenterPoints();
    return res;
}
```

Entfernen temporärer Variablen

```
string Customer::statement() {
  vector<Rental *>::iterator it;
  ostringstream res;

  res << "Rental Record for "
      << getName() << "\n";
  for (it = _rentals.begin();
          it != _rentals.end(); it++) {
    res << "\t"
        << (*it)->getMovie()->getTitle();
    res << "\t" << (*it)->getCharge() << "\n";
  }
  res << "Amount owed is " << getTotalCharge()
      << "\nYou earned "
      << getTotalFreqRenterPoints()
      << " frequent renter points";
  return res.str();
}
```

Hinzufügen neuer Funktionalität

Durch das Herausziehen der Berechnungen können wir eine Methode `htmlStatement` schreiben und den gesamten Berechnungscode wiederverwenden.

```
string Customer::htmlStatement() {
    vector<Rental *>::iterator it;
    ostringstream res;

    res << "<html>\n<head>\n";
    res << "<title>Statement</title>\n";
    res << "</head>\n<body>\n";
    res << "<h1>Rental Record for <em>"
        << getName();
    res << "</em></h1><p>\n";
```

```
    for (it = _rentals.begin();
            it != _rentals.end(); it++) {
        res << (*it)->getMovie()->getTitle();
        res << ": " << (*it)->getCharge()
            << "<br>\n";
    }

    res << "<p>You owe <em>" << getTotalCharge();
    res << "</em>\n<p>You earned <em>";
    res << getTotalFrequentRenterPoints();
    res << "</em> frequent renter points\n";
    res << "</body>\n</html>\n";

    return res.str();
}
```

Hinzufügen neuer Funktionalität

Anmerkungen:

- Berechnungscode, der in der Methode `statement` war, kann wiederverwendet werden, und zwar ohne Copy-and-Paste.
- Änderungen an der Berechnung können auch in Zukunft an einer einzigen Stelle vorgenommen werden.
- Jede andere Art der Ausgabe kann schnell und einfach eingebaut werden.

Weitere Änderung: Klassifikation der Filme

- Die Änderungen sind im einzelnen noch nicht ganz klar.
- Problematisch sind die komplizierten `switch`-Befehle in `getCharge()` und `getFrequentRenterPoints()`.

Methode extrahieren und verschieben

Regel: Verzweige nicht aufgrund der Werte anderer Klassen.

- Der `switch`-Befehl in `Rental::getCharge()` gehört in die Klasse `Movie`.
- Das Gleiche gilt für die Berechnung der Bonuspunkte.

Wende *Methode extrahieren* und danach *Methode verschieben* an.

Methode extrahieren und verschieben

```
class Movie {
.....
   Movie(string title, int code);
   int getPriceCode();
   void setPriceCode(int arg);
   string getTitle();
   int getFrequentRenterPoints(int days); // neu
   double getCharge(int days);             // neu
};

int Movie::getFrequentRenterPoints(int days) {
   int res = 1;

   if ((getPriceCode() == Movie::NEW_RELEASE)
           && days > 1)
       res += 1;
   return res;
}
```

Methode extrahieren und verschieben

```
double Movie::getCharge(int days) {
    double res = 0;
    switch (getPriceCode()) {
    case Movie::REGULAR:
        res += 2;
        if (days > 2)
            res += (days - 2) * 1.5;
        break;
    case Movie::NEW_RELEASE:
        res += days * 3;
        break;
    case Movie::CHILDRENS:
        res += 1.5;
        if (days > 3)
            res += (days - 3) * 1.5;
    }
    return res;
}
```

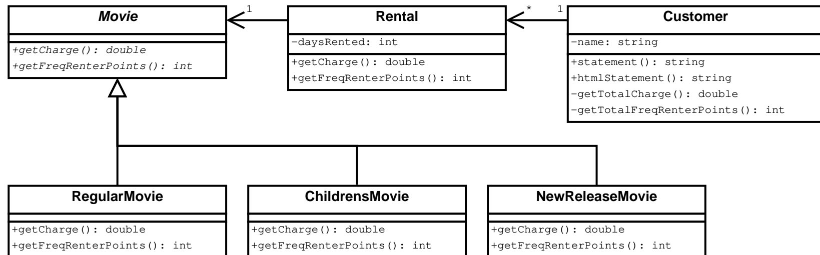
In der Klasse `Rental` ersetzen wir die `switch`-Befehle durch eine Delegation an das Objekt `_movie`:

```
double Rental::getCharge() {
    return _movie->getCharge(_daysRented);
}

int Rental::getFrequentRenterPoints() {
    return _movie->getFrequentRenterPoints(
                                _daysRented);
}
```

Ersetze Bedingung durch Polymorphie

Zu guter Letzt ersetzen wir den `switch`-Befehl durch Polymorphismus. *Typenschlüssel durch Unterklasse ersetzen.*

- Bewege jeden Ast der Fallunterscheidung in eine überladene Methode einer Unterklasse.
- Mache die ursprüngliche Methode abstrakt.



Ist dieser Software-Entwurf korrekt?

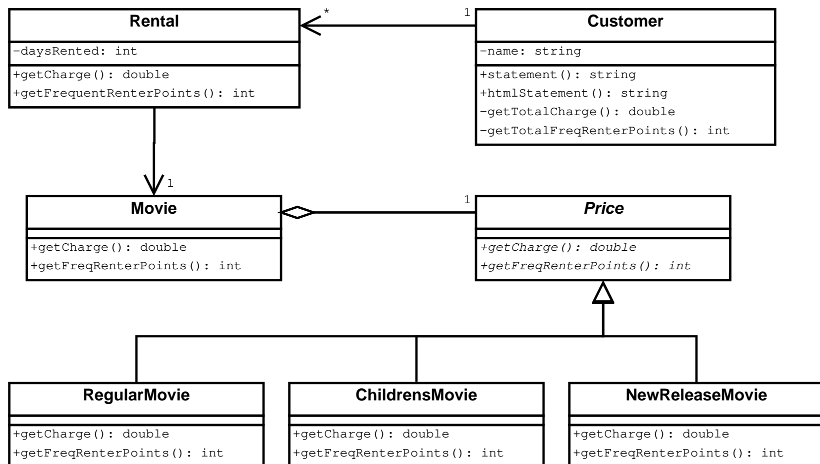
Ersetze Bedingung durch Polymorphie

Das war keine gute Idee! Denn ein Film kann seine Klassifizierung ändern: Aus einer Neuerscheinung wird ein regulärer Film.

- Wir dürfen nicht einfach Unterklassen von `Movie` bilden, sondern müssen das Entwurfsmuster *Strategie* oder *Zustand* anwenden.

$\rightarrow$ *Typenschlüssel durch Zustand/Strategie ersetzen*

Ersetze Bedingung durch Polymorphie



Ersetze Bedingung durch Polymorphie

Es gibt viel zu tun:

1. Verschiebe mittels *Typenschlüssel durch Zustand/Strategie ersetzen* das artabhängige Verhalten in das Zustandsmuster.
2. Verwende *Methode verschieben*, um den `switch`-Befehl in die Klasse `Price` zu verschieben.
3. Verwende *Bedingten Ausdruck durch Polymorphie ersetzen*, um den switch-Befehl zu entfernen.

Ersetze Bedingung durch Polymorphie

Eigenes Feld kapseln um sicherzustellen, dass alle Verwendungen durch get- und set-Methoden erfolgen.

```
class Movie {
private:
    string _title;
    Price *_price;                          // geaendert!
    .....
};
Movie::Movie(string title, int code) {
    _title = title;
    setPriceCode(code);                     // geaendert!
}
int Movie::getPriceCode() {
    return _price->getPriceCode(); // geaendert!
}
```

Ersetze Bedingung durch Polymorphie

```
// geaenderte Methode!
void Movie::setPriceCode(int arg) {
    switch(arg) {

    // jeweils Singleton zuweisen
    case REGULAR:
        _price = RegularPrice::exemplar();
        break;
    case CHILDRENS:
        _price = ChildrensPrice::exemplar();
        break;
    case NEW_RELEASE:
        _price = NewReleasePrice::exemplar();
        break;
    default:
        throw "incorrect price code";
    }
}
```

Ersetze Bedingung durch Polymorphie

```
class Price {
public:
    virtual int getPriceCode() = 0;
};

// Singleton-Klasse
class RegularPrice : public Price {
private:
    RegularPrice();
    static RegularPrice *_exemplar;

public:
    static RegularPrice *exemplar();
    int getPriceCode() {
        return Movie::REGULAR;
    }
};
.....
```

Ersetze Bedingung durch Polymorphie

Verwende *Methode verschieben*, um `getCharge()` von der Klasse `Movie` in die Klasse `Price` zu verschieben:

```
double Movie::getCharge(int days) {
    return _price->getCharge(days);
}

class Price {
public:
    virtual int getPriceCode() = 0;
    double Price::getCharge(int days);
};
```

Ersetze Bedingung durch Polymorphie

```
double Price::getCharge(int days) {
    double res = 0;
    switch (getPriceCode()) {
    case Movie::REGULAR:
        res += 2;
        if (days > 2)
            res += (days - 2) * 1.5;
        break;
    case Movie::NEW_RELEASE:
        res += days * 3;
        break;
    case Movie::CHILDRENS:
        res += 1.5;
        if (days > 3)
            res += (days - 3) * 1.5;
    }
    return res;
}
```

Jetzt können wir endlich *Bedingten Ausdruck durch Polymorphismus ersetzen* anwenden:

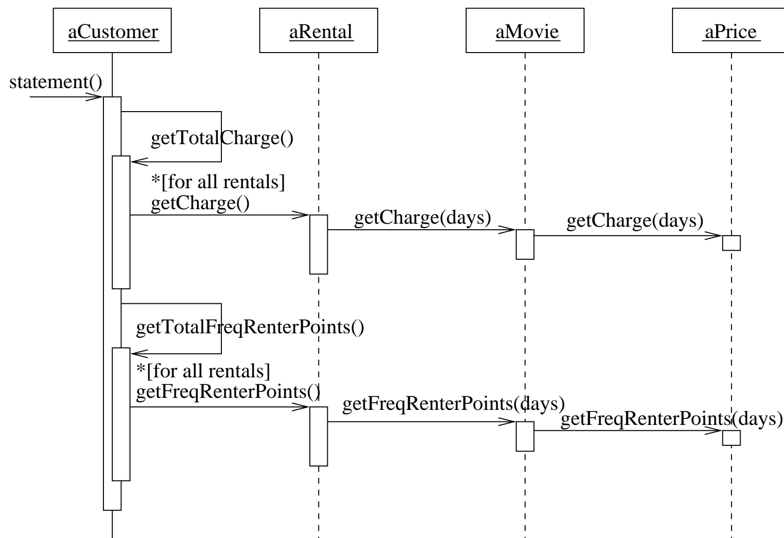
```
class Price {
public:
    virtual int getPriceCode() = 0;
    virtual double getCharge(int days) = 0;
};

double RegularPrice::getCharge(int days) {
    double res = 2;
    if (days > 2)
        res += (days - 2) * 1.5;
    return res;
}
```

Ersetze Bedingung durch Polymorphie

```
double ChildrensPrice::getCharge(int days) {
    double res = 1.5;
    if (days > 3)
        res += (days - 3) * 1.5;
    return res;
}

double NewReleasePrice::getCharge(int days) {
    return days * 3;
}
```

Danach wenden wir dasselbe Verfahren auf die Methode `getFrequentRenterPoints()` an.

Restrukturierte Applikation



Restrukturieren

Warum?

- Um das Design der Software zu verbessern.
- Um die Software leichter verständlich zu machen.
- Weil es hilft, Fehler zu finden.
- Weil man so schneller programmieren kann.

Wann?

- Beim Hinzufügen von Funktionen.
- Wenn Fehler zu beheben sind.
- Bei Code-Reviews.

Voraussetzungen:

- Automatisierte Tests und Änderung in kleinen Schritten.
- Einsatz von Entwurfswerkzeugen, um einfach Änderungen am Entwurf zu ermöglichen.
- Verwenden von Dokumentationswerkzeugen, um die Dokumentation auf dem neuesten Stand zu halten.
- Einsatz einer Versionsverwaltung zum Einspielen alter Versionen.
- Gute Kommunikation, damit alle im Team über Änderungen informiert sind.

Übersicht

Software-Reengineering

- Motivation und Begriffe
- Software-Qualität und -Metriken
- Refactoring am Beispiel
- *Katalog von Refactoring-Mustern*

Restrukturieren

Was? Übel riechender Code:

- duplizierter Code
- lange Methode
- große Klasse
- lange Parameterliste
- Neid, Fremdgänger
- Datenklumpen
- `switch`-Befehle
- Nachrichtenketten
- Vermittler
- und weitere ...

Duplizierter Code

Problem: Wenn sich Anforderungen ändern, müssen alle duplizierten Stellen konsistent geändert werden.

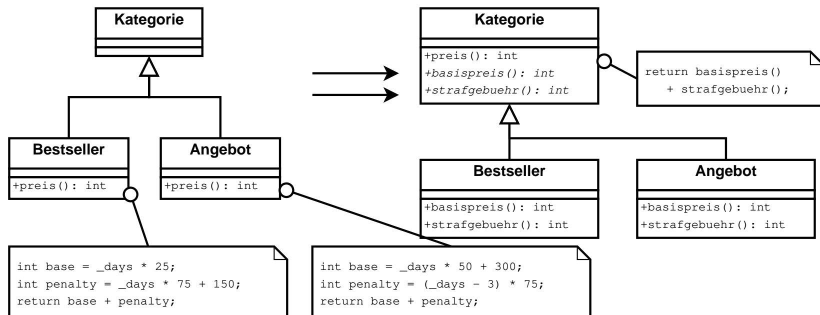
Lösung: Stelle sicher, dass der Code nur einmal da ist und nirgendwo sonst.

- gleicher Ausdruck in zwei Methoden einer Klasse
  - ⇒ *Methode extrahieren* anwenden und die neue Methode an beiden Stellen aufrufen
- gleicher Ausdruck in zwei verschwisterten Unterklassen
  - ⇒ *Methode extrahieren* in beiden Klassen anwenden und anschließend *Feld nach oben verschieben*

Zum Auffinden solcher duplizierter Stellen benötigen wir Tool-Unterstützung: Clone Detective, Clone Digger, CCFinder, Simian, usw.

Duplizierter Code

- ähnlicher Ausdruck in zwei verschwisterten Unterklassen
  - ⇒ *Methode extrahieren* anwenden, um Gemeinsamkeiten von den Unterschieden zu trennen
    danach *Template-Methode bilden* oder *Algorithmus ersetzen*



Duplizierter Code

- gleicher Code in voneinander unabhängigen Klassen

$\Rightarrow$ *Klasse extrahieren* auf eine Klasse anwenden und die neue Komponente in der anderen Klasse verwenden.

oder: wenn die Methode tatsächlich in eine der Klassen gehört $\rightarrow$ Methode von der anderen Klasse aufrufen

oder: wenn die Methode in eine dritte Klasse gehört $\rightarrow$ Methode von beiden Ausgangsklassen ansprechen

Lange Methode

Problem: Lange Methoden sind schwer zu verstehen und schlecht wiederverwendbar.

Lösung: Finde zusammengehörende Teile einer Methode und wende *Methode extrahieren* an.

Vorteil: Kurze Methoden können fast wie eine Folge von Kommentaren gelesen werden und lassen sich einfacher überschreiben.

> Schreiben Sie eine Methode, wenn Sie das Gefühl haben, sonst etwas kommentieren zu müssen.

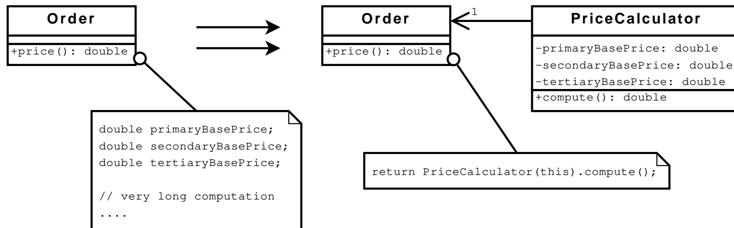
Lange Methode

Eventuell müssen viele Parameter und temporäre Variablen als Parameter an die neue Funktion übergeben werden, was zu schlecht lesbarem Code führt.

- Mittels *Temporäre Variable durch Abfrage ersetzen* die temporären Variablen entfernen → weniger Übergabeparameter
- Lange Parameterlisten durch *Parameterobjekt einführen* oder *Ganzes Objekt übergeben* verschlanken.
- *Methode durch Methodenobjekt ersetzen* (siehe nächste Folie)

Lange Methode

Methode durch Methodenobjekt ersetzen:



- anschließend sind alle lokalen Variablen entfernt

→ wir können die Methode `compute()` zerlegen, ohne irgendwelche Parameter übergeben zu müssen

Lange Methode

Wie identifiziert man den zu extrahierenden Codeklumpen?

- Suche nach Kommentaren: Ein Codeblock mit einem Kommentar kann durch eine Methode ersetzt werden, deren Name auf dem Kommentar basiert.
- Es lohnt sich sogar, eine einzelne Zeile zu entfernen, falls sie erläutert werden muss.
- Bei komplexen Bedingungen wende *Bedingung zerlegen* an.
- Schleifen: Extrahiere die Schleife und den Code innerhalb der Schleife in eine eigene Methode.

Bedingung zerlegen:

vorher:

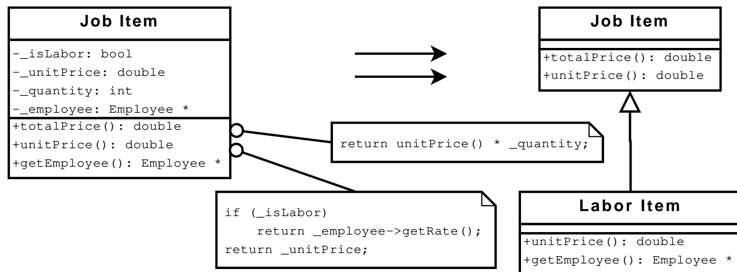
```
if (date.before(SUMMER_START)
        || date.after(SUMMER_END))
    charge = quantity * _winterRate
                + _winterServiceCharge;
else charge = quantity * _summerRate;
```

nachher:

```
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge(quantity);
```

Problem: Große Klassen sind schwer zu verstehen und oft eine Brutstätte für duplizierten Code.

Lösung: In kleinere Klassen aufspalten mittels *Klasse extrahieren* bzw. *Unterklasse extrahieren*, siehe unten.



Lange Parameterliste

Problem: Lange Parameterlisten sind schwierig zu benutzen und schwer zu verstehen.

Lösung:

- *Ganzes Objekt übergeben*, falls die Parameter zu einem Objekt gehören, oder
- *Parameterobjekt einführen*, falls die Daten aus verschiedenen Objekten stammen.

Neid, Fremdgänger

Problem: Eine Methode benutzt mehr Dienste einer anderen Klasse als der eigenen.

Lösung:

- *Methode verschieben* als Ganzes oder
- falls nur ein Teil der Methode unter Neid leidet, *Methode extrahieren* und *Methode verschieben* anwenden.

Datenklumpen

Problem: Dieselben Datenelemente treten als Attribute in verschiedenen Klassen und als Parameter in verschiedenen Methoden auf.

Lösung: Den Klumpen mittels *Klasse extrahieren* in ein Objekt wandeln, danach *Ganzes Objekt übergeben* oder *Parameterobjekt einführen* auf Parameterlisten anwenden.

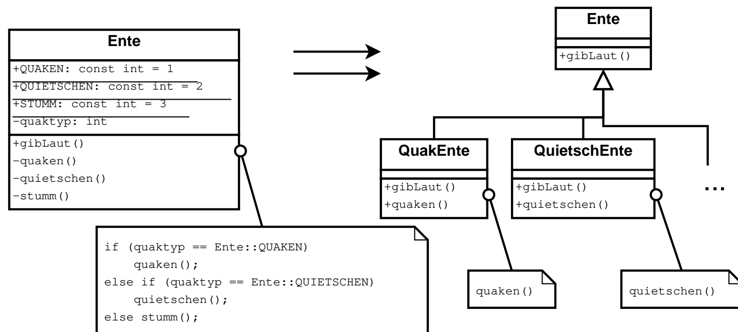
`switch`-Befehle

Problem: Meist Zeichen mangelnder Objektorientierung. Die gleichen `switch`-Befehle stehen oft an verschiedenen Programmstellen.

Lösung: Polymorphismus anwenden.

1. *Methode extrahieren*, um den `switch`-Befehl herauszuziehen
2. *Methode verschieben*, um sie in eine Klasse zu bekommen, in der Polymorphie genutzt werden kann.
3. *Typenschlüssel durch Zustand/Strategie ersetzen* oder *Typenschlüssel durch Unterklasse ersetzen* (nächste Folie)

Typenschlüssel durch Unterklasse ersetzen: Der Wert des Typschlüssels wird geprüft und abhängig davon wird verschiedener Code ausgeführt.



Nachrichtenketten

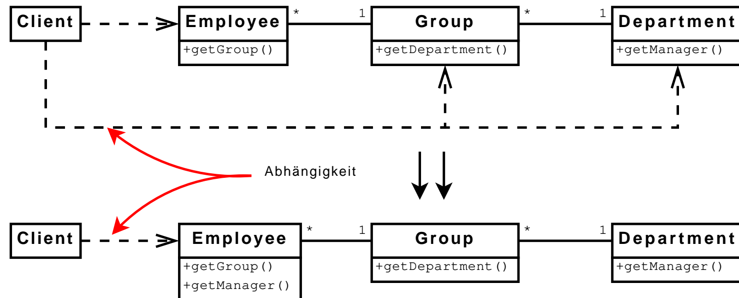
Problem: Bei Ketten wie

```
john.getGroup().getDepartment().getManager()
```

ist der Client zu stark mit der Struktur der Navigation gekoppelt. Änderungen am Client werden notwendig, wenn sich die Struktur ändert.

Lösung: *Delegation verbergen* durch abkürzende Methoden wie `john.getManager()`.

Delegation verbergen:



Vermittler

Problem: zu viele Abkürzungsmethoden wie

```
john.getManager()
```

Lösung: *Vermittler entfernen*

Kommentare

Obwohl Kommentare ein süßer Duft sind und kein übler Geruch, deuten sie doch auf ein Problem hin.

Problem: Oft Hinweis auf schwer verständlichen Code.

Lösung: *Methode extrahieren*, *Methode umbenennen*, *Zusicherung einführen*.

Zusicherungen sind

- Mittel zur Kommunikation: Sie helfen dem Leser zu verstehen, welche Annahmen der Code macht.
- hilfreich bei Fehlersuche: Fehler werden näher an Ihrem Entstehungsort entdeckt.

Zusicherung einführen

In C++ wird dazu `#include <cassert>` benötigt.

vorher:

```
string Date::toString() {
    // _df should not be NULL
    _df->format(this);
}
```

nachher:

```
string Date::toString() {
    assert(_df != NULL);
    _df->format(this);
}
```

Übersicht

Anwendung: Fallende Steine

- Spielidee
- Klasse `Date`
- Klasse `Player`
- Klasse `Highscore`
- Klasse `Position`
- Klasse `Brick`
- Klasse `Board`
- Logik

Spielidee

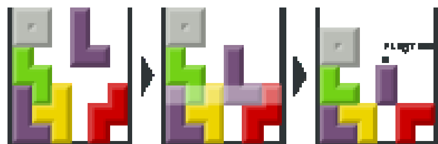
Wir wollen ein Spiel *Fallende Steine* mit objekt-orientierten Methoden implementieren. Die Spielelogik ist recht einfach:

- Am oberen Spielfeldrand werden nacheinander Steine eingebracht, die automatisch nach unten fallen.
- Mittels Abfrage der Tastatur wird festgestellt, ob der aktuelle Stein nach links oder rechts bewegt oder ob der Stein gedreht werden soll. Falls kein anderer Stein oder der Spielfeldrand im Weg sind, wird die Operation ausgeführt.
- Ist der Stein auf einen anderen Stein gefallen, so wird geprüft, ob eine Zeile vollständig besetzt ist. In diesem Fall wird die Zeile gelöscht, alle darüber befindlichen Zeilen werden eine Position nach unten verschoben, und der Punktestand wird erhöht. Ein neuer Stein wird oben im Spielfeld eingefügt.
- Das Spiel ist beendet, wenn oben kein neuer Stein eingefügt werden kann, weil bereits zu viele Zeilen gefüllt sind.

Spielidee

Die Steine verhalten sich physikalisch nicht korrekt:

- Die Formen bleiben in der Position liegen, in der sie landen, statt eventuell physikalisch korrekt zu kippen.
- Die nachrückenden Reihen füllen keine vorher vorhandenen Lücken auf. Auf diese Weise können Steine das Vervollständigen darunterliegender Reihen erschweren.



Quelle: `http://de.wikipedia.org/wiki/Tetris`

- Für ein richtiges Spiel benötigen wir eine Liste mit den besten, bisher erzielten Ergebnissen. → Klasse `Highscore`
- Ein Spieler wird characterisiert durch seinen Namen, seine erzielte Punktzahl und dem Datum, an dem diese Punktzahl erreicht wurde. → Klasse `Player`
- Um nicht mit den C-Funktionen `time()` oder `localtime()` arbeiten zu müssen, schreiben wir eine Wrapper-Klasse `Date`.
- Es gibt 7 verschiedene Spielsteine, die mit den Buchstaben I, J, L, O, S, T oder Z gekennzeichnet werden. Jeder Spielstein besteht aus 4 Teilen.

Quelle: `http://de.wikipedia.org/wiki/Tetris`

Anforderungsanalyse

- Das Spielfeld ist rechteckig und besteht aus 22 Zeilen und 13 Spalten.
- Da die Steine beim Löschen von gefüllten Zeilen nicht mehr in ihrer ursprünglichen Form erhalten bleiben, speichern wir an jeder Stelle des Spielfelds die ID des Steintyps, um bei der Darstellung eine farbliche Unterscheidung herstellen zu können.
- Mit den auf der Tastatur befindlichen Pfeil-Tasten $\leftarrow$ und $\rightarrow$ werden die Steine nach links bzw. rechts gelenkt, mittels der Leertaste wird der Stein um 90° rotiert und mittels 'q' wird das Spiel vorzeitig beendet.

Definition der Farben

```
#define BLACK_FG     1
#define GREEN_FG     2
#define WHITE_FG     3
#define RED_FG       4
#define CYAN_FG      5
#define MAGENTA_FG   6
#define BLUE_FG      7
#define YELLOW_FG    8

#define BLACK_BG     9
#define GREEN_BG    10
#define WHITE_BG    11
#define RED_BG      12
#define CYAN_BG     13
#define MAGENTA_BG  14
#define BLUE_BG     15
#define YELLOW_BG   16
```

colors.hpp

Klasse Date

date.hpp

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

class Date {
    friend std::istream&
    operator>>(std::istream& is, Date &d);
    friend std::ostream&
    operator<<(std::ostream& os, const Date &d);

protected:
    int _day, _month, _year;
    std::string toString(void) const;

public:
    Date(int d = 0, int m = 0, int y = 0);
    static Date getCurrentDate(void);
    bool operator<(const Date &d) const;
};
```

date.cpp

```
#include "date.hpp"
#include <sstream>
#include <iomanip>
using namespace std;

istream& operator>>(istream& is, Date &d) {
   char tmp;

   is >> d._day >> tmp
      >> d._month >> tmp
      >> d._year;
   return is;
}

ostream& operator<<(ostream& os, const Date& d){
   os << d.toString();
   return os;
}
```

Klasse Date

```
Date::Date(int day, int month, int year) {
    _day = day;
    _month = month;
    _year = year;
}

Date Date::getCurrentDate(void) {
    time_t now = time(0);
    tm *today = localtime(&now);

    int day = today->tm_mday;
    int month = today->tm_mon + 1;
    int year = today->tm_year + 1900;

    return Date(day, month, year);
}
```

```
string Date::toString(void) const {
    ostringstream os;

    os << setfill('0') << setw(2) << _day << "."
        << setfill('0') << setw(2) << _month
        << "." << _year;

    return os.str();
}

bool Date::operator<(const Date &d) const {
    int d1 = 10000 * _year + 100 * _month + _day;
    int d2 = 10000 * d._year + 100 * d._month +
            d._day;

    return d1 < d2;
}
```

Klasse Player

player.hpp

```
#include <string>
#include "date.hpp"

struct Player {
    std::string _name;
    int _score;
    Date _date;

    Player(std::string name, int score);
    Player(std::string name, int score,
           Date date);
    bool operator<(const Player& p) const;
};
```

player.cpp

```
Player::Player(string name, int score) {
    _name = name;
    _score = score;
    _date = Date::getCurrentDate();
}

Player::Player(string name, int score,
               Date date) {
    _name = name;
    _score = score;
    _date = date;
}

bool Player::operator<(const Player& p) const {
    return _score < p._score ||
           _score == p._score && _date < p._date;
}
```

highscore.hpp

```
#include <curses.h>
#include <string>
#include <set>
#include "player.hpp"

class Highscore {
private:
   std::string _filename;
   std::multiset<Player> _best;

public:
   void load(std::string filename);
   void insert(Player p);
   void toScreen(WINDOW *win);
   int minScore(void);
   void save(void);
};
```

highscore.cpp

```
#include "colors.hpp"
#include "highscore.hpp"
#include <fstream>
using namespace std;

int Highscore::minScore(void) {
   if (_best.size() < 10)
      return 0;

   multiset<Player>::iterator iter;
   iter = _best.begin();

   return iter->_score;
}
```

Klasse Highscore

```
void Highscore::insert(Player p) {
    _best.insert(p);

    if (_best.size() > 10) {
        multiset<Player>::iterator iter;

        iter = _best.begin();
        _best.erase(iter);
    }
}
```

Klasse Highscore

```
void Highscore::load(string filename) {
   _filename = filename;

   ifstream f(filename.c_str());
   if (!f)
      throw "could not open file";

   do {
      string name;
      int score;
      Date d;

      f >> name >> score >> d;
      if (!f.eof())
         _best.insert(Player(name, score, d));
   } while (!f.eof());
   f.close();
}
```

Klasse Highscore

```
void Highscore::save(void) {
    ofstream f(_filename.c_str());
    if (!f)
        throw "could not open file";

    multiset<Player>::iterator iter;

    for (iter = _best.begin();
           iter != _best.end(); iter++) {
        f << iter->_name << " ";
        f << iter->_score << " ";
        f << iter->_date << endl;
    }
    f.close();
}
```

Klasse Highscore

```
void Highscore::toScreen(WINDOW *win) {
    wbkgd(win, COLOR_PAIR(RED_BG));
    multiset<Player>::reverse_iterator iter;

    int row = 5;
    for (iter = _best.rbegin();
            iter != _best.rend(); iter++, row++) {
        mvwprintw(win, row, 2, "%8s: %6d  %s",
            (iter->_name).c_str(),
            iter->_score,
            iter->_date.toString().c_str());
    }
    mvwprintw(win, row + 2, 5,
            "press any key to continue");
    wrefresh(win);
    timeout(-1);
    getch();
}
```

position.hpp

```
struct Position {
   int _row, _col;

   Position(int r = 0, int c = 0);

   void down(void);
   void left(void);
   void right(void);

   Position operator+(Position &p) const;
};
```

```
Position::Position(int r, int c) {
    _row = r;
    _col = c;
}

void Position::down(void) {
    _row += 1;
}
void Position::left(void) {
    _col -= 1;
}
void Position::right(void) {
    _col += 1;
}

Position Position::operator+(Position &p) const{
   return Position(_row +p._row, _col + p._col);
}
```

position.cpp

Klasse Brick

brick.hpp

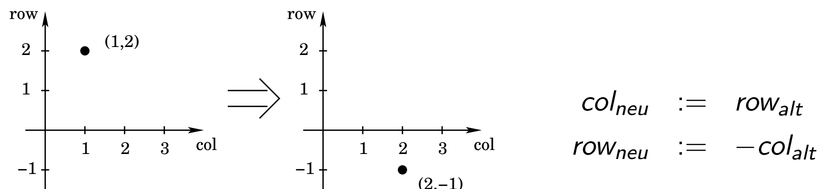
```
#include "position.hpp"

class Brick {
protected:
    Position _part[4];
    Position _pos;
    int _id;

public:
    const static int ID_I = 0;
    const static int ID_J = 1;
    const static int ID_L = 2;
    const static int ID_S = 3;
    const static int ID_Z = 4;
    const static int ID_O = 5;
    const static int ID_T = 6;
```

Klasse Brick

```
    Brick(int type);
    Brick(const Brick *b);

    int id(void);
    static int getColor(int id);

    void setPos(Position p);
    Position getPosOfPart(int i);

    void rotate(void);
    void fall(void);
    void left(void);
    void right(void);
};
```

Klasse Brick

Bevor wir uns die Implementierung der Klasse `Brick` ansehen, müssen wir uns überlegen, wie eine Rechtsrotation um 90° durchgeführt werden kann:



Wir müssen also für alle Teile eines Spielsteins die folgende Transformation durchführen:

```
int t = _part[i]._row;
_part[i]._row = - _part[i]._col;
_part[i]._col = t;
```

Klasse Brick

brick.cpp

```
Brick::Brick(int type) {
    _id = type;

    switch(type) {
    case ID_I:
        _part[0] = Position(-1, 0);   // #
        _part[1] = Position( 0, 0);   // *
        _part[2] = Position( 1, 0);   // #
        _part[3] = Position( 2, 0);   // #
         break;
    case ID_J:
        _part[0] = Position(-1, 0);   //  #
        _part[1] = Position( 0, 0);   //  *
        _part[2] = Position( 1, 0);   // ##
        _part[3] = Position( 1,-1);
        break;
```

Klasse Brick

```
case ID_L:
    _part[0] = Position(-1, 0);   // #
    _part[1] = Position( 0, 0);   // *
    _part[2] = Position( 1, 0);   // ##
    _part[3] = Position( 1, 1);
    break;
case ID_S:
    _part[0] = Position( 0, 0);   //  *#
    _part[1] = Position( 0, 1);   // ##
    _part[3] = Position( 1,-1);
    _part[2] = Position( 1, 0);
    break;
case ID_Z:
    _part[0] = Position( 0,-1);   // #*
    _part[1] = Position( 0, 0);   //  ##
    _part[2] = Position( 1, 0);
    _part[3] = Position( 1, 1);
    break;
```

Klasse Brick

```
    case ID_O:
        _part[0] = Position( 0,-1);  // #*
        _part[1] = Position( 0, 0);  // ##
        _part[2] = Position( 1,-1);
        _part[3] = Position( 1, 0);
        break;
    case ID_T:
        _part[0] = Position(-1,-1);  // ###
        _part[1] = Position(-1, 0);  //  *
        _part[2] = Position(-1, 1);
        _part[3] = Position( 0, 0);
        break;
    }
}
```

Klasse Brick

```
Brick::Brick(const Brick *b) {
    _id = b->_id;
    _part[0] = b->_part[0];
    _part[1] = b->_part[1];
    _part[2] = b->_part[2];
    _part[3] = b->_part[3];
    _pos = b->_pos;
}

int Brick::id(void) {
    return _id;
}
```

```
int Brick::getColor(int id) {
    int color = 0;

    switch (id) {
    case ID_I: color = RED_BG; break;
    case ID_J: color = YELLOW_BG; break;
    case ID_L: color = BLUE_BG; break;
    case ID_S: color = GREEN_BG; break;
    case ID_Z: color = MAGENTA_BG; break;
    case ID_O: color = WHITE_BG; break;
    case ID_T: color = CYAN_BG; break;
    }
    return color;
}
```

Klasse Brick

```
void Brick::setPos(Position p) {
    _pos = p;
}

Position Brick::getPosOfPart(int i) {
    if (i < 0 || i >= 4)
        throw "out of bounds";

    return _part[i] + _pos;
}

void Brick::rotate(void) {
    for (int i = 0; i < 4; i++) {
        int tmp = _part[i]._row;
        _part[i]._row = - _part[i]._col;
        _part[i]._col = tmp;
    }
}
```

Klasse Brick

```
void Brick::fall(void) {
    _pos.down();
}

void Brick::left(void) {
    _pos.left();
}

void Brick::right(void) {
    _pos.right();
}
```

Klasse Board

board.hpp

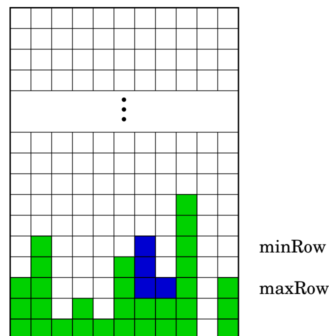
```
#include <curses.h>
#include "brick.hpp"

#define ROWS  22
#define COLS  13

class Board {
   char _board[ROWS][COLS];
   int _minRow, _maxRow, _score;
   Brick *_brick;
   WINDOW *_win;

   void removeLine(int line);
   bool isValid(Brick *b);
   void removeBrickFromBoard(void);
   void insertBrickToBoard(void);
   bool filledRow(int row);
   void eraseRow(int row);
```

Klasse Board

```
public:
    static const int DOWN = 0;
    static const int LEFT = 1;
    static const int RIGHT = 2;
    static const int ROTATE = 3;

    Board(WINDOW *win);
    void reset(void);
    bool isFinal(void);

    bool step(int operation);
    void toScreen(void);
    void tidy(void);
    void newBrick(void);
    int getScore(void);
};
```

Klasse Board

Bevor wir uns die Implementierung des Spielfeldes ansehen, müssen wir noch klären, welchen Sinn die Attribute `_minRow` und `_maxRow` haben.



Es können sich nur dort vollständige Zeilen befinden, wo der Spielstein stecken geblieben ist.

Das Spiel ist beendet, wenn `_minRow` kleiner oder gleich 1 ist, denn ein neu einzufügender Stein hat mindestens die Höhe 2.

Klasse Board

board.cpp

```
#include "colors.hpp"
#include "board.hpp"
#include "brick.hpp"
#include "position.hpp"
#include <cstdlib>

#define MIN(A,B)  (A) < (B) ? (A) : (B)
#define MAX(A,B)  (A) > (B) ? (A) : (B)

Board::Board(WINDOW *win) {
    _brick = nullptr;
    _win = win;
    reset();
}

int Board::getScore(void) {
    return _score;
}
```

```
void Board::reset(void) {
   if (_brick != nullptr)
      delete _brick;

   int r = rand() % 7;
   _brick = new Brick(r);
   if (r == Brick::ID_S || r == Brick::ID_Z ||
          r == Brick::ID_O)
      _brick->setPos(Position(0, COLS / 2));
   else _brick->setPos(Position(1, COLS / 2));

   _score = 0;
   _minRow = ROWS;
   _maxRow = 0;
   for (int r = 0; r < ROWS; r++)
      for (int c = 0; c < COLS; c++)
         _board[r][c] = ' ';
}
```

Klasse Board

```
bool Board::isFinal(void) {
    return _minRow <= 1;
}

void Board::removeBrickFromBoard(void) {
    for (int i = 0; i < 4; i++) {
        Position p = _brick->getPosOfPart(i);

        _board[p._row][p._col] = ' ';
    }
}

void Board::insertBrickToBoard(void) {
    for (int i = 0; i < 4; i++) {
        Position p = _brick->getPosOfPart(i);
        _board[p._row][p._col] = _brick->id();
    }
}
```

Klasse Board

```
bool Board::isValid(Brick *b) {
   for (int i = 0; i < 4; i++) {
      Position pos = b->getPosOfPart(i);

      if (pos._col < 0 || pos._col >= COLS ||
            pos._row >= ROWS)
         return false;

      if (_board[pos._row][pos._col] != ' ')
         return false;
   }
   return true;
}
```

Klasse Board

```
bool Board::filledRow(int row) {
   int cnt = 0;

   for (int c = 0; c < COLS; c++)
      if (_board[row][c] != ' ')
         cnt += 1;

   return cnt == COLS;
}

void Board::eraseRow(int row) {
   for (int r = row - 1; r >= 0; r--) {
      for (int c = 0; c < COLS; c++)
         _board[r+1][c] = _board[r][c];
   }
   _score += 50;
}
```

```
void Board::tidy(void) {
    // recalculate minRow and maxRow
    _minRow = ROWS;
    _maxRow = 0;
    for (int i = 0; i < 4; i++) {
        Position pos = _brick->getPosOfPart(i);
        _maxRow = MAX(_maxRow, pos._row);
        _minRow = MIN(_minRow, pos._row);
    }

    // filled rows?
    int r = _maxRow;
    while (r >= _minRow) {
        if (filledRow(r))
            eraseRow(r);
        else r -= 1;
    }
}
```

Klasse Board

```
void Board::newBrick(void) {
    int r = rand() % 7;

    _brick = new Brick(r);
    if (r == Brick::ID_S || r == Brick::ID_Z ||
            r == Brick::ID_O)
        _brick->setPos(Position(0, COLS / 2));
    else _brick->setPos(Position(1, COLS / 2));

    insertBrickToBoard();
}
```

```
bool Board::step(int operation) {
   Brick *b = new Brick(_brick);

   removeBrickFromBoard();
   switch (operation) {
      case Board::DOWN:   b->fall();   break;
      case Board::LEFT:   b->left();   break;
      case Board::RIGHT:  b->right();  break;
      case Board::ROTATE: b->rotate(); break;
   }
   bool moved;
   if (moved = isValid(b)) {
      delete _brick;
      _brick = b;
   } else delete b;
   insertBrickToBoard();
   return moved;
}
```

```
void Board::toScreen(void) {
    wbkgd(_win, COLOR_PAIR(BLACK_BG));

    for (int r = 0; r < ROWS; r++) {
        for (int c = 0; c < COLS; c++) {
            int type = _board[r][c];
            int color = Brick::getColor(type);

            if (color != 0)
                wattron(_win, COLOR_PAIR(color));
            mvwaddch(_win, r, c, ' ');
            if (color != 0)
                wattroff(_win, COLOR_PAIR(color));
        }
    }
    mvprintw(4, COLS + 5, "points: %d", _score);
    wrefresh(_win);
}
```

```
// ************************************
// Logik des Spiels "Fallende Steine"
// ************************************

#include <curses.h>
#include "brick.hpp"
#include "board.hpp"
#include "highscore.hpp"
#include "colors.hpp"
#include <ctime>
#include <cstdlib>
using namespace std;

int main(void) {
    initscr();
    keypad(stdscr, TRUE);
    curs_set(0);
    srand((unsigned int)time(0));
```

Logik

```
start_color();

init_pair(BLACK_BG,
          COLOR_WHITE, COLOR_BLACK);
init_pair(GREEN_BG,
          COLOR_WHITE, COLOR_GREEN);
init_pair(WHITE_BG,
          COLOR_WHITE, COLOR_WHITE);
init_pair(RED_BG,
          COLOR_WHITE, COLOR_RED);
init_pair(CYAN_BG,
          COLOR_WHITE, COLOR_CYAN);
init_pair(MAGENTA_BG,
          COLOR_WHITE, COLOR_MAGENTA);
init_pair(BLUE_BG,
          COLOR_WHITE, COLOR_BLUE);
init_pair(YELLOW_BG,
          COLOR_WHITE, COLOR_YELLOW);
```

Logik

```
init_pair(BLACK_FG,
          COLOR_BLACK, COLOR_BLACK);
init_pair(GREEN_FG,
          COLOR_GREEN, COLOR_BLACK);
init_pair(WHITE_FG,
          COLOR_WHITE, COLOR_BLACK);
init_pair(RED_FG,
          COLOR_RED, COLOR_BLACK);
init_pair(CYAN_FG,
          COLOR_CYAN, COLOR_BLACK);
init_pair(MAGENTA_FG,
          COLOR_MAGENTA, COLOR_BLACK);
init_pair(BLUE_FG,
          COLOR_BLUE, COLOR_BLACK);
init_pair(YELLOW_FG,
          COLOR_YELLOW, COLOR_BLACK);
```

Logik

```
int ch;
bool quit = false;
char name[50];

bkgd(COLOR_PAIR(YELLOW_BG));
WINDOW *win = newwin(ROWS, COLS, 1, 1);

Board game(win);
Highscore bestScores;

try {
    bestScores.load("scores.txt");
} catch(const char *) {}

erase();
mvprintw(3, 5, "Bitte Name eingeben: ");
mvscanw(4, 5, "%s", name);
```

```
noecho();

do {    // outer loop: another game
   erase();
   timeout(250);

   do {    // inner loop: one game
      game.toScreen();
      ch = getch();
      if (ch == KEY_LEFT)
         game.step(Board::LEFT);
      else if (ch == KEY_RIGHT)
         game.step(Board::RIGHT);
      else if (ch == ' ')
         game.step(Board::ROTATE);
      else if (ch == 'q')
         quit = true;
```

Logik

```
        bool moved = game.step(Board::DOWN);
        if (!moved) {
            game.toScreen();
            game.tidy();
            game.newBrick();
        }

        if (game.isFinal())
            quit = true;
    } while (!quit);  // end inner loop

    if (game.isFinal()) {
        erase();
        mvprintw(10, 4, "Game lost!");
        refresh();
```

Logik

```
// show high score if necessary
if (game.getScore() >
      bestScores.minScore()) {

   bestScores.insert(
      Player(name, game.getScore()));
   WINDOW *nwin = newwin(30, 35, 1, 1);
   bestScores.toScreen(nwin);
   delwin(nwin);
}
```

```
            erase();
            mvprintw(13, 4, "Another game? (y/n)");
            timeout(-1);
            ch = getch();
            if (ch == 'y') {
                game.reset();
                quit = false;
            }
        } // end if (game.isFinal())
    } while (!quit); // end outer loop

    bestScores.save();

    endwin();
    return 0;
}
```

Erweiterungen

Das Spiel lässt sich natürlich noch erweitern:

- Der nächste Stein wird in einem Vorschau-Fenster angezeigt.
- Es gibt verschiedene Level mit unterschiedlichen Spielfeldern.
- Die Geschwindigkeit, mit der ein Stein fällt, kann von Spieler beeinflusst werden, z.B. mittels der $\downarrow$-Taste.
- Je mehr Punkte erreicht wurden, umso schneller fallen die Steine.
- ...