C/C++ Materialpaket (Level C)
05a_OO_INH – Object Orientation (Inheritance)

Prof. Dr. Carsten Link

Zusammenfassung

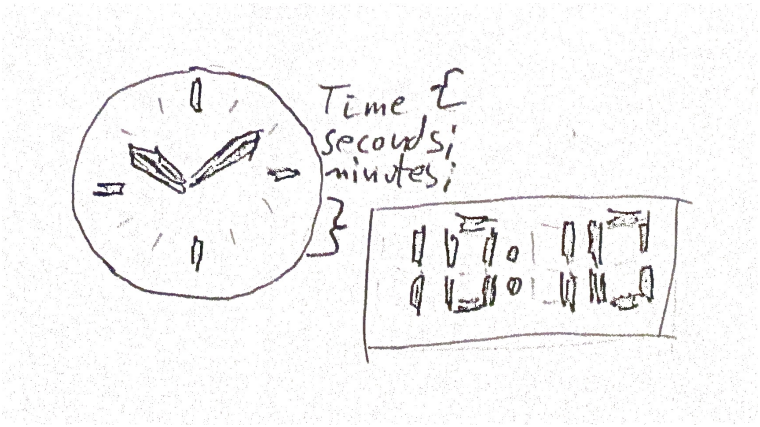

Abbildung 1: Objektorientierung; insbes. Vererbung

Inhaltsverzeichnis

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten und Fertigkeiten zur Problemlösung):

Sie können Programme zu gegebenen Problemstellungen unter Beachtung der Ideen der Objektorientierten Programmierung (OOP) gestalten und implementieren.

Die oben genannten Kompetenzen erwerben Sie, indem Sie Lernziele erreichen, welche sich prüfen lassen. Lernegebnisse: Sie können nachweislich[1]:

- die Begriffe Klasse und Objekt erläutern
- die wesentlichen Ideen der objektorientierten Programmierung umsetzen
- C++-Sprachmittel zur objektorientierten Programmierung einsetzen:
  - Klassen
  - Vererbung von Struktur und Verhalten (interface inheritance, implementation inheritance)
  - Methoden
  - Methoden: virtual und pure virtual
  - Konstruktoren
  - Objekte im free store (heap) anlegen und nach Verwendung wieder löschen

2 Konzepte

Im Folgenden werden die Grundideen der objektorientierten Programmierung vorgestellt. Ebenso wird das dabei wichtige Konzept der Vererbung erläutert.

2.1 Objektorientierte Programmierung

Bei der objektorientierten Programmierung steht bei der Modellierung von Programmen das *Objekt* im Vordergrund. Ein Objekt besteht aus Zustand (Daten) und Verhalten (Code), wobei der Zustand durch eingebaute Datentypen (`int, double, ...`) oder wiederum Objekte dargestellt wird. Der Code eines Objekts verwendet die aus C bekannten Funktionsaufrufe, Kontrollstrukturen (`if, for, while, ...`), Anweisungen usw. und darüber hinaus Methodenaufrufe auf Objekten.

Die wesentlichen Ideen der objektorientierten Programmierung sind:

- Das Objekt ist das Konstrukt mit dem die Entitäten (Dinge etc.), das Zusammenspiel und die Sachverhalte des Problems, das von einem Programm gelöst/berechnet werden soll, modelliert werden

[1] Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen.

- Ein Objekt hat Zustand (Variablen) und Verhalten (Methoden) und kann eine Identität[2] haben
- Kapselung bzw. Trennung von Interface und Implementierung: kein direkter Zugriff auf Daten eines Objektes. Dies sorgt dafür, dass benutzender Code von der tatsächlichen Implementierungsweise innerhalb des Objektes unabhängig ist; diese kann geändert werden, ohne dass der benutzende Code geändert werden muss.
- Wiederverwendung: Objekte können von verschiedenem benutzendem Code (client code) verwendet werden
- Vererbung: Objekte können andere Objekte auf zwei Arten verwenden:
  1. indem sie innerhalb ihrer Implementierung (Methodenrümpfe) andere Objekte benutzen
  2. indem Sie Daten und Verhalten von einer *Basisklasse erben*. Hierbei kann das Verhalten unverändert geerbt (übernommen) werden, oder modifiziert werden. Bzgl. der Daten können nur weitere Felder hinzukommen; geerbte Daten lassen sich nicht entfernen.

Bei reinen objektorientierten Programmiersprachen wie beispielsweise Smalltalk genügt das Abstraktionsmittel Objekt; bei C++ kommen Klassen hinzu: sie sind Deklarationen und Definitionen, die dem Compiler vorgeben, wie Objekte (welche ausschließlich zur Laufzeit existieren) realisiert werden sollen.

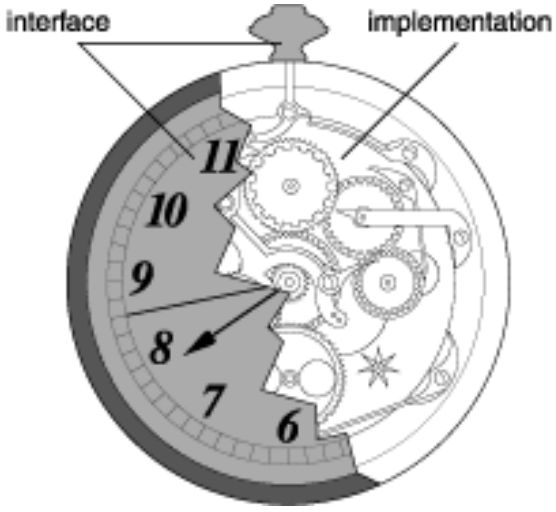

Abbildung 2: Interface vs. Implementation. Quelle: NeXT Manual OOP, NeXT Computer, Inc. 1995 und Apple[4]

[2] Der Zustand (z.B. die Farbe) läßt sich ändern, ohne die Identität zu ändern.

[4] https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/OOP_ObjC/Articles/ooOOP.html#//apple_ref/doc/uid/TP40005149-CH8-SW1

2.2 C++: Vom benuterdefinierten Datentyp zur Klasse

Bei benutzerdefinierten Datentypen, die bereits vorgestellt wurden, steht die Programmierung mit der Standardbibliothek im Vordergrund. Es sind `structs`, also zusammengesetzte Datentypen, welche insbesondere durch Operatorenüberladung in der selben Art und Weise verwendet werden können, wie eingebaute Datentypen.

Bei Klassen steht das *Objekt* im Vordergrund, das dynamische Programmiermittel der objektorientierten Programmierung.

2.3 Vererbung in C++

Das Konzept der Vererbung soll anhand von drei Klassen erklärt werden: `Vehicle`, `Car` und `Truck`. Autos und LKW sind Fahrzeuge. Diese drei Begriffe hängen zusammen – es gibt Gemeinsamkeiten und Unterschiede. Zunächst die Klassendefinitionen:

```
1 class Vehicle {
2 protected:
3     int _numSeats;
4 public:
5     Vehicle(int numSeats);
6     virtual int payload() = 0;
7     int numSeats();         // a 'getter' method to get a value; no 'setter' here
8 };
```

Die Klasse `Vehicle` bündelt die Gemeinsamkeiten von allen Fahrzeugarten: sie haben eine Anzahl von Sitzen und ein Zuladungsgewicht. Die Anzahl der Sitze wird in dieser Basisklasse `Vehicle` verwaltet. Das Zuladungsgewicht lässt sich zwar über die Schnittstelle abfragen; berechnen lässt es sich hier auf dieser abstrakteren Eben nicht – dazu fehlt Information, die erst in abgeleiteten Klassen zur Verfügung steht. Die Methode `payload()` ist

- `virtual`: kann in einer abgeleiteten Klasse überschrieben (neudefiniert) werden, und
- `pure virtual`: ist in der abstrakten Basisklasse `Vehicle` lediglich deklariert. Es fehlt eine Definition (angezeigt durch `= 0`).

Die Schlüsselworte `public:`, `protected:` und `private` dienen dem Zugriffsschutz, der vom Compiler durchgesetzt wird. Der wesentliche Zweck ist es, klare Trennlinien der bzgl. der Schnittstellen zu ziehen. Code einer Klasse hat Zugriff auf alle Member (Felder und Methoden) seiner eigenen Klasse und alle `protected`-Member der Basisklassen. Von fremden Klassen dürfen nur `public`-Member benutzt werden.

Heißt also: eine Klasse stellt ihren `public:`-Teil allen anderen Klassen und Funktionen zur Verfügung; der `protected:`-Teil ist abgeleiteten Klassen vorbehalten; der `private:`-Teil geht niemand anderen etwas an.

Ohne Angabe eine Zugriffsschutzes sind Member einer Klasse `private`; Member von `structs` sind ohne Angabe `public`.

Hier die erste abgeleitete Klasse:

```
class Truck : public Vehicle {
protected:
    int _payload;
    double _volume;
public:
    Truck(int numSeats, int payload, double volume);
    virtual int payload();
};
```

Die Klasse `Truck` verwaltet Zuladungsgewicht im Attribut `_payload`. Das Attribut `_numSeats` ist in Methoden der abgeleiteten Klasse `Truck` verwendbar, da es geerbt wird (`class Truck : public Vehicle`). Die obige Klassendefinition führt die Methode (member function) `payload()` ein.

Hier die zweite abgeleitete Klasse:

```
class Car : public Vehicle {
protected:
    int _maxWeight;     // zulässiges Gesamtgewicht
public:
    Car(int numSeats, int maxWeight);
    virtual int payload();

};
```

Die Klasse `Car` verwaltet das Zuladungsgewicht anders: es wird das zulässige Gesamtgewicht abzüglich des Leergewichtes gespeichert.

Zur Verdeutlichung des unterschiedlichen Verhaltens der Klassen `Car` und `Truck` sind hier die beiden Implementierungen der Methode `payload()` gegeben:

```
int Truck::payload(){
    return _payload;
}
```

hingegen:

```
int Car::payload(){
    return _maxWeight - (numSeats()*75); // subtract 75kg per person
}
```

Die Verwendung von zwei dieser `Vehicle`-Objekte (also Objekte von Klassen, welche von `Vehicle` abgeleitet sind) sieht wie folgt aus:

```
int main(int argc, const char * argv[]) {
    Car* c   = new Car(5, 1000);    // create a new object of class Car in free store
```

```
    Truck* t = new Truck(3, 7500, 80000.0);

    println("1 --------- ");
    println("c: numSeats=", c->numSeats(), " payload=", c->payload());
    println("t: numSeats=", t->numSeats(), " payload=", t->payload());
    println("");

    println("2 --------- ");
    Vehicle* v = nullptr;
    v = c;                      // a Car `is a` Vehicle => implicitly convertible
    printVehicleInfo(v);        // println(v); does work, too
    v = t;                      // a Truck `is a` Vehicle => implicitly convertible
    printVehicleInfo(v);        // println(v); does work, too

    // release memory occupied by t,c for use by future objects created by `new`
    // do NOT release v  -- it is only an alias
    delete t;
    delete c;
    return 0;
}
}
```

Wie oben zu sehen ist, erfolgt der Zugriff auf Methoden (und ggf. member variables) mittels des Pfeiloperators `->`.

Der Effekt von virtuellen Methoden wird deutlich an den Zeilen nach `// invoke via pointer to base class`. Dort die Variable `v` den Typ `Vehicle*` – also Pointer auf Basisklasse. Da ein `Car`-Objekt auch ein `Vehicle`-Objekt ist, kann ein Zeiger vom Compiler implizit von `Car*` in `Vehicle*` umgewandelt werden[5]. Daher kann die Funktion `printVehicleInfo()` auf dem Typ `Vehicle*` arbeiten:

```
void printVehicleInfo(Vehicle* v){
    println("typeid=`", typeid(*v).name(), "`",
            " numSeats=", v->numSeats(), // invoke via pointer to base class
            " payload=", v->payload());  // invoke via pointer to base class
}
```

Die Funktion `printVehicleInfo()` setzt auf einer recht hohen Abstraktionsstufe an. Sie kennt die Typen `Car` und `Truck` nicht. Dennoch kann sie auf Objekten dieser beiden Typen arbeiten. Insbesondere kann die Funktion `printVehicleInfo()` auch auf Objekte von Klassen arbeiten, die zu ihrem Erstellungszeitpunkt noch gar nicht existieren; Klassen also, welche irgendwann in der Zukunft von Programmierern von `Vehicle` abgeleitet werden.

Die Ausgabe des Programms ist:

[5] so wie ein `int` in ein `double` oder ein `int` in ein `BinaryOctet`, sofern der `operator int()` definiert ist

```
1 ---------
c: numSeats=5 payload=625
t: numSeats=3 payload=7500

2 ---------
typeid=`3Car` numSeats=5 payload=625
typeid=`5Truck` numSeats=3 payload=7500
```

2.4 Erzeugung von Objekten

Objekte des Typs `T` werden mit `new T` erzeugt[6]. Dadurch wird durch den Compiler im Zusammenspiel mit den Bibliotheken dynamischer Speicher angefordert (*free store*, *heap*). Der frisch allokierte Speicher ist in der Regel mit zufälligen Werten gefüllt, daher wird vom Compiler zusätzlich ein Konstruktoraufruf generiert, der diesen rohen Speicher zu einem wohlgeformten Objekt des angeforderten Typs – also einer Ausprägung (engl. instance) des Typs / der Klasse `T` – macht.

```
Vehicle::Vehicle(int numSeats){
    _numSeats = numSeats;
}
```

Ein Konstruktor hat per Konvention den Namen `<Klassenname>::<Klassenname>`. Oben ist ersichtlich, wie der Konstruktor `Vehicle::Vehicle()` den frischen Speicher initialisiert, indem dem einzigen Feld `_numSeats` ein Wert zugewiesen wird.

Mit der Deklaration der Methode `Vehicle(int numSeats=0);` (Signatur innerhalb der Klassendeklaration von `Vehicle`) wurde dem Compiler ein *default argument* mitgeteilt: an Stellen, an denen im Code kein Argument übergeben wird, setzt der Compiler das Argument `0` ein. Als Nebeneffekt wird dadurch dieser Konstruktor zum *default constructor*, da ja nun kein Argument mehr zur Konstruktion nötig ist.

Die Konstruktoren der beiden abgeleiteten Klassen verfahren analog dazu:

```
Car::Car(int numSeats, int maxWeight){
    _numSeats = numSeats;
    _maxWeight = maxWeight;
}
```

```
Truck::Truck(int numSeats, int payload, double volume){
    _numSeats = numSeats;
```

[6] In diesem C++-Kurs werden Objekte ausschließlich im free store platziert. Globale, lokale und temporäre Objekte (Ausdrücke, Übergabe von Parametern und Rückgabewerten von Funktionen) auf dem Heap sind nicht zulässig. Instanzen von benutzerdefinierten Datentypen unterliegen nicht dieser Einschränkung und können daher auf dem Stack allokiert werden. Allerdings dürfen Instanzen von benutzerdefinierten Datentypen keine Zeiger enthalten (auch nicht auf Objekte)

```
    _payload = payload;
    _volume = volume;
}
```

Hinweis: auf die Zerstörung von Objekten mittels Destruktoren wird im nächsten Materialpaket eingegangen. Hier ist es ausreichend, Objekte mit `delete` zu löschen, so dass der Compiler-generierte Destruktor aufgerufen wird und der vom Objekte belegte Speicher freigegeben werden kann.

3 Material zum aktiven Lernen

Regelmäßiger Hinweis: Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im Folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (C/C++Programm) erstellt, welches dann auf mehrere Arten modifiziert wird. Insbesondere die Modifikationen ermöglichen es dem Lernenden (und auch dem Lehrenden), die Qualität des Kompetenzerwerbs bzgl. dieses Materialpakets bewerten zu können.

3.1 Aufgabe: Grundgerüst

Achten Sie darauf, bei den Projekteinstellung den C++17-Standard zu aktivieren (`clang++ -std=c++17 -Wall -Wextra -Wpedantic -O0`

Nehmen Sie den folgenden Code als Ausgangspunkt:

```
#include <iostream>
#include <cmath>
#include <vector>
#include "AnsiConsole.h"

struct Position{
  int x;
  int y;
  Position(int x_=0, int y_=0){ x=x_;y=y_;}
};

class Point{
protected:
  Position _position;
public:
  Point(int x=0, int y=0);
  void draw();
};

Point::Point(int x, int y){
  _position = Position(x,y);
```

```
22 }
23
24 void Point::draw(){
25   ansiConsole.printText(_position.x,_position.y,"*", Colors::RED);
26 }
27
28 class Circle{
29 protected:
30   Position  _position;
31   int       _radius;
32 public:
33   Circle(int x, int y, int radius);
34   void draw();
35 };
36
37 Circle::Circle(int x, int y, int radius){
38   _position = Position(x,y);
39   _radius=radius;
40 }
41
42 void Circle::draw(){
43   /* see https://de.wikibooks.org/wiki/Formelsammlung_Mathematik:_Geometrie
44    * Höhensatz des Euklid
45    * */
46   int x_start = _position.x - _radius/2;
47   int x_stop  = _position.x + _radius/2;
48
49   for(int i=x_start; i<=x_stop; i++){
50     double x_relative = double(i) - double(x_start);
51     double h = sqrt(x_relative*(x_stop-x_start-x_relative));
52     ansiConsole.printText(_position.x + int(x_relative)- _radius/2,
53                           _position.y +h,"#",
54                           Colors::GREEN);
55     ansiConsole.printText(_position.x + int(x_relative)- _radius/2,
56                           _position.y -h,"#",
57                           Colors::GREEN);
58
59   }
60 }
61
62 int main(int argc, char **argv)
63 {
64   // x=1 and y=1 is the upper left corner
65   // x and y are more like column and row
66   ansiConsole.printText(5,5,"Hello, World!");
```

```
67
68    Circle* c = new Circle(30, 15, 10);
69    c->draw();
70
71    std::vector<Point *> manyPoints;
72    Point* p = new Point(10,10);
73    manyPoints.push_back(p);
74
75    Point* p2 = new Point(2,10);
76    manyPoints.push_back(p2);
77
78    manyPoints.push_back(new Point(30,15));
79
80    // let's draw the points!
81    for (int i=0; i<manyPoints.size();i++){
82      manyPoints[i]->draw();
83    }
84
85
86    // delete unused objects
87    for (int i=0; i<manyPoints.size();i++){
88      delete manyPoints[i];
89    }
90
91    delete c;
92
93    return 0;
94  }
```

Hinweis: Das obige Programm verwendet die Klasse `AnsiConsole`, um Ausgaben zu machen. Mit Hilfe dieser Klasse ist es möglich, textuelle Ausgaben an vorgegeben Stellen und in Farbe zu machen. Dies ist möglich, da sich die Textfenster (Terminals) unter Linux mit Steuersequenzen konfigurieren lassen (ANSI Escape Codes – siehe hierzu `AnsiConsoleDemo.cpp` und die Implementierung in `Ansiconsole.cpp`). Es ist wichtig, dass die verwendeten Koordinaten nicht außerhalb des sichtbaren Bereichs des Terminalfensters liegen (d.h.: verwenden Sie kleine Koordinatenwerte)

Erweiterungen des Grundgerüst:

1. Erstellen Sie eine Klasse Rectangle, welche analog zu `Circle` arbeitet (nicht rund, sondern eckig – also `_width` und `_height` statt `_radius`)
2. Fügen Sie zu den Klassen `Point`, `Circle` und `Rectangle` einen Member `_color` vom Typ `Colors` (definiert in `AnsiConsole.hpp`) hinzu, der jeweils im Konstruktor mit einem Wert versehen wird, welcher später zum Zeichnen verwendet wird.
3. Erstellen Sie eine Klasse `Scene`, welche mehrere Zeiger auf `Point`,

`Circle`, `Rectangle` enthalten kann (jeweils ein `vector<>` pro Typ) und diese nacheinander zeichnet (hinzfügen mit `addPoint(Point*)`, `addCircle(Circle*)` etc.; zeichnen mit `drawAll()`). Die Klasse `Scene` braucht nicht von `Shape` abgeleitet werden. Die Klasse `Scene` soll auch für das Löschen der `Shape`-Objekte zuständig sein. Fügen Sie noch keine gemeinsame Basisklasse hinzu!

4. Erzeugen Sie in `main()` ein Objekt der Klasse `Scene`, welches grob einen Schneemann, Schneeflocken und ein Geschenk (Karton) zeichnet.
5. Erstellen Sie eine gemeinsame Basisklasse `Shape` für die Klassen `Point`, `Circle` und `Rectangle`.
6. Modifizieren Sie Ihre Klasse `Scene`, so dass diese nur noch einen einzigen `vector<Shape*>` enthält, in dem sich alle zu zeichnenden Objekte befinden. Denken Sie daran, dass einige Methoden in `Shape` und den davon abgeleiteten Klassen nun `virtual` sein sollten (insbesondere der Destruktor `Shape::~Shape()`)

Wenn Sie die oben genannten Erweiterungen des Grundgerüstes umgesetzt haben, können Sie die Dateien für die Modifikationen verwenden.

3.2 Aufgabe: Modifikationen

Regelmäßiger Hinweis: Weiter unten ist eine Liste mit Modifikationen gegeben, die zwei Zwecken dienen: 1) Sie dienen als Richtschnur für das Praktizieren und Üben der Inhalte dieses Materialpakets. 2) Die Modifikationen können im Rahmen eines Testats als Aufgabe verwendet werden, durch deren Lösung Studierende nachweisen können, dass sie den Stoff dieses Materialpakets beherrschen. Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (ca. 5 - 10) vor Zuschauern (Testatsituation) umsetzen können. Konkret sollen Sie im Testat in der Lage sein, das gegebene Grundgerüst um mindestens eine zufällig ausgewählte Modifikation zu erweitern. Bereiten Sie dazu auf ihrer Arbeitsumgebung ein Verzeichnis vor, welches ausschließlich das Grundgerüst enthält. **Arbeiten Sie also auf einer Kopie des Verzeichnisses `src-cpp-student/`!** Achten Sie darauf, dass der Text auf Ihrem Bildschirm in heller Umgebung aus einem Meter Abstand heraus gut lesbar ist (light mode, große Schrift).

Modifikationen (bauen auf den Erweiterungen des Grundgerüstes auf):

1. Fügen Sie der Klasse `Shape` eine Methode `double area()` hinzu, welche den Flächeninhalt berechnet. Da in der Klasse Shape hierzu keine Informationen vorhanden sind, sollte die Methode hier pure virtual deklariert sein und erst in den abgeleiteten Klassen definiert werden
2. Erstellen Sie eine Klasse `Cross`, welche ein Kreuz darstellen kann (wie `+` oder `x`)
3. Verschieben Sie den Member `_position` und `_color` der abgeleiteten Klassen in die gemeinsame Basisklasse `Shape` (Level C: machen Sie diese

Member dort `private`)

4. (Level C) Fügen Sie eine Klasse `Line` hinzu (erbt von `Shape`), welche eine Linie zeichnet, sonst vom Verhalten aber den anderen `Shape`-Klassen entspricht. Verwenden Sie einen existierenden Algorithmus dazu (z. B. den Bresenham-Algorithmus)

3.3 Verständnisfragen

Nach Bearbeitung des Kapitels "Konzepte", der Erstellung des Grundgerüsts sowie dem Üben der Modifikationen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten.

1. Auf welche Methoden und Felder von anderen Klassen hat Code, der sich in einer anderen Klasse befindet, Zugriff?
2. Was sind die wesentlichen Unterschiede bzgl. des Lebenszyklus' von Objekten im free store (z.B. `Truck` oder `Circle`) und solchen, die nicht dort angesiedelt sind (z.B. lokale Variablen oder Membervariablen vom Typ `Position` oder `PascalString`)?
3. Welchen Zweck erfüllen Basisklassen, von denen andere Klassen abgeleitet sind?
4. Zu welchem Zeitpunkt wird entschieden, welche Methodenimplementierung aufgerufen werden soll, bei a) `virtual`-Methoden und b) nicht-`virtual`-Methoden.
5. Was passiert beim Aufruf der Funktion `void foo(Base*)` mit einem Zeiger auf ein Objekt der Klasse `Derived`? (Klasse `Derived` ist abgeleitet von `Base`)
6. Was passiert beim Aufruf der Funktion `void foo(Derived*)` mit einem Zeiger auf ein Objekt der Klasse `Base`? (Klasse `Derived` ist abgeleitet von `Base`)
7. Welche Aufgaben hat ein Konstruktor?
8. Welche Vorteile hat es, Code gegen eine Basisklasse (z.B. `Shape`) zu schreiben, statt gegen konkrete Klassen (z.B. `Rectangle, Circle`)?
9. Es wurde Code gegen eine Basisklasse (z.B. `Shape`) geschrieben, von der konkrete Klassen (z.B. `Rectangle, Circle`) abgeleitet sind. Muss der Code angepasst werden, falls in der Zukunft eine neue Klasse `Triangle` hinzukommt? Welche Quelldateien müssen neu übersetzt werden?
10. Freiwillig: Zeigen Sie auf, inwiefern das Piktogram auf der Titelseite dieses Materialpaketes den Inhalt zusammengefasst darstellt.

4 Nützliche Links

- Apple, Object-Oriented Programming with Objective-C, Object-Oriented Programming[7]

[7] https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/OOP_ObjC/Articles/ooOOP.html#//apple_ref/doc/uid/TP40005149-CH8-SW1

- Apple, Object-Oriented Programming with Objective-C, The Object Model[8]

5 Literatur

- [OOSC] Meyer, Bertrand: Object-oriented Software Construction
- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++
- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition

[8] https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/OOP_ObjC/Articles/ooObjectModel.html#//apple_ref/doc/uid/TP40005149-CH5-SW4