Programmiertechnik 2

Unit 2: Objektorientierter Entwurf und Programmierung

Ablauf

- Programmieren im Großen
- Von Modulen zu Objekten
- OO Konzepte im Überblick
- Regeln für objektorientierten Entwurf
- Liskov Substitution Principle (LSP)
- Vererbung
- Polymorphie

Programmieren im Großen

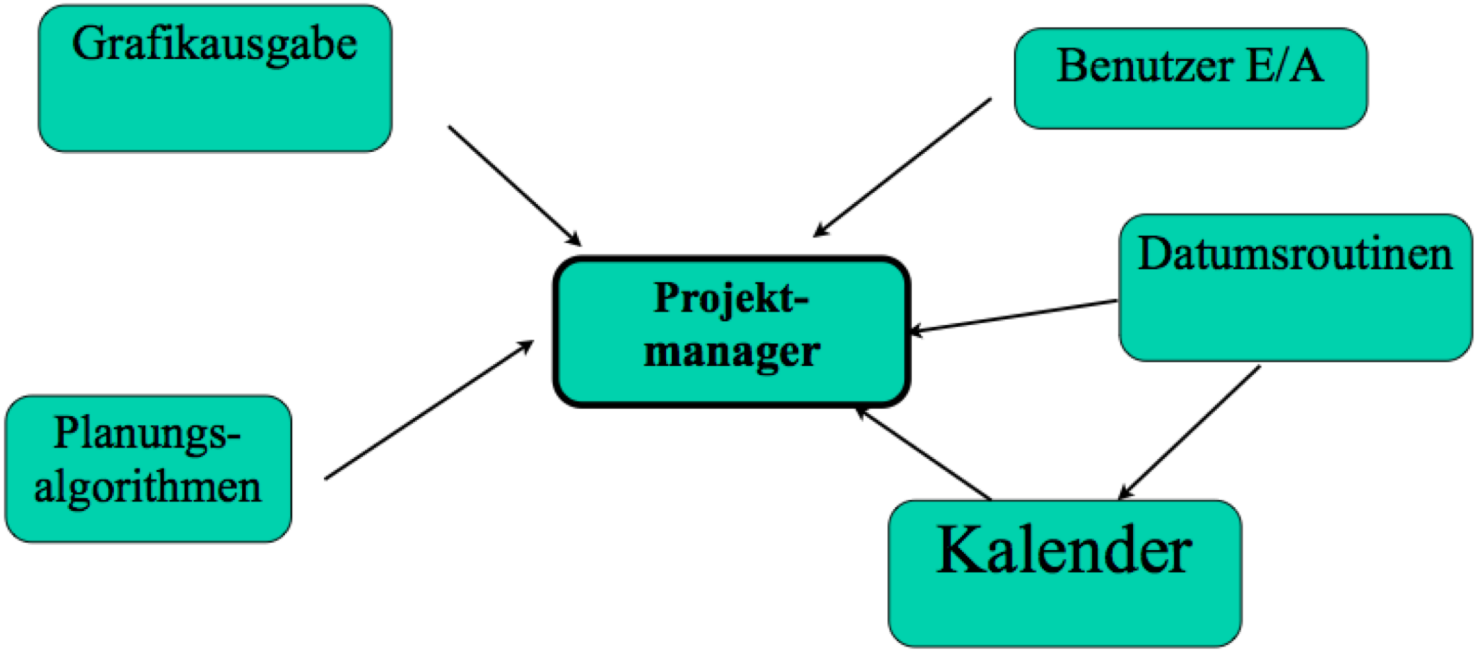
- Strukturierte Programmierung: Blockkonzept (ab 1960) – Algorithmen können in Teilalgorithmen zerlegt werden
- Große Systeme: auch strukturierte Programme werden unübersichtlich
  - Entwicklung durch einzelnen Autor nicht mehr möglich
  - Lebenszeit des Softwaresystems u.U. länger als ursprünglicher Autor am System arbeitet
- Lösungsansatz: Modulare Programmierung (ab 1970)
  - Arbeitsteilung: Entwickler sind für verschiedene Module zuständig
  - Entkopplung von Entwicklungszyklen: einzelne Module können separat fertiggestellt und revidiert werden
- Lösungsansatz: Objektorientierte Programmierung (ab 1980) – ursprünglich getrieben von interaktiven graphischen Systemen (Maus)

Modulare Programmierung

- Zerlegung des Gesamtsystems in Teilsysteme – üblich: Funktionsblöcke
- Spezifikation der einzelnen Module
  - Definition der Schnittstelle: Welche Funktionen/Prozeduren werden angeboten? Welche Datentypen sind Parameter und Ergebnis
  - Beschreibung der Semantik jedes Moduls
- Integration der Module
  - DeRemer, Kron: Programming In the Large versus Programming In the Small, 1975
- Module in verschiedenen Programmiersprachen entwickelt
  - “module interconnection language”: MIL 75
  - “glue languages”: Skriptsprachen, z.B. VisualBasic, Python...
  - heute: Sprachen unterstützen sowohl Moduldefinition als auch Modulintegration
- Komponentenbasierte Programmierung

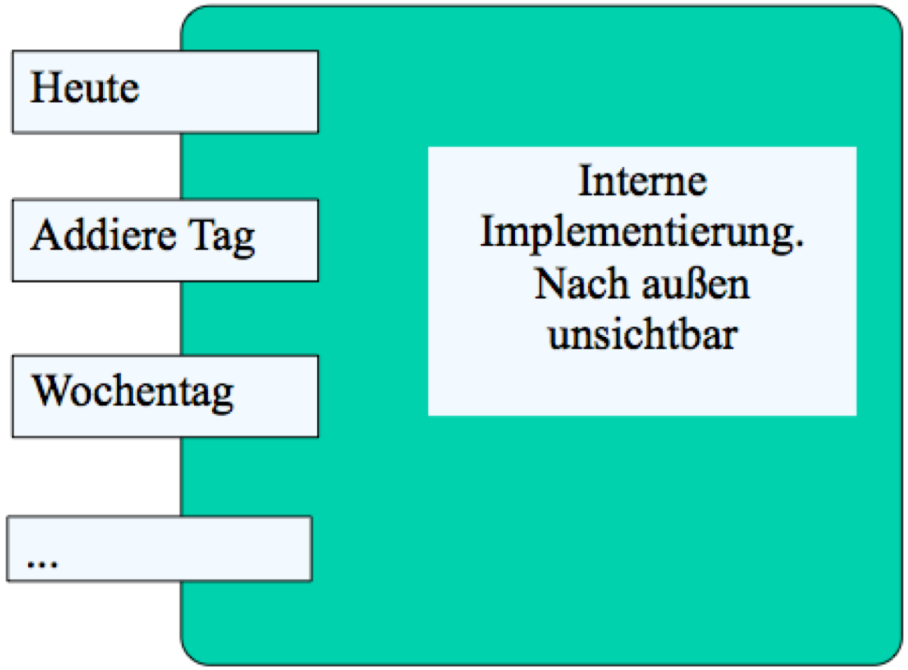
Modulare Programmierung (2)

- Beispiel: Ein Projektmanager



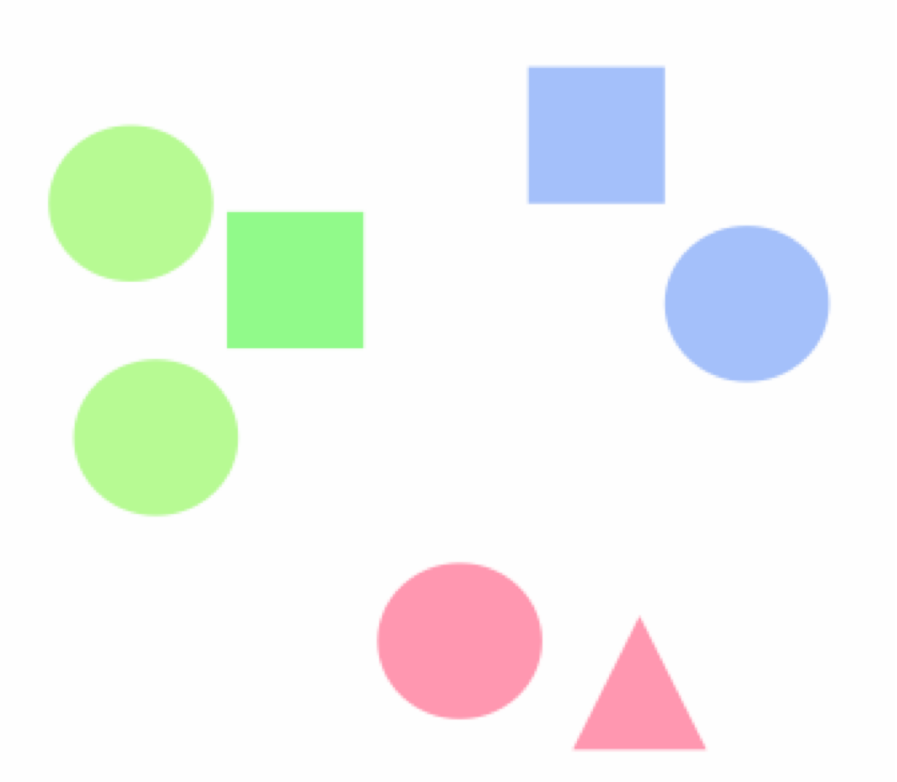
Schnittstellen

- Schnittstelle eines Kalendermoduls



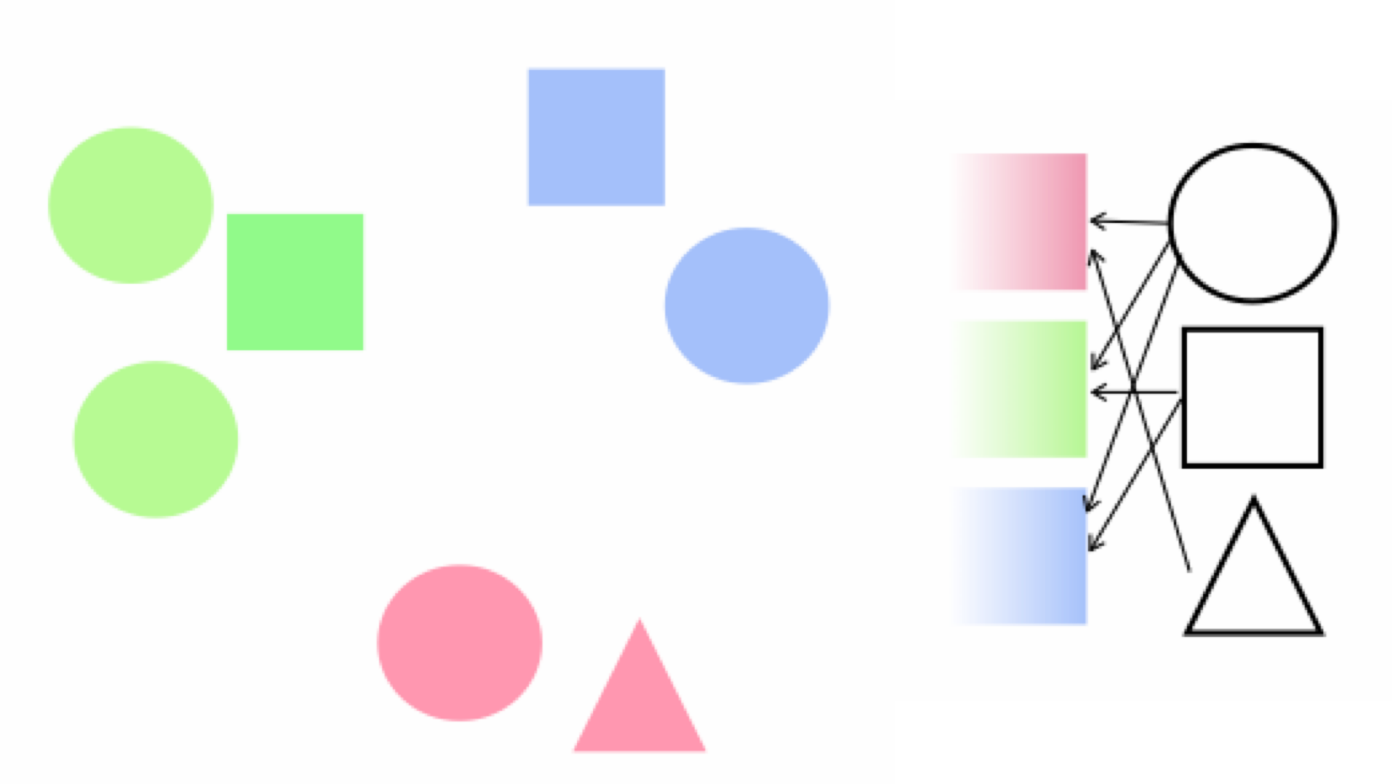
Von Modulen zu Objekten

- Farbe repräsentiert Daten;
- Form repräsentiert Verhalten



Ausdruck von Kombination

- Verschiedenes Verhalten benutzt gleiche Datenstrukturen



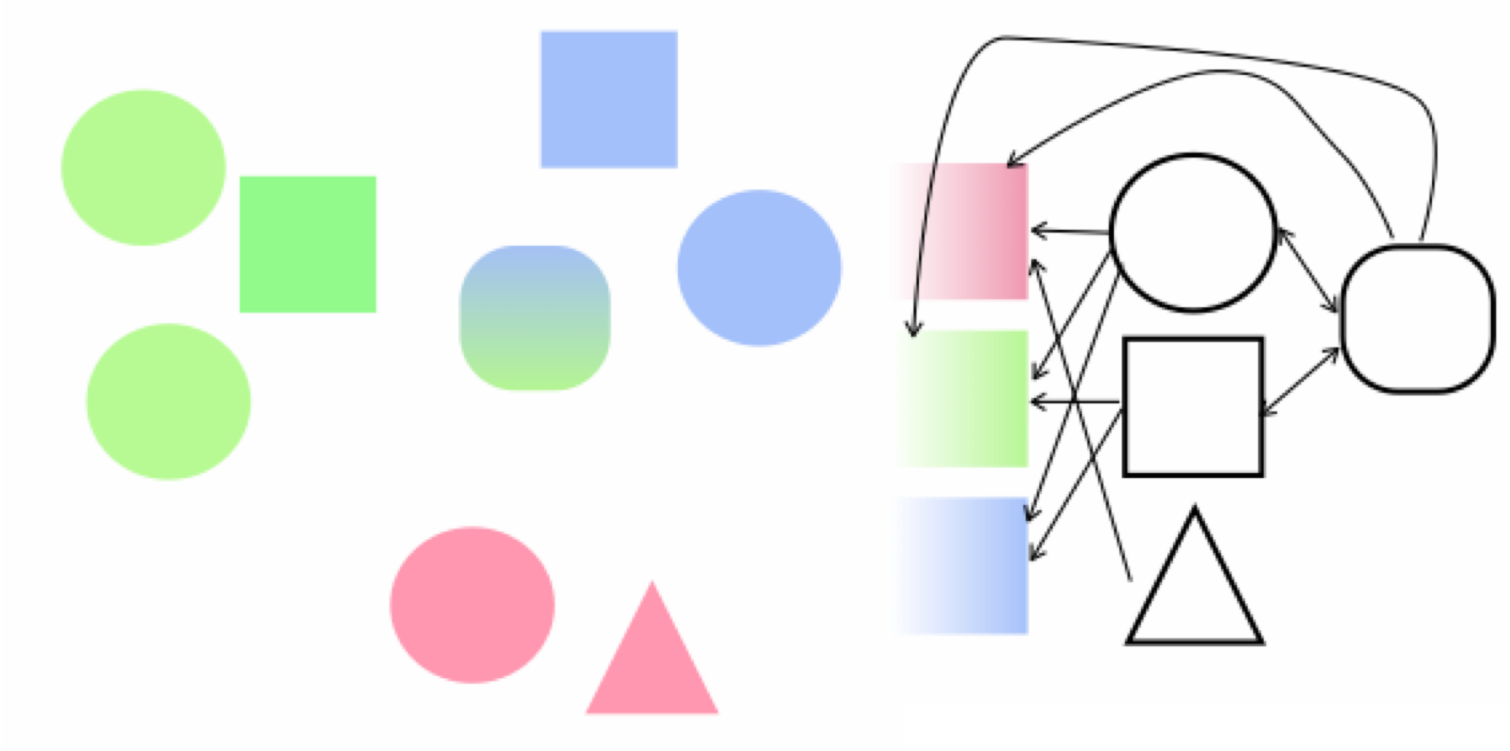
Beispiel in C

```
struct Point {
  const void * class;
  int x, y;
};

void draw (void * self) {
  switch(self) {
    case POINT_CLASS: ...
    case CIRCLE_CLASS: ...
  }
}
```

```
struct Circle {
  const void * class;
  int x, y, rad;
};

  void move (void * self) {
    switch(self) {
      case POINT_CLASS: ...
      case CIRCLE_CLASS: ...
    }
  }
```

Erweiterung von Daten und Verhalten

- Problem: Spaghetti Code

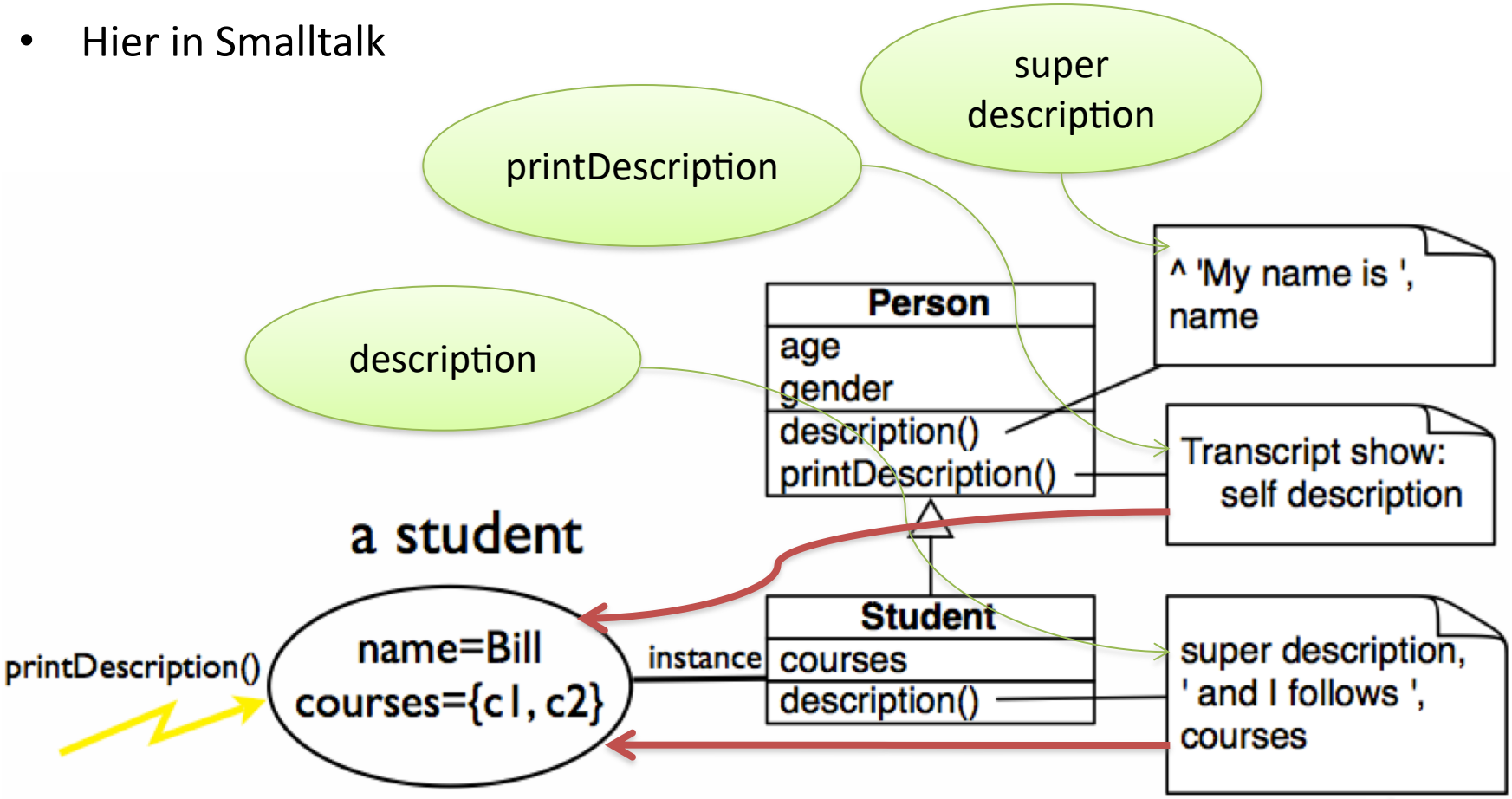


Objektorientierte Programmierung als Lösung

- Simula 67 adressierte die kombinatorische Explosion von Daten und Verhalten
- Eine Klasse ist eine ***object factory***
  - Enthält Vorschriften zur Erstellung von Objekten
- Klasse = Superklasse + Attribute + Methoden
  - Super class == Oberklasse
- Objekte reagieren auf Nachrichten (Methodenaufruf)

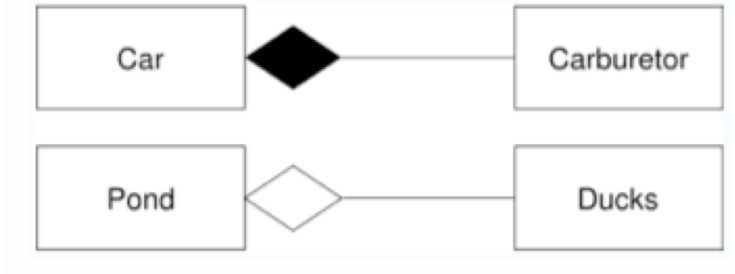
Objekte als programmiersprachliche Einheiten

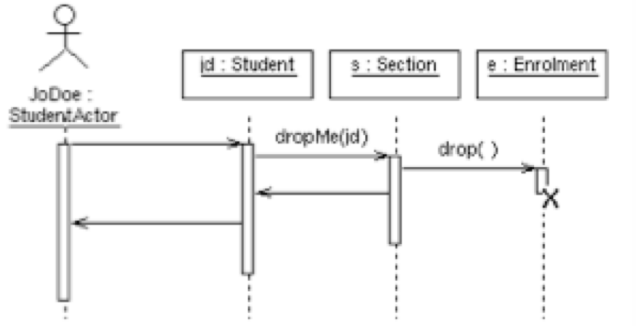
- Hier in Smalltalk

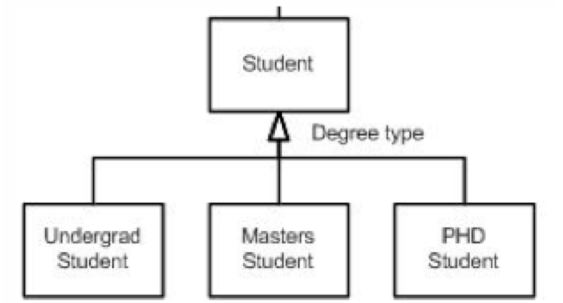


OO Konzepte im Überblick

- Datenkapselung
  - Abstraktion & Information Hiding
- Komposition
  - Verschachtelte Objekte
- Verteilung von Verantwortlichkeiten
  - Separation of concerns (e.g., HTML, CSS)
- Nachrichtenaustausch
  - Delegation von Verantwortung
- Vererbung
  - Konzeptionelle Hierarchien
  - Polymorphie
  - Wiederverwendung





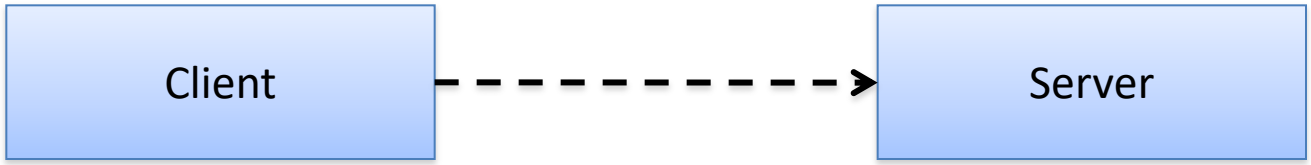


Allgemeiner: Objektorientierter Entwurf (OOD)

- Drei Ziele:
  - Verständlichkeit von Modellen und Diagrammen
  - Korrektheit von Modellen und Diagrammen
  - Wiederverwendbarkeit
- Keine strikten Regeln
- Pragmatisch
  - basierend auf heuristischen Erfahrungen
  - Muster für Vorgehen
- Nicht auf UML eingeschränkt
- Robert Martin: Engineering Notebook (C++-Report)
  - 8 Principles for OOD

Open-Closed-Principle

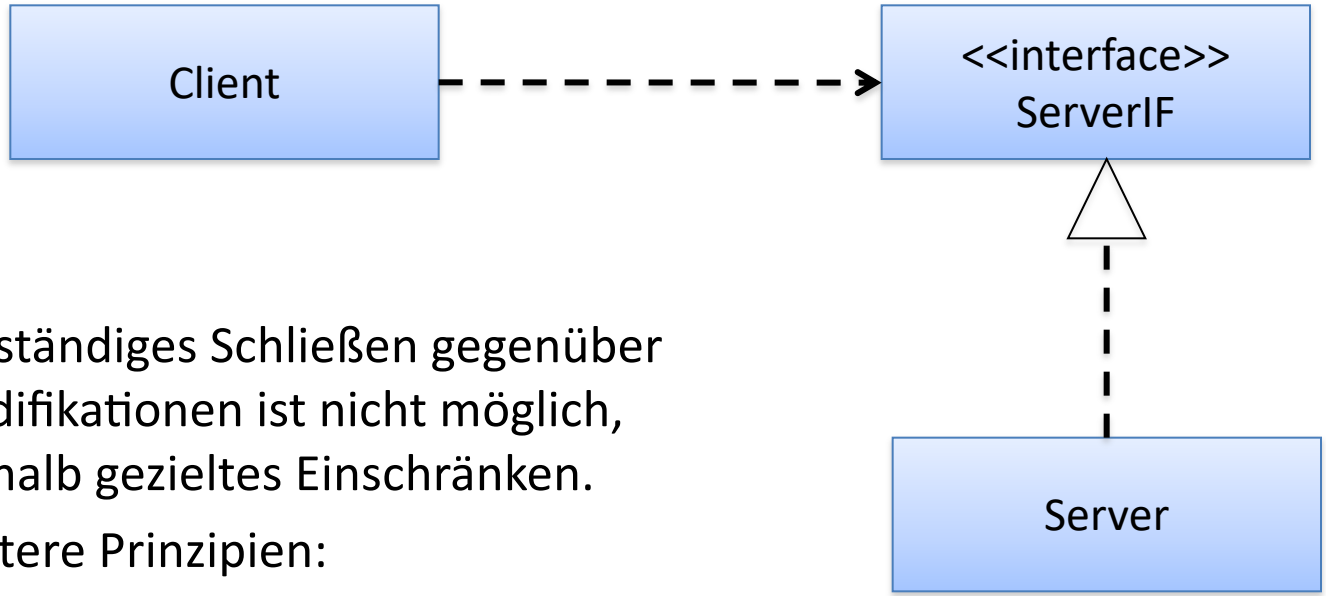
- Softwareeinheiten (Klassen, Module etc) sollen offen für Erweiterungen, aber geschlossen gegenüber Modifikationen sein!



- Client hängt vom Server ab; z.B.
  - Rufen von Operationen
  - Parameter von Operationen
- Einführung eines neuen oder Änderungen des Servers erfordert Änderungen am Client!

Lösung

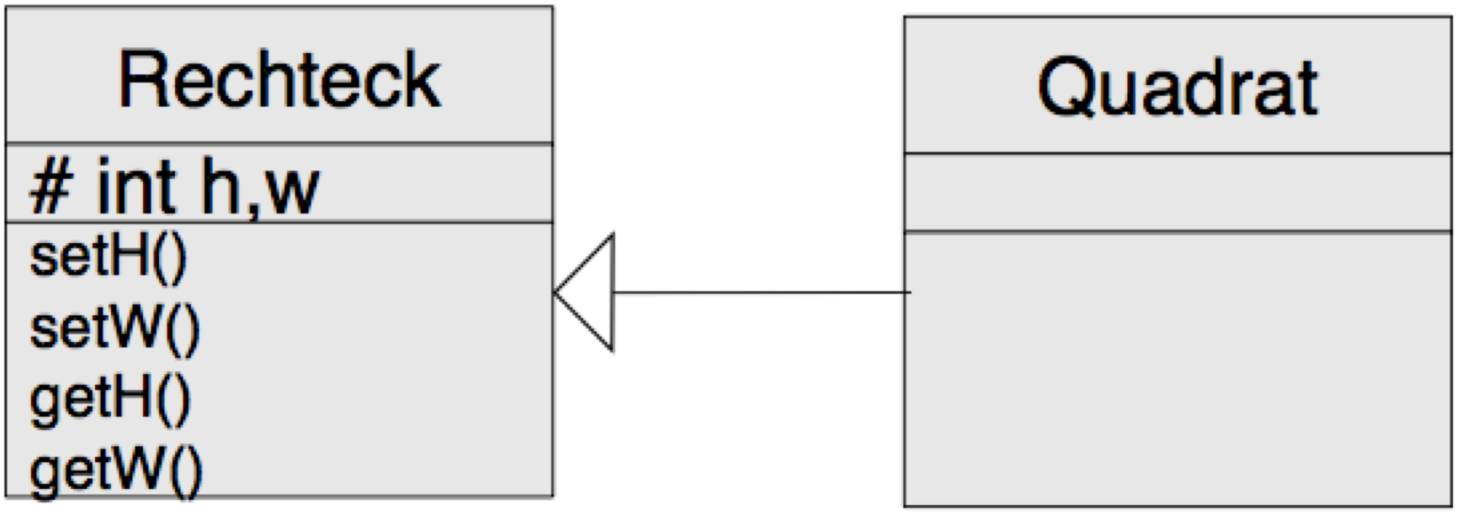
- Einführen von Interfaces oder abstrakten Klassen!
- Client ist unabhängig von Änderungen am Server.



- Vollständiges Schließen gegenüber Modifikationen ist nicht möglich, deshalb gezieltes Einschränken.
- Weitere Prinzipien:
  - Attribute grundsätzlich private, Modifikationen nur durch Operationen
  - keine Globalen Variablen -> strikte Abkapselung

Liskov Substitution Principle (LSP)

- Klassen und Operationen die Objekte einer Basisklasse verwenden, müssen Objekte davon abgeleiteter Klassen verwenden können, ohne dies zu merken!
- Barbara Liskov, 1988



LSP: Diskussion

- Quadrat kann anstelle von Rechteck genutzt werden
  - setH, setW sind redefiniert, um gleiche Kantenlängen zu garantieren
  - um korrekt zu arbeiten, müssen diese Funktionen aber in Rechteck virtuell sein!
- Trotzdem:
  - Nutzer von Rechteck merkt Unterschied:

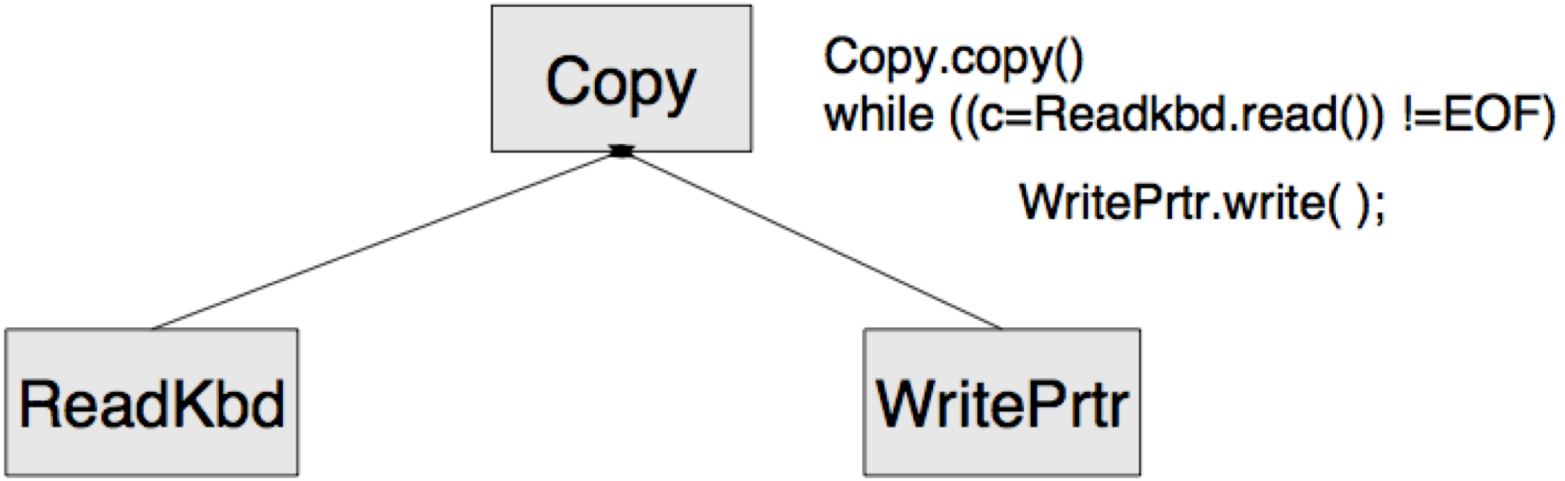
```
r.setW(3); r.setH(4);
x=r.getW() * r.getH(); // 12 bzw.16!!!
```

„ist-ein“ ?

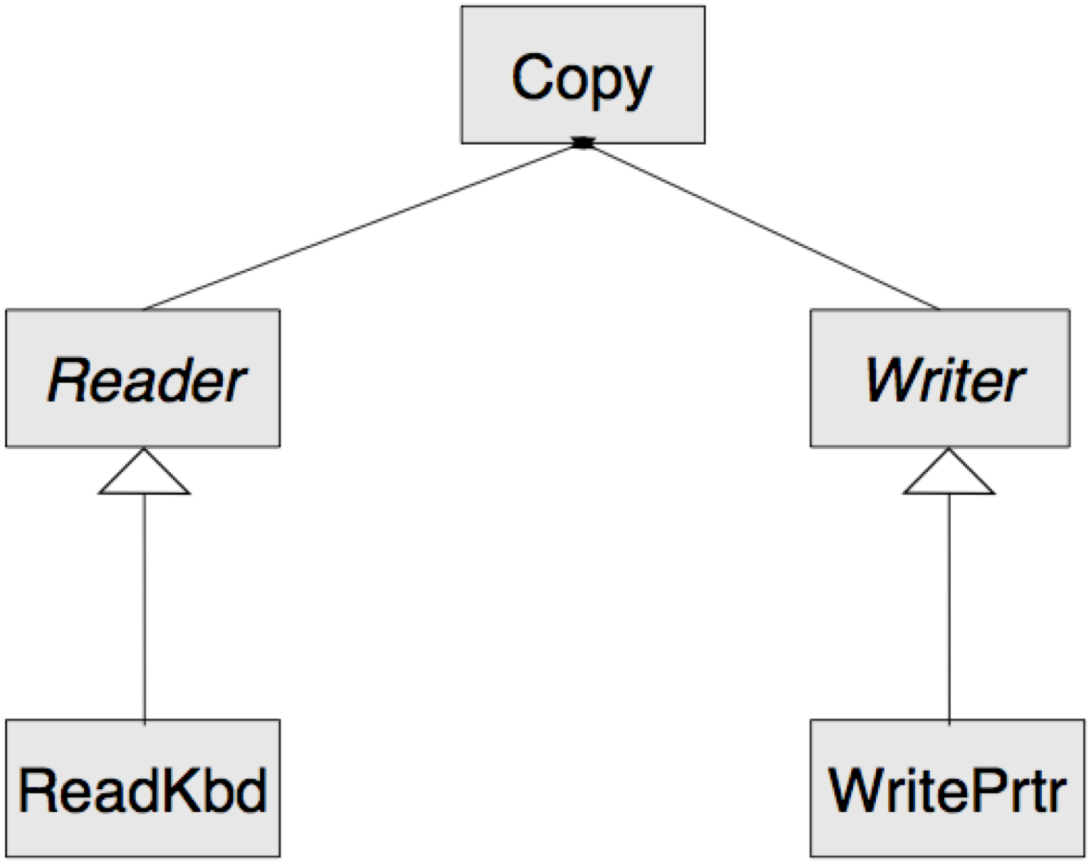
- Ist ein Quadrat kein Rechteck?
  - Als Struktur ja, aber Verhalten ist unterschiedlich!
- Verhalten muss immer mit berücksichtigt werden!
- Regel um LSP zu garantieren:
  - Eine Operation in einer Basisklasse sollte in einer abgeleiteten Klasse nur durch eine Operation redefiniert werden, die eine schwächere **Vorbedingung** und eine stärkere **Nachbedingung** hat!
  - **Nachbedingungen** bei Quadrat setH und setW sind schwächer als bei Rechteck!

Dependency Inversion Principle

- Module auf einem höheren Level sollen nicht von Modulen auf einem niedrigen Niveau abhängen!
- Abstraktionen sollen niemals von Details abhängen, Details sollen von Abstraktionen abhängen!



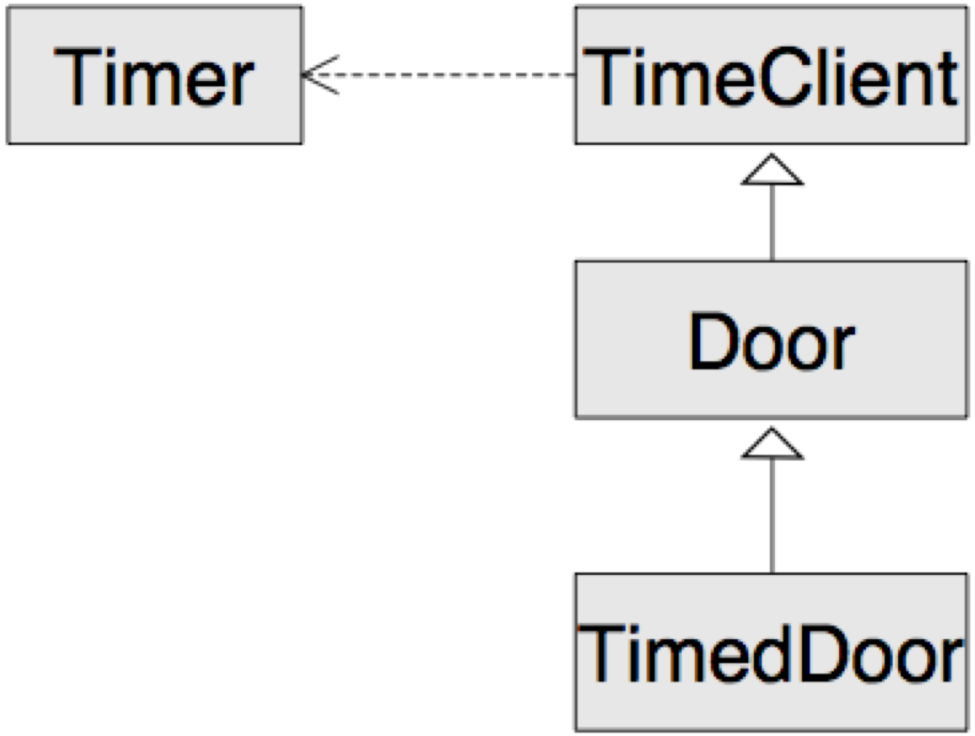
Abstraktion unabhängig von Implementierungsdetails



- Andere Lösung: Interfaces

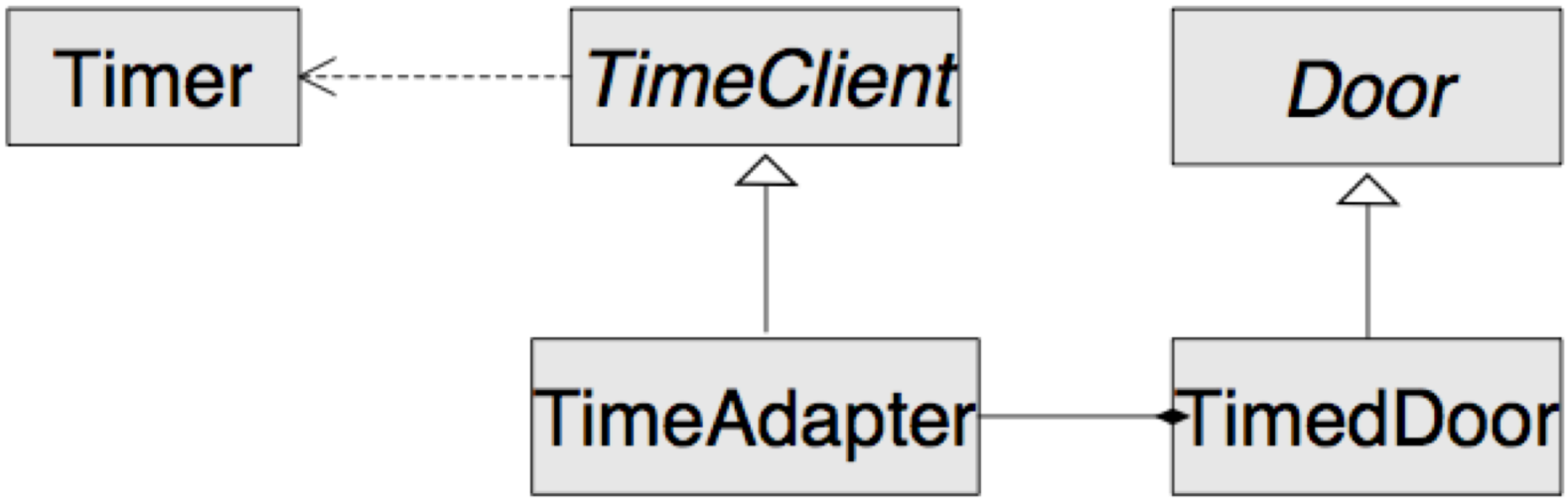
Interface Segregation Principle

- Klienten sollten nicht von Interfaces abhängen, die sie nicht benutzen!



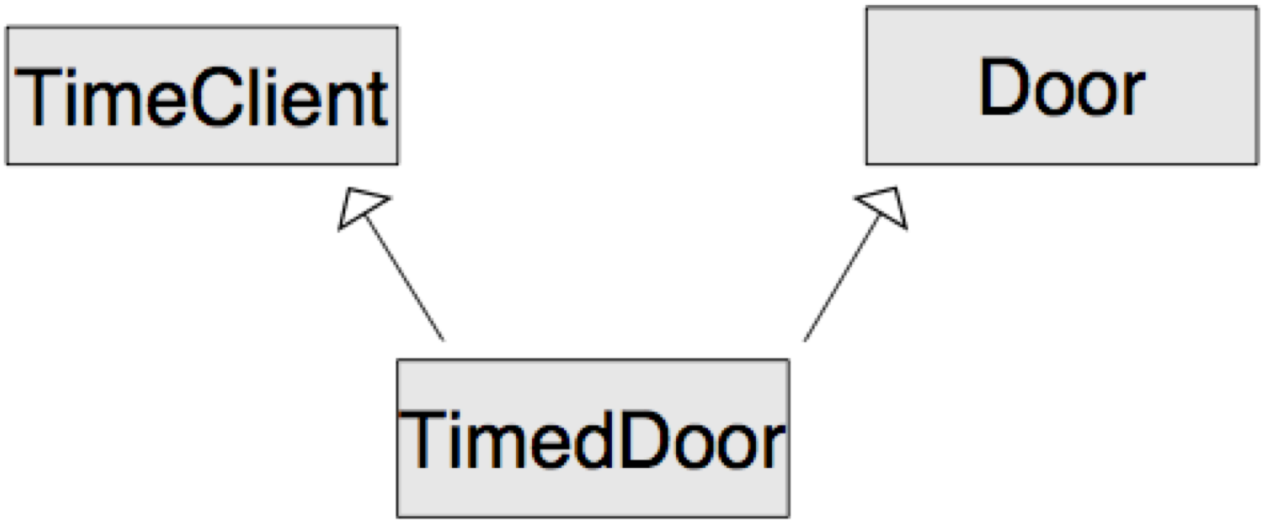
Bessere Lösung

- Separierung durch Delegation



Alternative

- Separierung durch Mehrfachvererbung



Granularitätsprinzip

- Reuse/Release Equivalence
- Die Granularität wiederverwendbarer Einheiten wird durch die Granularität von Releases bestimmt!
- Nur Komponenten, die durch eine Versionsverwaltungssystem freigegeben wurden sind effektiv wiederverwendbar.
- Zielgranularität: Package

Strukturierungsprinzipien

- Common Reuse
  - Die Klassen in einem Package werden zusammen wiederverwendet.
  - Abhängigkeit von einer Klasse bedeutet Abhängigkeit von allen Klassen
- Common Closure
  - Die Klassen innerhalb eines Packages sollten geschlossen sein gegenüber gleichen Arten von Änderungen.
  - Änderungen im Package betreffen alle Klassen im Package
- Acyclic Dependencies
  - Die Abhängigkeiten zwischen Packages sollen einen azyklischen gerichteten Graphen bilden
  - Aufbruch von Zyklen durch Dependency Inversion

Stabilität

- Die Abhängigkeiten zwischen Packages sollten in Richtung der Stabilität der packages verlaufen. Ein Package sollte nur von stabileren Packages abhängen!
- Stabilitätsmetrik:
  - CA: Anz. Von Klassen außerhalb des Packages, die von Klassen im Package abhängen
  - CE: Klassen im Package, die von Klassen außerhalb des Packages abhängen.
- Stabilität S::=CE/(CA+CE)
  - 0 - maximal stabil (unverantwortlich u. abhängig)
  - 1 - maximal instabil (verantwortlich u. unabhängig)
- Maximal stabile Packages sollten maximal abstrakt sein, instabile Packages sollten konkret sein.
  - Abstraktion sollte in Proportion zur Stabilität stehen.
  - Abstraktion::= Anz. Abstrakte Klassen / Gesamtzahl

Objektorientierte Programmierung

- Objekte = Daten + Methoden – genauer: Daten + Methoden + Identität
- Objekte sind nicht nur reine Datensätze, sondern enthalten auch (Verweise auf) Operationen für die Daten

- Methoden sind Unterprogramme, die für einen und innerhalb eines Datentyps definiert sind
  - Kapselung/Verkapselung (encapsulation): Zusammenfassung der Daten und Operationen zu einem abstrakteren Typ

- Klasse: Zusammenfassung von Datensatzstruktur und Methoden
  - Daten heißen Felder (fields) oder Attribute (attributes) der Klasse, oft auch members

OOP (contd.)

- Objekt: konkrete Belegung der Felder
  - oft auch Exemplar (instance):
  - Ein Exemplar der Klasse Button
  - seltener: Ein Objekt der Klasse Button

- fälschlich oft Instanz
  - In|s|tanz, die; -, -en <lat.> (zuständige Stelle bei Behörden oder Gerichten); In|s|tan|zen|weg (Dienstweg) [Quelle: Duden, 22. Auflage, 2000]

Klassen

- doppelte Verwendung des Begriffs “Klasse”:
  1. Teil einer Klassifikation
  2. Programmkonstrukt zur Kapselung und Wiederverwendung von Datenstrukturen und Algorithmen
- Klassifikationen: Unterteilung einer Menge in Teilmengen • z.B. mittels Äquivalenzrelation in Äquivalenzklassen
  - z.B. mittels Taxonomie in Taxa (sing. Taxon, Gruppe)
  - Biologie: Rang einer Gruppe: Art, Gattung, Familie
- synonym: Systematik
- Einordnung eines Objekts in eine Klasse: Klassierung
- jedes Objekt kann u.U. zu mehreren Klassen gehören, u.U. auch seine Klassenzugehörigkeit mit der Zeit ändern
- Klassen in der OOP: u.U. auch Klassifikation
  - aber: Programmstruktur steht im Vordergrund

Datenkapselung

- Ziel: Verstecken der Datenstrukturen (information hiding)
  - Austausch der Repräsentation von Daten ohne Änderung der Methodensignaturen wird möglich
  - Beispiel: Punkte auf der Ebene entweder in kartesischen oder Polarkoordinaten repräsentierbar
- Ziel: Durchsetzung von Regeln für Daten unabhängig von Anwendungsprogramm (Protokolle, protocols)
  - z.B. Wahrung von Invarianten
  - "Der Kontostand muss immer oberhalb des Dispositionskredits sein."
  - z.B. Durchsetzung der Buchführung (logging)
  - Ziel: Gleichzeitige Verwendung unterschiedlicher Realisierungen eines Protokolls (Polymorphie, polymorphism)
- z.B. GUI: verschiedene Klassen (`Button, Label, PictureBox`) implementieren alle eine Methode `.Show`)

Vererbung

- Erweiterung einer bestehenden Klasse um neue Eigenschaften (neue Daten, neue Methoden)
  - nicht durch Änderung der Klasse, sondern durch Definition einer neuen Klasse
  - neue Klasse: erweiterte oder abgeleitete oder Unterklasse (subclass)
  - bestehende Klasse: Basis- oder Oberklasse (superclass)
- Beispiel:
  - Basisklasse: Verzeichniseintrag (Attribute: name, besitzer, datum der letzten Änderung)
  - Ableitungen:
    - Datei (Attribut: Größe, Inhalt) – Methoden: read, write
    - Verzeichnis (Attribute: Liste von Verzeichniseinträgen) – Methoden: create_file, make_directory, read_entries

Interpretation von Vererbung

- Vererbung im juristischen Sinne:
  - Eigentum geht in Besitz des Erben über
- Vererbung im biologischen Sinne:
  - Erbgut wird als Kopie in neues Leben übergeben – Eltern besitzen weiterhin das Erbgut
- OOP-Begriff angelehnt an biologischen Begriff

Polymorphie

- von “ πολυμορφία”: vielgestaltig
- Ziel: Austausch von Klassen als Parameter eines Algorithmus, ohne Änderung der Klasse
- Beispiel: Bestimmung der 2D-Objekte, die sich an einem bestimmten Punkt befinden

```
p = Point(10,7)
for o in objekte: if o.enthaelt(p):
        print o.als_text()
```

Polymorphie (contd.)

- Geometrische Objekte: Bestimmt durch geschlossene Linie
  - Form der Linie abhängig von Objekt
  - allgemeines, objekt-unabhängiges Kalkül für geschlossene Linien schwer realisierbar
  - Betrachtung von “interessanten” Spezialfällen:
- Kreise, achsenparallele Rechtecke
- Verallgemeinerung: Ellipsen, Vielecke
- Verallgemeinerung: Linienzüge auf Basis von Strecken, Kreisbögen, Splines
  - Thema der Computergrafik

Liskov Substitution Principle (LSP)

- Barbara Liskov, Data Abstraction and Hierarchy, SIGPLAN Notices. 23(5), May 1988
- Funktionen, die Referenzen auf die Basisklasse erwarten, sollen Exemplare der Ableitung verarbeiten können, ohne es zu wissen.
- “If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T.”

„ist-ein“ oder „hat-ein“

- Relationen zwischen Klassen A und B:
  - A kann Basisklasse von B sein
  - A kann Element/Feld von B sein
    - genauer: Exemplare von A können Felder von Exemplaren von B sein
- “B ist ein A” (is-a): Exemplare von B können überall da auftreten, wo auch Exemplare von A erlaubt sind
  - Beispiel: Rechteck ist-ein Zwo_D_Objekt
  - Interpretation als Klassifikation: B ist Teilmenge von A
  - besser: Interpretation im Sinne von LSP
- “B hat ein A” (has-a): Attribute von B haben A als Typ
  - Beispiel: Kreis hat-einen Punkt (Rechteck sogar zwei)

Weitere Konzepte Objektorientierter Programmierung

- Mehrfachvererbung (Python, C++)
- Schnittstellen (interfaces, Java, C#)
- Properties (C#, Python)
- Integration von OOP in Ausnahmebehandlung
- Reflection/Introspection
- Parametrisierte Typen (generic types) (C++, C#, Java)

Zusammenfassung

- Programmieren im Großen
- Von Modulen zu Objekten
- OO Konzepte im Überblick
- Regeln für objektorientierten Entwurf
- Liskov Substitution Principle (LSP)
- Vererbung
- Polymorphie