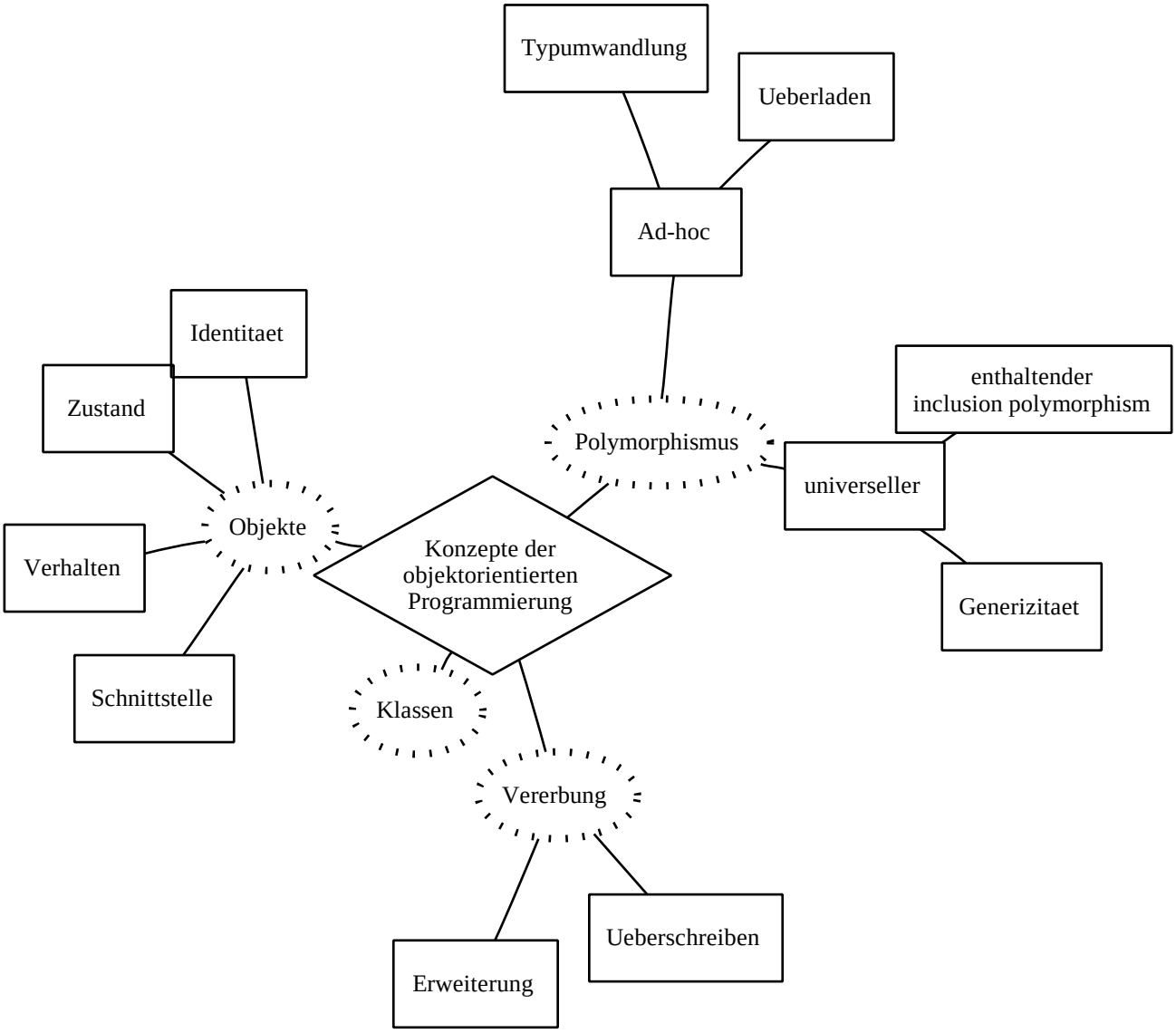
1 objektorientierte Programmierung

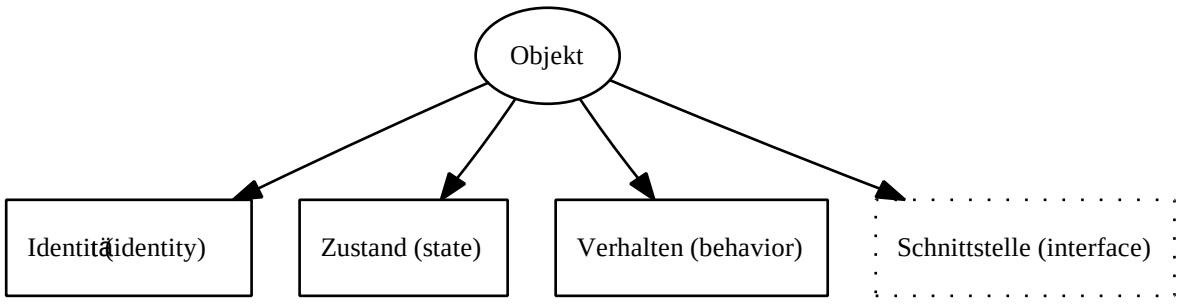
Ersetzbarkeitsprinzip und Zusicherungen *Design by Contract*
kovariante Spezialisierungen

1.1 Konzepte objektorientierter Programmierung



1.1.1 Objekt

- leichte Wartbarkeit
- *Objekt* tauschen untereinander *Nachrichten (Messages)* aus (keine public-Variable)
- Zusammenfügen von Daten und Routinen = *Kapselung*



Identität (`identity`) = `hashCode`, vereinfacht, die Adresse im Hauptspeicher; unveränderlich; Kopie eines Objektes hat unterschiedliche Identität

Zustand (`state`) = `~equals`, Inhalt aller Variablen; 2 Objekte sind gleich, wenn sie das gleiche Verhalten und den gleichen Zustand haben; Kopie eines Objektes hat den gleichen Zustand

Verhalten (`behavior`) = `.method()` was Objekt tut, wenn eine Nachricht (mit ggf. Argumenten) empfangen wird

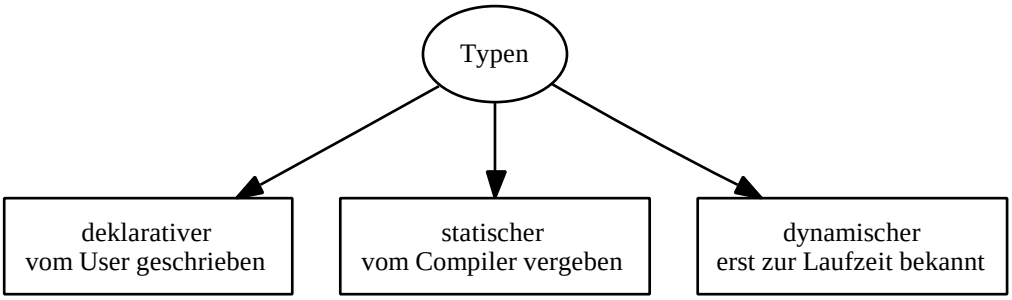
Schnittstelle (`interface` optional) = `interface` beschreibt was Objekt tut (ohne Implementierung); *Datenabstraktion*

1.1.2 Klassenkonzept

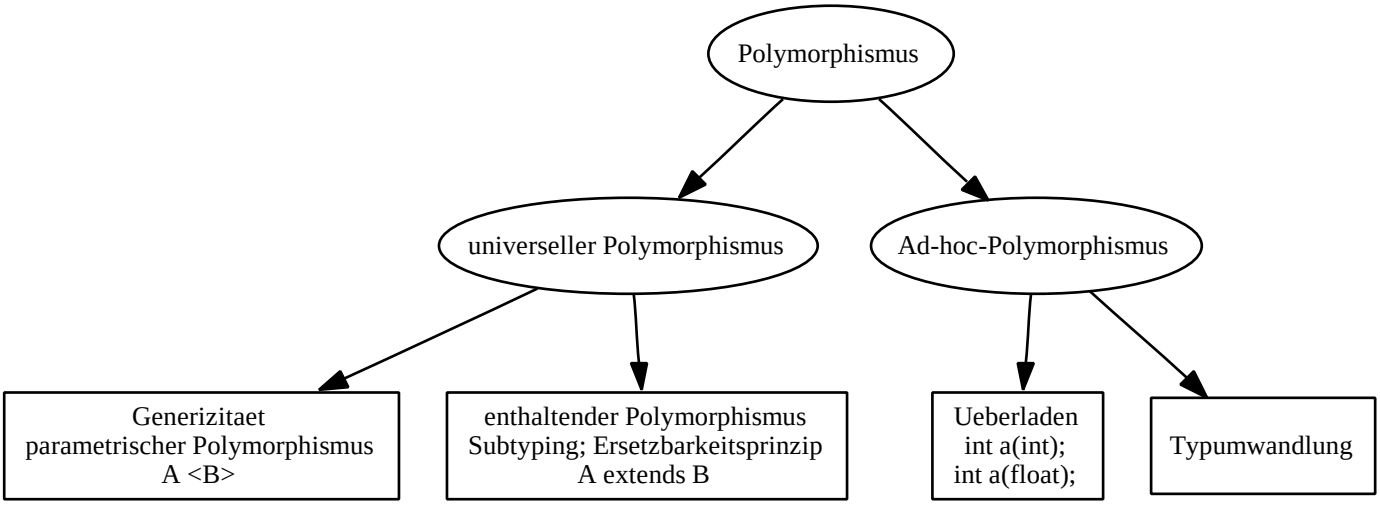
Objekte sind Instanzen einer Klasse welche bei der Initialisierung Konstruktoren (constructors) aufrufen.
Unterschiedliche Instanzen haben unterschiedliche Identitäten

1.1.3 Polymorphismus

wenn eine Variable oder eine Routine gleichzeitig mehrere Typen hat.
Konventionelle Sprache (wie z.B: Pascal) sind *monomorph,* objektorientierte Sprache sind *polymorph.*



Deklarierter Typ: *`Type` `var`;* explizite Typdeklaration
Statischer Typ: wird vom Compiler ermittelt (Optimierung) und kann spezifischer sein als deklarativer Typ; **NICHT** `static !!`
Dynamischer Typ: Können sich mit jeder Zuweisung ändern; werden für Typüberprüfung zur Laufzeit verwendet; oft spezifischer als deklarativer Typ



Generizität (`genericity`) = `List<String>` *parametrischer Polymorphismus*
Enthaltender Polymorphismus (`inclusion polymorphism`) = `Student extends Person` *Subtyping* Objekte vom Typ Student sind auch vom Typ Person; Person hat den *Untertyp* Student (⇒ Student ist ein *Obertyp* von Person); U extends T (gesprochen: U erweitert T); s.S: 80f

> `Definition` (`Ersetzbarkeitsprinzip`): U ist ein Untertyp von T (bzw. T ein Obertyp von U), wenn eine Instanz von U überall verwendbar ist, wo eine Instanz von T erwartet wird.

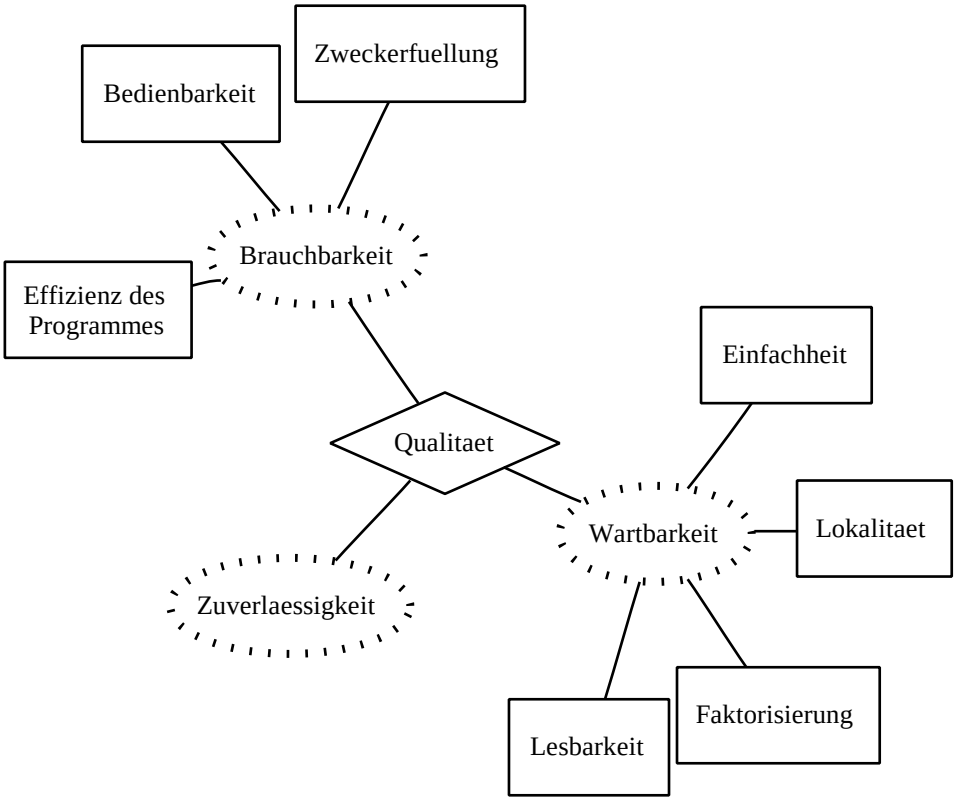
Die auszuführende Methode wird erst zur Laufzeit festgestellt = *dynamisches Binden (`dynamic binding`)*
Überladen (`overloading`) = *ad-hoc-polymorph* = Routine verhält sich bei unterschiedlichen Argumenten anders; gibt es auch in traditionellen Sprachen (z.B: "/"-Operator)
Typumwandlung (`type coercion`) = wenn z.B: von `char` in `int` impliziet bei einer Argumentenübergabe in C umgewandelt wird. ⇒gibt es auch in traditionellen Sprachen

1.1.4 Vererbung

Unterklasse (`subclass`) = abgeleite Klasse, *Oberklasse (`superclass`)* = Basisklasse
Erweiterung = Unterklasse erweitert Oberklasse
Überschreiben = Unterklasse überschreibt Methoden von der Oberklasse

1.2 Softwarequalität

1.2.1 Qualität von Programmen



Brauchbarkeit **Zweckerfüllung** - *Features* = Eigenschaften, die Anwender nicht braucht
Bedienbarkeit - wie hoch ist der *Einlernaufwand*;
Effizienz des Programmes - höhere Qualität, wenn sparsam mit Resourcen umgegangen wird

Zuverlässigkeit keine fehlerhaften Programmergebnisse und Systemabstürze; von Art der Anwendung abhängig (Software im Flugzeug oder am Home-Computer)

Wartbarkeit *Lebenszyklus* eines Programmes endet meistens, wenn es keine Wartung mehr gibt. Wartung = Anpassung auf sich ständig ändernde Bedingungen

Gute *Wartbarkeit* kann die Gesamtkosten erheblich reduzieren

Faktoren für ein *Programm*:
Einfachheit - Komplexität des Programmes so einfach wie möglich
Lesbarkeit - abhängig von *Programmierstil* und *Programmiersprache*
Lokalität - Effekte jeder Programmänderung soll auf kleine Programmteile beschränkt sein (wenig "globale" Variablen)
Faktorisierung = zusammengehörige Eigenschafte und Aspekte zu Einheiten zusammenfassen; =~ prozedural/funktional Programmieren

Gute *Faktorisierung* kann die Wartbarkeit eines Programmes wesentlich erhöhen

In OOP surch wesentlich erhöht.

1.2.2 Effizienz

ollen Objekte die reale Welt simulieren (nicht alles in Objekte zerlegen!) swoeit es nicht die Komplexität des Programmes dadurch wesentlich erhöht.

1.2.3 Effizienz der Programmerstellung und Wartung

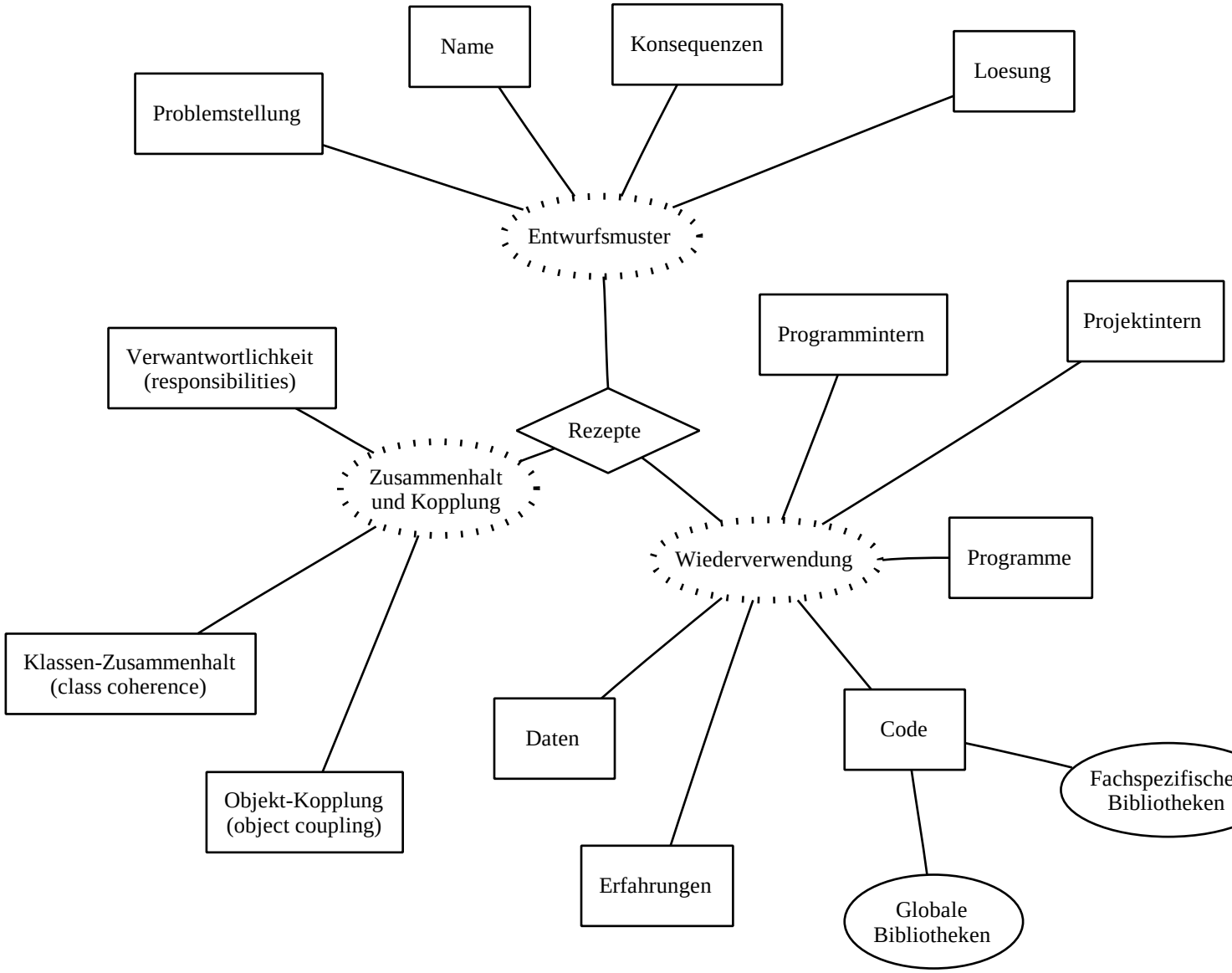
Softwareentwicklungsprozeß (`Wasserfallmodell`)

Analyse (`analysis`)**:** erstellen eines *Anforderungsdokuments*
⇓
Entwurf (`Design`)**:** *Entwurfsdokumentation* beschreibt, wie Anforderungen erfüllt werden sollen
⇓
Implementierung (`implementation`)**:** Entwurf wird in Programmstücke umgesetzt
⇓
Verifikation (`verification`) **und Validierung** (`validation`)**:** Überprüfung, ob Programm die Anforderungen, welche im *Anforderungsdokument* stehen erfüllt

Das Wasserfallmodell eigent sich für Projekte mit klaren Anforderungen.

Heute verwendet man die *zyklische Softwareentwicklungsprozesse*, welche das Wasserfallmodell zyklisch zuerst in kleinen Schritten, teilw. überlappend, durchläuft (= *schrittweise Verfeinerung*).

1.3 Rezepte für gute Programme



1.3.1 Zusammenhalt und Kopplung

Verantwortlichkeit: "was ich weiß" (Beschreibung des Zustands der Instanzen), "was ich mache" - Verhalten der Instanzen, "wen ich kenne" - sichtbare Objekte, Klassen, etc.

Klassen-Zusammenhalt = der *Grad der Beziehungen zwischen den Verantwortlichkeiten der Klasse.*

Objekt-Kopplung = *Abhängigkeit der Objekte voneinander* und ist stark, wenn

Faktorisierung = Zerlegung eines Programmes in Einheiten mit zusammengehörigen Eigenschaften

- public-Methoden und Variablen hoch
- viele Methodenaufrufe und Variablezugriffe zwischen unterschiedlichen Objekten
- Anzahl der Parameter dieser Methode hoch sind

Faustregeln

- Klassen-Zusammenhalt soll hoch sein (⇒*Refaktorisierung* kleiner)
- Objekt-Kopplung soll schwach sein
- verknünftiges Maß an rechtzeitiger Refaktorisierung führt häufig zu gut faktorisierten Programmen

Klassen-Zusammenhalt und Objekt-Kopplung sind umgekehrt proportionall beeinflußt voneinander.

1.3.2 Wiederverwendung

Programme können auch nur 1x verwendet werden!

Erfahrungen - Austausch von Konzepten und Ideen zwischen verschiedenen Projekten

Am Anfang eher weniger Zeit in Wiederverwendung investieren. Man kann dies durch Refaktorisierung noch nachholen, bei Bedarf.

1.3.3 Entwurfsmuster (design patterns)

= verbale (abstrakte) Beschreibung für die Lösung eines Problems im Softwareentwurf. Es besteht aus

Name (z.B: `Factory Method`), der **Problemstellung**, einer **Lösung** und den **Konsequenzen** (Vor- und Nachteile), die diese Lösung hat.

Zuviel Einsatz von Entwurfsmuster macht das Programm sehr komplex und schlecht wartbar. Sie sind daher nur im begrenzten Maße hilfreich.

1.4 Paradigma der Programmierung

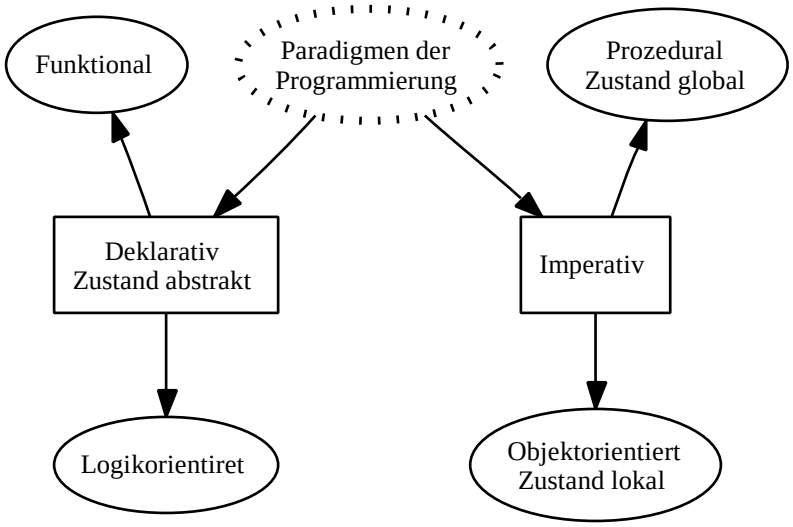
Paradigma = Stil in dem Programme geschrieben sind

1.4.1 Imperative Programmierung

beruht auf hardwarenahe Modelle; *Rechnerarchitektur* - z.B: von Neumann-Architektur; besteht aus *Anweisungen*, welche in einer festgelegten Reihenfolge ausgeführt werden und *destruktiven Zuweisungen* = eine Variable bekommt einen neuen Wert, unabhängig vom alten Wert; zeitlich aufeinanderfolgende Zustände

prozedurale Programmierung = traditioneller Programierstil; schreibt man in *strukturierte Programmierung*; Programmzustände sind im wesentlichen global; Algorithmus steht meist an einer Stelle

Objektorientierte Programmierung = Weiterentwicklung von der prozeduralen Programmierung; Objekt steht im Mittelpunkt; Algorithmus ist manchmal auf mehrere Programmteile aufgeteilt; Programmzustände sind lokal



1.4.2 Deklarative Programmierung

Beschreibt Beziehungen zwischen Ausdrücken **ohne** zeitlich aufeinanderfolgende Zustände; entstammen aus mathematischen Modellen; grundlegendes Sprachelement = *Symbol* (⇒*symbolische Programmierung*); Programmzustände sind abstrakt

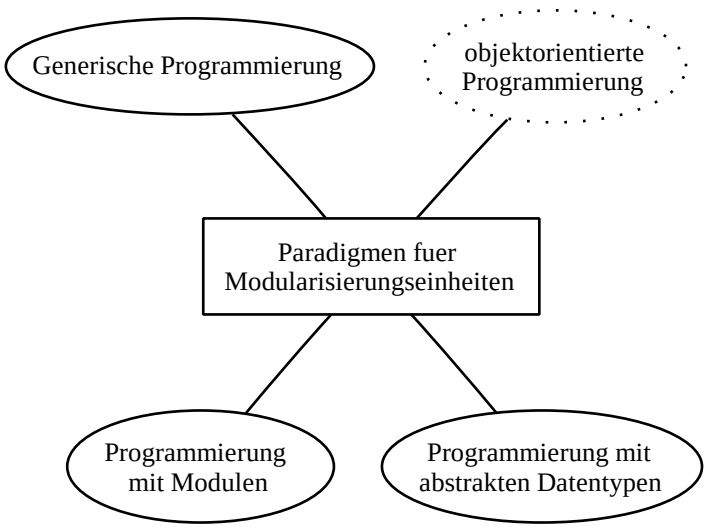
Funktionale Programmierung = basiert auf den *Lambda-Kalkül* welches *Funktion* formal beschreibt (`Sprachen: Lisp, ML, Mranda, Haskell`). Alle Ausdrücke werden als Funktion aufgefasst; *referentielle Transparenz* = ein Ausdruck bedeutet immer das selbe, egal wo er steht (es gibt auch keine Unterscheidung zwischen Objekt und der Kopie eines Objektes)

Logikorientierte Programmierung basiert auf Prädikatenlogik der ersten Stufe; alle wahren Aussagen werden mittels Fakten und Regeln beschrieben (`Sprache: Prolog` und Datenbankabfragesprachen).

objektorientierte Programmierung ist gut, wenn die Komplexität des Gesamtsystem jene der einzelnen Algorithmen übersteigt

1.4.3 Paradigmen für Modularisierungseinheiten

Programmierparadigmen beziehen sich auch auf die Faktorisierung. Folgende Paradigmen können mit imperativer und deklarativen Paradigmen kombiniert werden:



Programmierung mit Modulen legt großen Wert auf die Modularisierung (Sprachen: `Modula-2, Ada`); keine Unterscheidung zwischen Klassen und Objekt; Module entsprechen Objekten

Programmierung mit abstrakten Datentypen; ein ADT versteckt interne Darstellung; nur von ADT exportierte Operationen können angewendet werden; Klassen entsprechen ADT; die Programmierung mit ADT entspricht der objektorientierten Programmierung ohne Polymorphismus und Vererbung.

Generische Programmierung; Templates; wird vor allem in der objektorientierten und funktionalen Programmierung verwendet; (siehe auch *Generzität*); Bsp.: `List<Person>`, ...

(objektorientierte Programmierung); Es gibt auch die *deklarative objektorientierte Programmierung*, welche aber noch einige Probleme macht.

siehe S. 41 - WiederholungsFragen

2 Enthaltender Polymorphismus und Vererbung

2.1 Ersetzbarkeitsprinzip

= wichtigste Grundlagen des enthaltenden Polymorphismus

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

`class` Unterklasse `extends` Oberklasse `{ // B.S: 80`

2.1.1 Untertypen und Schnittstellen

U ist Untertyp von T wenn folgendes gilt

- $U = T$
- für jede Konstante T_{const} in T (vom Typ A) existiert eine Konstante U_{const} in U (vom Typ B), wobei B Untertyp von A; Typ(T_{const}) = Untertyp von Typ(U_{const})
- für jede Variable T_{Var} in T existiert eine Variable U_{Var} in U gleichen Typs; Typ(T_{Var}) = Typ(U_{Var})
- für jede Methode t in T existiert eine Methode u in U, wobei
  - Parameteranzahl gleich ist
  - Eingangsparametertypen in T Untertypen der in U; Typ($u_{Eingang}$) = Obertyp von Typ($t_{Eingang}$)
  - Ausgabeparametertypen in U Untertypen der in T; Typ($u_{Ausgang}$) = Untertyp von Typ($t_{Ausgang}$)
  - Ergebnisparameter: Typ($u_{Ergebnis}$) = Untertyp(Typ($t_{Ergebnis}$))
  - Durchgangsparameter = Eingangs- und Ausgangsparameter = Typ($u_{Durchgang}$) = Typ($t_{Durchgang}$)

Beim Definieren von **neuen Untertypen** kann man bestehende Typen verändern (variieren):

- **Kovarianz**: Untertyp-Beziehung; Parametertyp variiert MIT dem Klassentyp (ko = mit); Konstantentypen, Ergebnistypen, Ausgangsparametertypen
- **Kontravarianz:** Obertyp-Beziehung; Parametertyp variiert GEGEN den Klassentyp (kontra = gegen); Eingangsparameter
- **Invarianz:** Typen bleiben gleich; Durchgangsparameter

Beispiel in Java-ähnlicher Sprache (geht nicht in Java!):

```
class A {
  public A meth(B par) { ... }
}
class B extends A {
  public B meth(A par) { ... }
}
public C extends A {
  public A meth(C par) { ... }
}
```

In Klasse `A` und `B` ist der Ergebnistyp *kovariant* verändert und der Parametertyp *kontravariant*. Klasse `C` erfüllt NICHT die Kriterien der Untertypenbeziehung., sondern hat eine *binäre Methode* = Methode, bei welcher das Methodenargument vom gleichen Typ ist, wie die Klasse, wo es definiert ist (binär weil es 2x vorkommt, einmal als `this` und einmal als `par` in Klasse `C`).

In Java sind alle Typen invariant, außer dem Ergebnistypen (ab 1.5), welcher kovariant ist.

Faustregel: Kovariante Eingangsparametertypen und binäre Methoden widersprechen dem Ersetzbarkeitsprinzip.

2.1.2 Untertypen und Codewiederverwendung

Man soll auf Ersetzbarkeit achten, um

- Codewiedervrewendung zwischen Versionen zu erreichen
- interne Codewiederverwendung im Programm zu erzielen
- Programmänderungen lokal zu halten

Schnittstellen sollen stabil bleiben. Gute Faktorisierung hilft dabei.

Man soll nur Untertypen von stabilen Obertypen bilden.

Man soll Parametertypen vorausschaund und möglichst allgemein wählen.

2.1.3 Dynamisches Binden

Beispiel:

```
class A {
  public String ma() { return "Aa"; }
  public String mb() { return mx(); }
  public String mx() { return "Ax"; }
}
class B extends A {
  public String ma() { return "Ba"; }
  public String mx() { return "Bx"; }
}
```

```
class Test {
  public static void main(String args[]) {
    test(new A());
    test(new B());
  }
  public static void test(A x) {
    System.out.println(x.ma());
    System.out.println(x.mb());
  }
}
```

Ausgabe: `Aa`, `Ax`, `Ba`, `Bx!`

`mx` wird in `B` aufgerufen, da `B` der spezifischste Typ der Umgebung ist!!!

Dynamisches Binden kann man in traditionellen Sprachen mit `switch`-Anweisungen umschreiben. In der Objektorientierten Programmierung ist dynamisches Binden `switch`-Anweisungen und geschachtelten `if`-Anweisungen stets vorzuziehen.

2.2 Ersetzbarkeit und Objektverhalten

Weitere Bedingungen, welche nicht vom Compiler überprüft werden können, sind hinsichtlich der Ersetzbarkeit notwendig.

2.2.1 Client-Server-Beziehung

Objekte sind sowohl Client (weil sie Dienste in Anspruch nehmen) als auch Server (weil sie Dienste anbieten). *Objektverhalten* beschreibt was Objekt beim Aufruf einer Methode macht. Gesucht ist ein Mittelding (welches Vertrag (*Design by* Contract) zwischen Client- und Server heißt) zwischen Schnittstelle (welche das Objekt nur rudimentär beschreibt) und Implementierung (welche das Objekt zu vollständig beschreibt).

Die Bedingung (= *Zusicherungen* bzw. *assertions*) des Vertrages sind wie folgt:

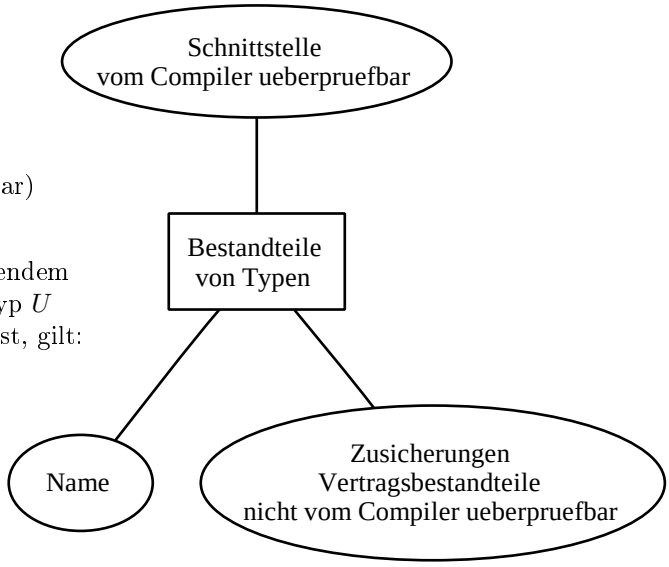
- **Vorbedingung (preconditions):** Diese Bedingungen muß der Client vor Ausführung der Methode erfüllen.
  Bsp.: Zahlen im Wertebereich von 0 bis 99 oder bereits vorherige Methodenaufrufe am Server
- **Nachbedingungen (postconditions):** Diese Bedingung muß der Server nach Ausführung der Methode erfüllen.
  Bsp.: Einfügen eines Elements in einer Menge - Element muß auf jeden Fall in dieser Menge sein

- **Invarianten (invariants):** Diese Bedingung muß der Server vor und nach Ausführung JEDER Methode erfüllen. Public-Variablen kann Server nicht überprüfen.
  Bsp.: Gutehaben auf Sparbuch muß immer positiv sein, egal welche Operation durchgeführt wird.

Teilweise sind Vor- und Nachbedingungen bereits durch Parameter- und Ergebnistyp festgelegt. `Eifel` kann im Gegensatz zu `Java` Vor- und Nachbedingungen.

Faustregeln bzgl. Zusicherungen:

- Sollen stabil bleiben, besonders an der Wurzel der Typhierachie.
- Keine unnötigen Details und Zusicherungen festlegen
  Verbessert Wartbarkeit
- Sollen sprechenden Namen haben (oder zumindest ein Kommentar)



2.2.2 Untertypen und Verhalten

Für Zusicherungen die zu Typen gehören und wo man den enthaltendem Polymorphismus anwendet gilt auch das Ersetzbarkeitsprinzip. Für Typ U mit der Methode u, welcher Untertyp vom Typ T mit der Methode t ist, gilt:

- **Vorbedingungen** in u können schwächer sein als in t, durch eine Oder-Verknüpfung.
  zB $t_{Vorbedingung} = x > 0 \Rightarrow u_{Vorbedingung} = x > 0 \ \vee \ x = 0$
- **Nachbedingungen** in t können stärker sein als in u, durch eine Und-Verknüpfung.
  zB $t_{Nachbedingung} = x > 0 \Rightarrow u_{Vorbedingung} = x > 0 \ \wedge \ x > 2$
- **Invariant** wie Nachbedingung

Instanzvariable sollen nicht durch andere Objekte verändert werden.
Bsp.: Zusicherung in Java: `IteratorWithoutRemove, ..., ReadOnlyList`

2.2.3 Abstrakte Klassen

Definition, wie in Java. Eine abstrakte Methode (= nicht implementierte Methode) ist eine Zusicherung, daß die Methode in einer nicht abstrakten (= *konkreten)* Unterklasse implementiert ist. Abstrakte Klassen, die keine Implementierung enthalten, sind eher stabil!

2.3 Vererbung versus Ersetzbarkeit

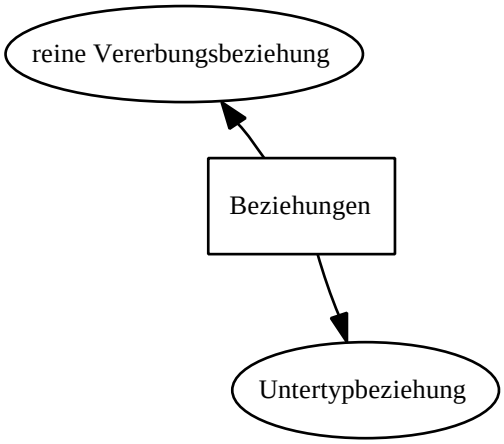
Wenn man Einschränkung durch enthaltenden Polymorphismus ignoriert, kann man Oberklasse frei verändern

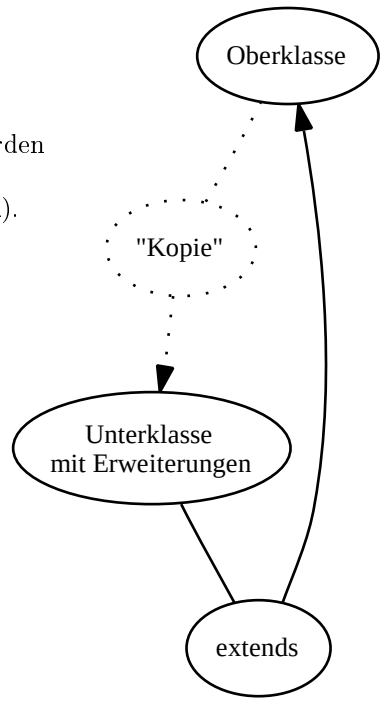
2.3.1 Reale Welt versus Vererbung versus Ersetzbarkeit

Untertypbeziehung = Ersetzbarkeitsprinzip

Vererbungsbeziehung = Code kann (muß aber nicht) übernommen werden kein Ersetzbarkeitsprinzip

Reale-Welt-Beziehung = *is-a* Beziehung (z.B: `Student` ist eine `Person`). Ggf. werden Beziehungen wieder aufgelöst bei der Refaktorisierung, da sie dem Ersetzbarkeitsprinzip widersprechen.





reine Vererbungsbeziehung = soviel Teile wie möglich von der Oberklasse direkt in der Unterklasse wiederverwenden (höherer Grad an Codewiederverwendung). Aber: Nichtbeachtung des Ersetzbarkeitsprinzip!

Untertypbeziehung = Ersetzbarkeitsprinzip (⇒ leichtere Wartung ⇒ besser Wahl)
In Java nicht unterscheidbar.

2.3.2 Vererbung und Codewiederverwendung

Manche Sprachen (z.B: C++) bieten mehr Möglichkeiten bei der Vererbung, stehen aber im Widerspruch zum Ersetzbarkeitsprinzip.

3 Generizität und Ad-hoc-Polymorphismus

Kein dynamisches Binden erforderlich, da es im wesentlichen ein statischer Mechanismus ist.

3.1 Generizität

Wird auch in nicht-objektorientierten Programmiersprachen angeboten.

Beispiel (mit *generischer Methode*):

```
abstract class MyCollection <A> {
   void add(A elem);
   Iterator<A> iterator();
   /* generische Methode:
   */
   public static <A> A max(Collection<A> xs, Comperator<A> c) {
      ...
      return ...;
   }
}
```

Im Beispiel muß `A` immer den selben Typ haben (also z.B: `Collection<String>` und `Comperator<String>`).

Typinferenz = Vorgang zur Berechnung (Überprüfung/Ersetzung) der Typen.

3.1.1 Gebundene Generizität in Java

```
class ObjectCloner<C extends Cloneable> {  ...  }
```

Gebundene Typparameter haben eine Klasse oder ein Interface als *Schranke* (in diesem Fall `Cloneable`; Typen von Typ `C` können alle Methoden von `Cloneable` verwenden).

Die *F-gebundene Generizität* (wie auch in dem Beispiel), unterstützt KEINE Untertypenbeziehung:

```
class Integer implements Comparable<Integer> {  ...  }
```

Bei Arrays gibt es (aus historischen Gründen) eine Untertypen beziehung. D.h. (z.B: `String[]` ist ein Untertyp von `Object[]`, da `String` ein Untertyp von `Object` ist, obwohl dies beim Schreiben falsch sein kann). Beispiel:

```
Strings xs[] = new String[10];
Object ys[] = xs; // Kein Compilerfehler!
ys[0] = y; // wirft ArrayStoreException!
```

Bei gebundenen Generizitäten geht das nicht, aber es gibt auch einen Nachteil (beim z.B. Lesen). Beispiel:

```
class Triangle extends Polygon { ... }
... void drawAll(List<Polygon> p) { ... }
{ Trinangle t = ...;
  drawAll(t); // geht nicht!
}
```

Abhilfe schaffen hier *gebundene Wildcards* (Beispiel):

`...  void drawAll(List<?  extends Polygon> p) { ...  }`

Der Compiler erlaubt nur Untertypbeziehung wo Kovarianz gefordert ist (= nur lesenden Zugriff), da schreibender Zugriff fehlerhaft sein könnte (wie beim z.B. Array).

`void add(List <?  extends Square> from, List<?  super Square> to) { ...  }`

In diesem Beispiel verlangt der Compiler, das beim `to`-Argument nur Kontravariante Untertypenbeziehungen sind (= Schreibzugriffe). `extends` steht für "Untertyp von" und `super` für "Obertypen von".

In Java kann man Typparameter nicht zur Erzeugung von neuen Objekten verwenden (`new A()` ist illegal, wenn `A` ein Typparameter ist). Das Erzeugen von Arrays (wie z.B: `new A[n]`) geht hingegen schon (und ist mitunter nicht statisch Typsicher!).

3.1.2 Richtlinien für die Verwendung von Generizität

Generell ist der Einsatz immer sinnvoll wenn die Wartbarkeit verbessert wird

Einsatz von Generizität bei

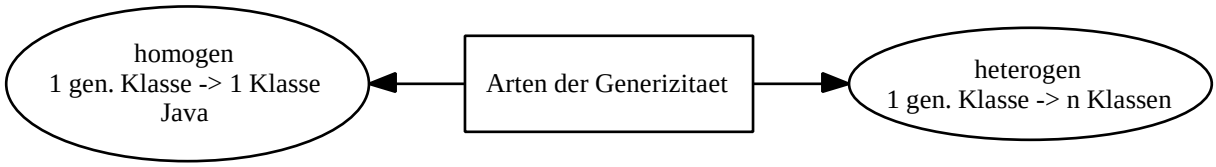
Gleich strukturierte Klassen oder Routinen, wie

- Containerklassen (Listen, Stacks, Hashtabellen, Mengen) und Routinen die auf solche zugreifen (Suchfunktionen und Sortierfunktion)
- Bibliotheken (Routinen und Klassen)

Abfangen erwarteter Änderungen

- gleich von Anfang an Typparameter verwenden, wenn man Änderungen dbzgl. erwartet
- Generizität und Untertyprelationen ergänzen sich - Entscheidung, was man verwenden soll
- Generizität und Untertypen sind auch austauschbar (B.S: 106)
- keine Effizienzüberlegungen, ob man Generezität oder Untertypenbeziehungen einsetzt

Arten der Generizität



homogene Generizität - Jede Klasse, wird in eine Klasse im Bytecode übersetzt (`Java`). Alle Klassen haben somit einen (mitunter automatisch erzeugten) Obertyp.

heterogene Generizität - Aus alle möglichen Typen, welche in EINER generischen Klasse vorkommen können werden MEHRERE reale Klassen erzeugt (`C++`; Prinzip wie "Copy- and Paste") ⇒ größerer Code, aber schnellere Laufzeit

Bei manche Sprachen ist die Art der Generizität nicht definiert. In `Ada` kann man auch Routinen (wie die Typen) generisch an eine Funktion übergeben!

3.2 Typabfragen und Typumwandlung

Dynamische Typinformation

prozedurale Sprachen	funktionale Sprachen	objektorientierte Sprachen
Nein	Nein	Ja (z.B: `getClass` in `Java`)

Faustregel: Typabfragen und Typumwandlungen von Objekten sollen nach Möglichkeit vermieden werden.

Die *homogene* Übersetzung der generischen Klassen ist sehr einfach: Alle `< >` Klammern werden weggelassen und die Typen durch die untere Schranke (falls angegeben) oder `Object` ersetzt.

Falls ein generischer Typ zurückgeliefert wird muß der Compiler diesen evt. in den entsprechenden Typ umgewandelt werden. Beispiel:

```
List<Integer> xs = new List<Integer>();          ⇒    List xs = new List();
xs.add(new Integer(0));                               xs.add(new Integer(0));
String y = xs.iterator().next();                      String y = (String)xs.iterator().next();
```

Man soll nur sichere Formen der Typumwandlung einsetzen. Typumwandlungen sind sicher wenn:

- ein Obertyp des deklarierten Typs umgewandelt wird (= "down-cast")
- bzw. duch dynamische Typabfrage sicherstellen
- "up-cast" Typumwandlung

Aufpassen, bei z.B: `List<String>` und `List<Integer>`, da die beiden Typen eigentlich NICHT gleich sind (auch wenn der Typ nach der generischen Übersetzung `List` heißt)!

3.2.1 Kovariante Probleme

Kovariante Eingangsparametertypen wiedersprechen dem Ersetzbarkeitsprinzip, sind aber u.u. erwünscht ⇒ *kovariantes Problem.* Beispiel:

```
abstract class Futter { ... }
class Gras extends Futter { ... }
class Fleisch extends Futter { ... }
abstract class Tier {
  public abstract void friss(Futter x);
  ...
}
class Rind extends Tier {
  public void friss(Gras x) { ... }
  public void friss(Futter x) {
    if ( x instanceof Gras )
      friss((Gras)x);
    else
      erhoeheWahrscheinlichkeitFuerBSE();
  }
}
```

```
class Tiger extends Tier {
  public void friss(Fleisch x) { ... }
  public void friss(Futter x) {
    if ( x instanceof Fleisch )
      friss((Fleisch)x);
    else
      fletscheZaehne();
  }
}
```

Problem: Wenn `Rind` `Fleisch` bekommt, kommt es zu unerwünschten Verhalten (in diesem Fall `BSE`).
Einzige vernüftige Lösung wäre, `friss` aus `Tier` zu entfernen ⇒ man kann Tiere nur mehr füttern wenn man die Tierart genau kennt.

Faustregel: Kovariante Probleme soll man vermeiden!

Spezialfall - *binäre Methoden* - Beispiel:

```
abstract class Point {
  public final boolean equal(Point that) {
    if ( this.getClass() == that.getClsas())
      return uncheckedEqual(that);
  }
  protected abstract boolean uncheckedEquals(Point p);
}
class Point2D extends Point {
  private x, y;
  protected boolean uncheckedEqual(Point p) {
    Point2D that = (Point2D)p;
    return x==that.x && y==that.y;
  }
}
```

```
class Point3D extends Point {
  private int x, y, z;
  protected boolean uncheckedEqual(Point p) {
    Point3D that = (Point3D)p;
    return x==that.x && y==that.y
        && z==that.z;
  }
}
```

Trick: `uncheckedEqual` wird immer in der richtigen Klasse aufgerufen und `equal` liefert `false` falls beide Klassen ungleich sind!

3.3 Überladung versus Multimethoden

z.B: in Java ist Typ von `y` bei `x.equal(y)` beim dynamischen Binden unrelevant. Wenn man den Typ von `y` auch noch in `equal` einbeziehen möchte spricht mann von *Multimethoden* (dies geht leider nicht in Java). Beispiel:

```
Rind rind = new Rind();
Futter gras = new Gras();
rind.friss(gras);  // Rind.firss(Futter x)
rind.friss((Gras)gras);  // Rind.friss(Gras x)
```

Es wird auf jeden Fall `friss` in Klasse `Rind` ausgeführt, egal ob `rind` als `Tier` oder `Rind` deklariert ist. Es zählt NUR der deklarierte Typ bei `gras` ⇒ es wird in der letzten Zeile `friss` von `Rind` ausgeführt!!!

Allgemeine Regel in Java (die aber nicht immer zutrifft): Es wird immer jene Methode mit den am besten passenden Argumenten ausgewählt! Wichtig ist Überladen und Überschreiben zu beachten! Überschrieben wird nur dann, wenn die Parameter in Java (außer den Rückgabewert) GLEICH sind!!!)

3.3.1 Simulation von Multimethoden

Multimethode verwenden *mehrfaches, dynamisches Binden*, welches in Java durch wiederholtes *einfaches dynamisches Binden* simuliert werden kann. Beispiel:

```
abstract class Tier {
  public abstract void friss(Futter x);
  ...
}
class Rind extends Tier {
  public void friss(Futter x) {
    x.vonRindGefressen(this);
  }
}
class Tiger extends Tier {
  public void friss(Futter x) {
    x.vonTigerGefressen(this);
  }
}
abstract class Futter {
  public abstract void
    vonRindGefressen(Rind rind);
  public abstract void
    vonTigerGefressen(Tiger tiger);
}
```

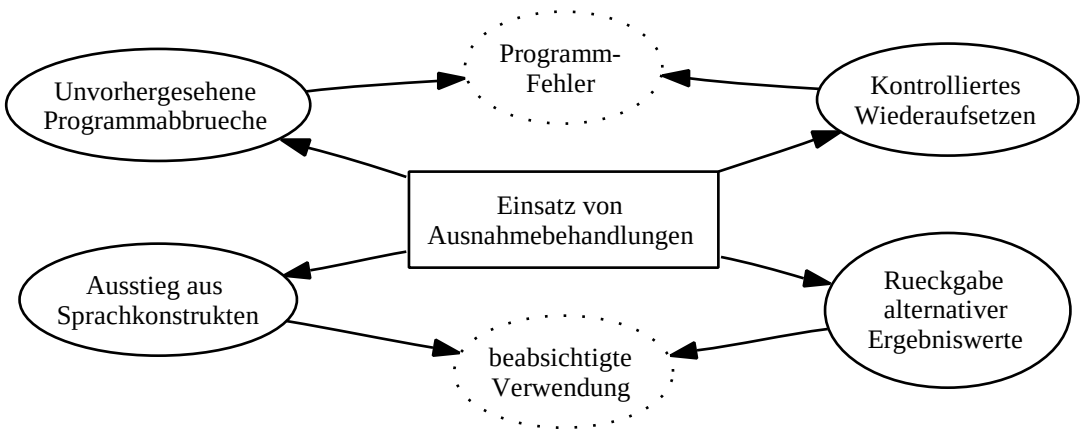
```
class Gras extends Futter {
public abstract void vonRindGefressen(Rind rind) {
   ... }
  public abstract void vonTigerGefressen(Tiger tiger) {
     tiger.fletscheZaehne();
  }
}
class Futter extends Futter {
public abstract void vonRindGefressen(Rind rind) {
  rind.erhoeheWahrscheinlichkeitFuerBSE();
}
  public abstract void vonTigerGefressen(Tiger tiger) {
   ... }
}
```

Beim Aufruf von `tier.friss(futter)` wird *2x* dynamisch gebunden. 1. Unterscheidung zwischen Instanz `Rind` (= `vonRindGefressen`) und `Tiger` (= `vonTigerGefressen`) und 2. Unterscheidung zwischen Instanz `Gras` und `Fleisch`.

Diese Lösung entspricht dem *Visor Pattern*. `Futter` = Visitor und `Tier` = Element. Nachtteil: Anzahl der Methoden wird rasch sehr groß!!!

3.4 Ausnahmebehandlung

In Java wird nach der 1. passenden `catch`-Klausel gesucht und diese verwendet (nicht nach der, die am besten passt)!



kontrolliertes Wiederaufsetzen: An einem Punkt weitermachen und Fehler (so weit es geht) korrigiern.
Ausstieg aus Sprachkonstrukten: Abbruch des gerade bearbeiteten Scope samt aufgerufener Unterfunktionen.
Rückgabe alternativer Ergebniswerte: Die Exception kann als alternativer Ergebniswert betrachtet werden.
Ausnahmebehandlungen sollen:

- sparsam
- nur bei Vereinfachung der Programmlogik
- unter der Berücksichtigung von nicht-lokalen Effekten

eingesetzt werden

3.5 Nebenläufige Programmierung - Threads

Synchronisation in `Java` mit `synchronized` - bei Blöcken wird ein *Lock* auf das Objekt gesetzt um es gegenüber anderen Methoden zu blockieren. Mit `wait` (und optionaler Zeitangabe) kann man auf die Aufhebung eines Locks warten. Mit `notifyAll` kann man ein Lock von einem Objekt wieder löschen (und alle Caller von `wait` damit aufwecken durch eine `InterruptedException` (mit `nofity` wird nur ein zufälliger aufgeweckt)).

`Collections.synchronizedList` erzeugt eine synchronisierte Liste. Achtung *Deadlocks* können bei massiven Einsatz von Synchronisation auftreten.

Unerwünschte gegenseitige Behinderungen von Threads in einem Programm nennt man *Liveness Properties*.

Es ist schwierig gute nebeläufige objektorientierte Programme zu schreiben!