Objektorientierung entzaubert

Objektorientiert in prozeduralen Sprachen

Michael Elschner

hitforum

18. März 2008

Inhalt

1. Paradigmen
2. Objektorientierung
3. Perl
4. ANSI-C

Programmierparadigmen

Programmiersprachen lassen sich anhand gemeinsamer Eigenschaften verschiedenen Paradigmen zuordnen, z. B.:

- imperative Programmierung
  Programm als Folge einfacher Anweisungen
- strukturierte Programmierung
  Programmfluss wird durch Strukturen gesteuert; keine „wilden Sprünge“
- prozedurale Programmierung
  Anweisungen werden in Unterprogrammen gruppiert
- modulare Programmierung
  Anweisungen werden in Unterprogrammen gruppiert
- objektorientierte Programmierung
  Gruppierung von Anweisungen und Daten

Paradigmen – Eigenschaften der Sprache?

- DOS-Batches haben keine bedingte Schleifenanweisung (nur eine – eingeschränkte – `for`-Anweisung existiert)
- Java besitzt keine `goto`-Anweisung
- Dennoch kann man in DOS-Batches strukturiert programmieren und in Java völlig unstrukturiert.

Batch strukturiert

```
1 @ECHO OFF
2
3 :beginLoop
4 set /p prod=Was ergibt 6 mal 9?
5 if %prod%==54 echo Sicher?
6 if %prod%==42 goto endLoop
7 goto beginLoop
8 :endLoop
```

Java unstrukturiert

```
1  class Unstrukturiert {
2      public static void main(String argv[]) {
3          int i = 0;
4
5          for (;;) {
6              switch (i) {
7                  case  0: System.out.print("Was kommt "); i = 30; break;
8                  case 10: System.out.print("wenn man "); i = 50; break;
9                  case 30: System.out.print("dabei heraus, "); i = 10; break;
10                 case 50: System.out.print("sechs mit neun ");
11                 default: System.out.println("malnimmt?"); System.exit(0);
12             }
13         }
14     }
15 }
```

Paradigmen – Eigenschaften der Programmierweise!

- Es ist also keine Sprache per se strukturiert oder objektorientiert
- Es ist nur leichter, in bestimmten Sprachen in einem bestimmten Paradigma zu programmieren
- Natürlich schließen manche Sprache auch Paradigmen aus: Objektorientiertes Brainf*ck ist vermutlich nur schwer zu erreichen.

Was ist eigentlich Objektorientierung?

Grundlegende objektorientierte Konzepte sind:

- Kapselung
  Objekte werden nicht direkt manipuliert, sondern über eine definierte Schnittstelle.
- Polymorphie
  Verschiedene Objekte können die gleiche Schnittstelle besitzen; sie sind dadurch im gleichen Zusammenhang nutzbar.
- Vererbung
  Objekte können voneinander Attribute und Methoden *erben*.

Diese Begriffe werden im folgenden genauer erklärt.

Objekt

- Objekte sind die zentralen Elemente der objektorientierten Programmierung.
- Sämtliche Operationen werden auf bzw. von Objekten vorgenommen.
- Einander ähnliche Objekte werden in Klassen zusammengefasst.
- Wobei es auch – völlig unmarxistische – „klassenlose“ Sprachen gibt.

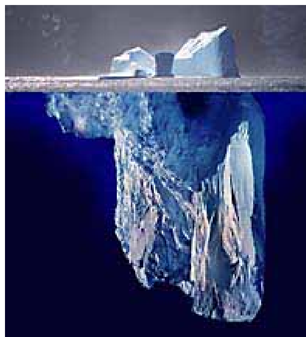
Klasse

Eine Klasse beschreibt die Struktur der ihr angehörenden Objekte.
Dazu gehören:

- Attribute, also die Daten die ein konkretes Objekt dieser Klasse beschreiben
- Methoden, also die Verhaltensweisen, die die Objekte dieser Klasse ausmachen
- Zusätzlich gibt es auch Methoden und Attribute, die die ganze Klasse betreffen.
- Eine besondere solche Klassenmethode ist der Konstruktur, der neue Objekte zurückgibt.

Die nach außen sichtbaren Methoden bilden dabei die Schnittstelle der Klasse. Alles nicht sichtbare ist in der Klasse gekapselt.

Kapselung



Eine gute Kapselung verhält sich wie ein Eisberg:

- nur ein kleiner Teil – die Schnittstelle – ist nach außen sichtbar
- der größte Teil – die Implementierung – bleibt verborgen

Polymorphie – In der realen Welt

Realwelt-Beispiele für polymorphe Schnittstellen:

- **Lampengewinde**
  In das gleiche Gewinde passen unterschiedlich geformte Glüh- und Energiesparlampen unterschiedlicher Leistung.
- **Steckdosen**
  Verschiedenste Elektrogeräte können an die gleiche Steckdose angeschlossen werden.
  (Oder auch nicht, dann werden Adaptoren gebraucht)

Der Schnittstellenvertrag

Der Begriff Schnittstelle schließt im objektorientierten Sinne auch das Verhalten (den Vertrag) mit ein:

- Eine Lampe muss nicht nur „passen“, sie muss auch leuchten, um den Vertrag der Schnittstelle „Lampengewinde“ einzuhalten.
- Entsprechend muss der Stecker eines Elektogreräts nicht nur „passen“, das Gerät muss auch arbeiten.

Dabei kann das „Leuchten“ bzw. das „Arbeiten“ frei ausgestaltet werden:

- Verschiedene Lampen leuchten unterschiedlich hell und in verschiedenen Farben.
- Eine Waschmaschine „arbeitet“ etwas gänzlich anderes als ein Kühlschrank.

Polymorphie – In der Programmierung

In die Programmierung bedeutet Polymorphie:

- Zuweisungskompatibilität
  einer Variablen können Objekte unterschiedlicher, kompatibler Typen zugewiesen werden
- späte Bindung
  Methodenaufrufe werden erst aufgelöst, wenn der konkreten Typ des Objektes bekannt ist – also zur Laufzeit

So ist es möglich, einen Kühlschrank oder eine Waschmaschine überall da zu verwenden, wo ein Elektrogerät erwartet wird und sie dennoch jeweils ihre Aufgabe erledigen zu lassen.

Vererbung

Vererbung bedeutet die Übertragung von Verhalten und Eigenschaften. Eine Klasse *A*, die von einer Klasse *B* erbt, lässt sich häufig sprachlich in eine „ist ein"-Beziehung zu dieser Klasse bringen:

- Eine Energiesparlampe *ist eine* Lampe.
  Alle Lampen verfügen über eine Leistung, eine Helligkeit, eine Farbe und können leuchten.
- Eine Waschmaschine „ist ein" Elektrogerät. Alle Elektrogeräte verfügen über eine Leistung und können arbeiten.
- Vielleicht ist eine Lampe auch ein Elektrogerät?
  VORSICHT: Die Schnittstelle (Gewinde vs. Schuko-Stecker) passt nicht!

Trennung Polymorphie und Vererbung

- Klassen, die von einer gemeinsamen anderen Klasse erben, sind immer auch polymorph zueinander.
- So kann ich z. B. überall, wo ein Elektrogerät erwartet wird, eine Waschmaschine oder einen Kühlschrank verwenden.
- Vererbung ist aber nicht unproblematisch, weil neben der Spezifikation (der Schnittstelle) auch immer die gesamte Implementierung an die erbenden Klassen weitergegeben wird.
- Sprache wie Java trennen deswegen zwischen der Vererbung der Implementierung und Vererbung der Spezifikation

Trennung Aggregation und Vererbung

- Objekte lassen sich aus anderen Objekten zusammensetzen.
- Im Gegensatz zur Vererbung („ist ein“) wird häufig von einer „hat ein“-Beziehung gesprochen.
- Dennoch lassen sie sich häufig gegeneinander austauschen:
  - Ein Kreis *ist ein* Punkt mit einem Radius.
  - Ein Kreis *hat einen* Mittelpunkt und einen Radius.
- Im späteren C-Beispiel werden wir sehen, warum Aggregation und Vererbung tatsächlich gar nicht so verschieden zueinander sind.

Objektorientierung in Perl 5

Perl 5 ist eine vollwertige objektorientierte Sprache.
Es besitzt jedoch keine eigene objektorientierte Syntax, stattdessen:

- ist ein Objekt eine Referenz
  - jede beliebige Referenz kann einer Klasse zugeordnet werden
  - von diesem Zeitpunkt an ist die Referenz ein Objekt
- ist eine Klasse ein Paket
  - die Subroutinen des Pakets sind die Methoden des Objektes bzw. der Klasse
  - die Paketvariablen sind die globalen Klassenattribute
- ist eine Methode eine Subroutine
  - beim Methodenaufruf wird ein zusätzlicher Parameter übergeben
  - Klassenmethoden bekommen die Klasse (das Paket) übergeben
  - Objektmethoden das Objekt (die Referenz)
  - Methoden können gleichzeitig Klassen- und Objektmethode sein

Sprachergänzungen - Syntax

Durch dieses „simple“ Modell werden nur kleine Sprachergänzungen benötigt:

- Eine „Mache-diese-Referenz-zu-einer-Instanz“-Anweisung:
  bless($obj, "Klasse")
- (Mindestens) eine Syntax für einen Methodenaufruf
  Perl stellt (typisch) deren gleich zwei zur Verfügung:

C++-like	**Smalltalk-like**
Paket−>methode($par1, $par2)	methode Paket $par1, $par2
$objekt−>methode($par1, $par2)	methode $objekt $par1, $par2

Beachte:

- Bei der „Smalltalk-Syntax“ steht kein Komma nach dem Paket- oder Objektnamen!
- Von ihr ist aber aufgrund syntaktischer Mehrdeutigkeiten generell abzuraten.

Die bless-Anweisung

- Die Anweisung **bless** legt die Verbindung Referenz->Klasse unwiderruflich fest.
- Der Name dieser Anweisung spiegelt wohl den verhinderten Missionar im Perl-Autor Larry Wall wider.
- Daher existiert auch keine Gegenstücksanweisung curse o. ä., mit der die Verbindung gelöst werden könnte.
- Allerdings gibt es eine Anweisung **sin**,
  welche jedoch nichts mit Objektorientierung zu tun hat. . .

Sprachergänzungen - Intern

Im internen Handling ist freilich einiges anders:

- Methodenaufrufe werden erst zur Laufzeit aufgelöst und nicht wie Prozeduraufrufe bereits zur Compile-Zeit
- Bei dieser Auflösung muss die Vererbung berücksichtigt werden: Die in der Paketvariable our @ISA; definierten Superklassen werden daher nach nicht direkt in der Klasse implementierten Methoden durchsucht.
- Aufruf von Superklassen-Methoden: durch SUPER::methode oder explizite Nennung.

Aufgaben Konstruktor in Perl

Die Aufgaben eines Konstruktors in Perl entsprechen denen in anderen Sprachen:

- Speicher reservieren
- Attribute initialisieren
- zusätzlich: Die Verbindung Referenz->Klasse herstellen

Der Konstruktor in Perl

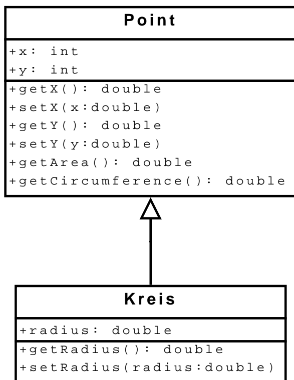
```perl
 1 sub new {
 2     my $class = shift;
 3     my $x = shift;
 4     my $y = shift;
 5
 6     # Speicher reservieren und Attribute initialisieren
 7     my $this = {
 8         x => $x,
 9         y => $y,
10     };
11
12     # Verbindung Referenz->Klasse herstellen
13     bless $this, $class;
14
15     # Objekt zurückliefern
16     return $this;
17 }
```

Anmerkungen zum Konstruktor

- die Syntax { . . . } erstellt in Perl einen *Hash* und gibt die Referenz auf ihn zurück
- Synonyme für *Hash*: assoziatives Array, Dictionary
- diese Hashreferenz wird zur „Objektreferenz" erklärt
- theoretisch können auch andere Referenzen genutzt werden z. B. auch Array- oder Skalarreferenzen
- Hashes bieten wegen der benannten Felder die größte Bequemlichkeit

Perl-Beispiel

Zum Testen wird eine denkbar einfache Klassenhiearchie gebildet:



Point

Circle

Testprogramm

Zusammenfassung

- Eine Klasse in Perl besteht aus einem Modul mit einer sub new, die eine Referenz zurückliefert, auf die bless angewendet worden ist.
- Eine Methode wird in der Form $var->example() aufgerufen. Sie erhält die „Objekt-Referenz“ als ersten Parameter.
- Vererbung erfolgt über das @ISA-Array, daher ist auch Mehrfachvererbung möglich.
- Polymorphie und Überschreiben von Methoden ist möglich, da in den Subklassen Methoden wieder definiert werden können.
- Mögliches Problem: Namensraumüberschneidungen bei Datenfeldern möglich
  Die Datenfelder der Subklasse liegen ja im gleichen Hash wie die der Superklasse.

Checkliste: Ist Perl objektorientiert?

Bzw. „Lässt sich in Perl objektorientiert programmieren?“

- Kapselung
  - Möglich über den Modul-Mechanismus.
  - Allerdings wird die Verwendung der Schnittstelle nicht erzwungen.
  - Attribute könnten von außen manipuliert werden.
  - „nicht-öffentliche“ Methoden könnten ebenfalls aufgerufen werden.
  - Zu starker Zwang wäre allerdings auch perl-untypisch:
    Wer sich in den Fuß schießen möchte, dem steht es frei. . .
- Polymorphie
  - Perl ist keine streng typisierte Sprache, Zuweisungskompatibilität besteht daher grundsätzlich immer.
  - Späte Bindung ist über das ISA-Array möglich.
- Vererbung
  - Für Methoden möglich über das ISA-Array.
  - Bei Attributen gibt es evtl. Probleme, da vererbende und erbende Klasse den gleichen Hash als Grundlage verwenden.
  - Diese Problematik kann aber effektiv gelöst werden.

In Perl kann also objektorientiert programmiert werden!

Exkurs: „Echte“ Kapselung in Perl

- „Echte“ private Variablen sind möglich, wenn man sie im jeweiligen Package mit `my` lokalisiert.
- Package-Variablen sind aber eigentlich Klassen-Variablen.
- Der Trick ist: Sie werden in einem Hash auf Basis der Objekt-Referenz abzulegen!
- Damit ist echte, lexikalische Privatheit garantiert!
- Übrigens ist das auch in C/C++ eigentlich der einzige Weg, auch dort „java-ähnliche“ Privatheit zu erzeugen.
  (Aufgrund der Möglichkeiten, dort mit Pointern in Objektreferenzen „herumzuwildern“).

Beispiel: „Echte“ Kapselung in Perl

```
1  package Object;
2
3  my %PRIVATE = ();
4
5  # Destruktor zum Loeschen der privaten Objektvariablen
6  sub DESTROY { delete $PRIVATE{shift}; }
7
8  sub new
9  {
10     my $class = shift;
11     my $self = {};
12     bless $self, $class;
13
14     $self->{LOGIN} = undef;
15     $PRIVATE{$self}->{PASSWORD} = undef;
16
17     return $self;
18 }
```

ANSI-C

- 1989 von der ANSI definierter Standard der Sprache C („C89“, „ANSI-C“).
  - Überarbeitungen der Festlegungen durch Kerninghan und Richie von 1979 (K&R-C)
  - 1999 von der ISO nochmals überarbeitet und ergänzt („C99“).
- Der Standard von 1989 ist allerdings der verbreitetste.
  - Verfügbarkeit für nahezu jede beliebige Plattform.
  - Vom Embedded System bis zum Mainframe.
  - Konkretes Beispiel: C89 mit Amendments und POSIX-Funktionen verfügbar für das Mainframe-Betriebssystem z/VSE.
- Schlägt in der Verbreitung und der Vollständigkeit der Implementierungen C++ um Längen!

Objektorientierung in ANSI-C

In ANSI-C kann objektorientiert programmiert werden!

Um dies zu demonstrieren, werden im die Punkte der folgenden Checkliste nach und nach behandelt und in C umgesetzt:

- Kapselung
- Polymorphie
- Vererbung

Kapselung

Kapselung wird von C praktisch nativ unterstützt:

- Quellcode kann auf verschiedene Quelltextdateien verteilt werden.
- Diese Dateien werden getrennt übersetzt und zum Programm dazugelinkt.
- Die Schnittstelle wird in der „Header“-Datei festgelegt.
- Alles, was mit `static` deklariert wurde, bleibt lokal zur Quelltextdatei.

Dieser Kapselungsmechanismus unterstützt bereits das Programmieren mit „Abstrakten Datentypen“.

Programmieren mit Abstrakten Datentypen

Programmieren mit Abstrakten Datentypen bedeutet:

- Kapselung von Datentypen und darauf anzuwendender Funktionen in einer Einheit.
- Die Implementierung des Datentyps bleibt vor den Benutzern verborgen.
- Nur über die Schnittstelle können Manipulationen am Abstrakten Datentyp vorgenommen werden.
- Programmieren mit Abstrakten Datentypen ist damit praktisch die erste Stufe zur Objektorientierung.

Der Abstrakte Datentyp Punkt

- Hier wird das Beispiel aus dem Perl-Kapitel wieder aufgegriffen.
- Die Klasse Punkt wird als abstrakter Datentyp in C umgesetzt, mit folgender Schnittstelle:

```
1  #ifndef POINT_H
2  #define POINT_H
3
4  void *PointNew(void);
5  void PointDelete(void *this);
6
7  double PointGetX(void *this);
8  void   PointSetX(void *this, double x);
9
10 double PointGetY(void *this);
11 void   PointSetY(void *this, double y);
12
13 double PointGetArea(void *this);
14 double PointGetCircumference(void *this);
15
16 #endif
```

Der Abstrakte Datentyp Punkt – Benutzung

Die Benutzung „fühlt“ sich bereits recht objektorientiert an:

```
 1 #include <stdio.h>
 2
 3 #include "Point.h"
 4
 5 int main(void)
 6 {
 7     void *pPoint = PointNew();
 8
 9     PointSetX(pPoint, 3.0);
10     PointSetY(pPoint, 4.0);
11
12     printf("Der␣Punkt␣liegt␣an␣der␣Position␣(%.0f|%.0f)␣"
13            "mit␣dem␣Durchmesser␣%.0f␣und␣der␣Fläche␣%.0f.\n",
14            PointGetX(pPoint), PointGetY(pPoint),
15            PointGetArea(pPoint), PointGetCircumference(pPoint) );
16
17     PointDelete(pPoint);
18 }
```

Der Abstrakte Datentyp Punkt – Implementierung

Konkret teilt sich das Ganze auf die folgenden Dateien auf:

Point.c

Point.h

test.c

Verwendung des void-Zeigers

Zum externen Zugriff auf den Abstrakten Datentyp wird ein void-Zeiger verwendet:

- void-Zeiger sind die universellen Zeiger in C, die auf „nichts bestimmtes“ zeigen
- Im konkreten Beispiel kann so der Aufbau des Abstrakten Datentyps vor dem Benutzer verborgen werden.
- Problematisch ist hier die geringere Typsicherheit:
  An die Funktionen des Datentyps Point könnte auch alles mögliche andere übergeben werden...

Spätestens für die Realisierung der Polymorphie erweist sich der Zeiger auf „nichts bestimmtes“ als sehr praktisch.

Polymorphie

- Wie können in C zueinander „kompatible" Datentypen erstellt werden, auf die sich die gleichen Operationen anwenden lassen?
- Die Antwort liegt im Aufbau der Strukturen und den Zusicherungen, die der C-Standard darüber macht:
  - Ein Zeiger auf eine Struktur zeigt immer gleichzeitig auf ihr erstes Element.
  - Die Elemente liegen im Speicher aufsteigend in der Reihenfolge ihrer Deklarationen.
  - Strukturen *A* und *B* mit den gleichen, in der gleichen Reihenfolge deklarierten, Elementtypen sind zueinander kompatibel.
  - Daraus lässt sich folgern, dass eine Struktur *B* in ihrem führenden Part zu einer Struktur *A* kompatibel ist, wenn die Elemente dieser Struktur *A* den führenden Elementen aus Struktur *B* entsprechen

Am besten zeige ich das mal am Beispiel ;-)

Polymorphie – ein Beispiel

- Man nehme zwei Strukturen `Point` und `Circle`, sowie zwei Variablen `pPoint` und `pCircle`:

```
1  typedef struct {
2      double x;
3      double y;
4  } Point;
5
6  typedef struct {
7      double x;
8      double y;
9      double radius;
10 } Circle;
11
12 Circle circle;
13
14 void *pPoint  = &circle;
15 void *pCircle = &circle;
```

- Dann können die Elemente `x` und `y` über jeden dieser Zeiger zugegriffen werden.

Der Abstrakte, Polymorphe Datentyp `Circle`

- Analog zum Perl-Beispiel wird der – gerade gezeigte – Datentyp `Circle` definiert, der den Datentyp `Point` erweitert.
- Auf ihn können alle Operationen angewendet werden, die auch auf `Point` anwendbar sind.
- Außerdem werden weitere Operationen definiert, außerdem wird ein anderes Verhalten für `GetArea` und `GetCircumference` vorgesehen.

```
 1 #ifndef CIRCLE_H
 2 #define CIRCLE_H
 3
 4 void *CircleNew(void);
 5 void CircleDelete(void *this);
 6
 7 double CircleGetRadius(void *this);
 8 void CircleSetRadius(void *this, double radius);
 9
10 double CircleGetArea(void *this);
11 double CircleGetCircumference(void *this);
12
13 #endif
```

Verwendung des Datentyps Circle

```
1  #include <stdio.h>
2
3  #include "Point.h"
4  #include "Circle.h"
5
6  int main(void)
7  {
8     void *pCircle;
9
10    pCircle = CircleNew();
11
12    PointSetX(pCircle, 3.0);
13    PointSetY(pCircle, 4.0);
14    CircleSetRadius(pCircle, 5.0);
15
16    printf("Der␣Kreis␣liegt␣an␣der␣Position␣(%.0f|%.0f)␣"
17           "mit␣dem␣Durchmesser␣%f␣und␣der␣Fläche␣%f.\n",
18           PointGetX(pPoint), PointGetY(pPoint),
19           CircleGetArea(pCircle), CircleGetCircumference(pCircle) );
20
21    printf("Als␣Punkt␣gesehen␣hat␣er␣aber␣nur␣den␣"
22           "Durchmesser␣%f␣und␣die␣Fläche␣%f.\n",
23           PointGetArea(pCircle), PointGetCircumference(pCircle) );
24
25    CircleDelete(pCircle);
26 }
```

Der Abstrakte Datentyp Circle – Implementierung

Hier die Dateien für den Datentyp Circle:

Circle.c

Circle.h

test.c

Vererbung

- Im Perl-Beispiel hat `Circle` von `Point` geerbt.
- Hier sind momentan die Typen `Circle` und `Point` völlig unabhängig voneinander.
- Man kann den Typ `Circle` aber auch etwas anders fassen:

```
1 typedef struct {
2     Point super;
3     double radius;
4 } Circle;
```

- Jetzt ist klar ersichtlich,
  dass `Circle` eine Erweiterung von `Point` ist.

Dynamische Methoden

- Wir haben also soeben Polymorphie und Vererbung in C entdeckt.
- Was noch nicht funktioniert, ist Dynamische Bindung,
  also die Auflösung von Methodenaufrufen zur Laufzeit,
  abhängig von der Klasse.
- Dazu muss natürlich zunächst einmal die Klasse zum Zeitpunkt des Aufrufs bekannt sein.
  Dann kann aus Informationen zu dieser Klasse auch tatsächlich zu nutzende Methode ermittelt werden.

Klassen in C

- Eine Klasse stellt allgemeingültige Informationen zu allen ihrer Objekte zur Verfügung.
- Das sind u. a. – und vor allem – die Methoden dieser Klasse, in C kann man sich das für `Point` folgendermaßen vorstellen:

```
 1 typedef struct {
 2     void    *(*new)(void);
 3     void     (*delete)(void *this);
 4     double   (*getX)(void *this);
 5     void     (*setX)(void *this, double x);
 6     double   (*getY)(void *this);
 7     void     (*setY)(void *this, double x);
 8     double   (*getArea)(void *this);
 9     double   (*getCircumference)(void *this);
10 } ClassPoint;
```

- Diese Zeiger auf die Methoden müssen einmalig initialisiert werden.

Vererbung von Klassen

- Die Klasse `Circle` muss ueber die Methoden von `Point` verfügen
  und kann zudem eigene Methoden einführen:

```
1 typedef struct {
2     ClassPoint super;
3     void    (*setRadius)(void *this, double radius);
4     double  (*getRadius)(void *this);
5 } ClassCircle;
```

- Werden die geerbten Zeiger anders gesetzt, haben wir polymorphe Methoden.

Dispatching der Methoden

- Jede Klasse hat jetzt ihre eigenen Methoden-Implementationen.
- Beim Aufruf einer Methode muss immer die passende Implementation gewählt werden:

```
1  void *new(void *class)
2  {
3      Object *this = ((ClassPoint *)class)->new();
4      this->class = class;
5      return this;
6  }
7
8  double getX(void *this)
9  {
10     return ((ClassPoint *)((Object *)this)->class)->getX(this);
11 }
12
13 void setX(void *this, double x)
14 {
15     ((ClassPoint *)((Object *)this)->class)->setX(this, x);
16 }
```

Checkliste: Ist C objektorientiert?

Bzw. „Lässt sich in C objektorientiert programmieren?“

- Kapselung
  - Möglich über verschiedene Dateien, Schnittstellen und Includes.
  - Verwendung der Schnittstelle kann (weitgehend) erzwungen werden;
    höhere Sicherheit hier geht aber auf Kosten der Typsicherheit bei der Übersetzung.
  - Über Zeiger-Manipulationen dennoch Umgehung der Schnittstelle möglich.
- Polymorphie
  - Möglich über generische `void`-Zeiger.
  - Späte Bindung ist – wenn auch etwas umständlich – möglich.
- Vererbung
  - Vererbung von Attributen und Methoden möglich.
  - Vererbung klar als Variante der Aggregation erkennbar.

In C kann also objektorientiert programmiert werden!

Realwelt-Beispiele

- Das Framework GObject des GNOME-Projektes.
- Die Bibliothek libwww des W3C.

Bücher, Arbeiten

American National Standard for Information Systems. *Programming Language C X3.159-1989*
1989

Axel-Tobias Schreiner. *Object-Oriented Programming With ANSI-C*
Hollage, 1993.

`http://openbook.galileocomputing.de/oo/index.htm`