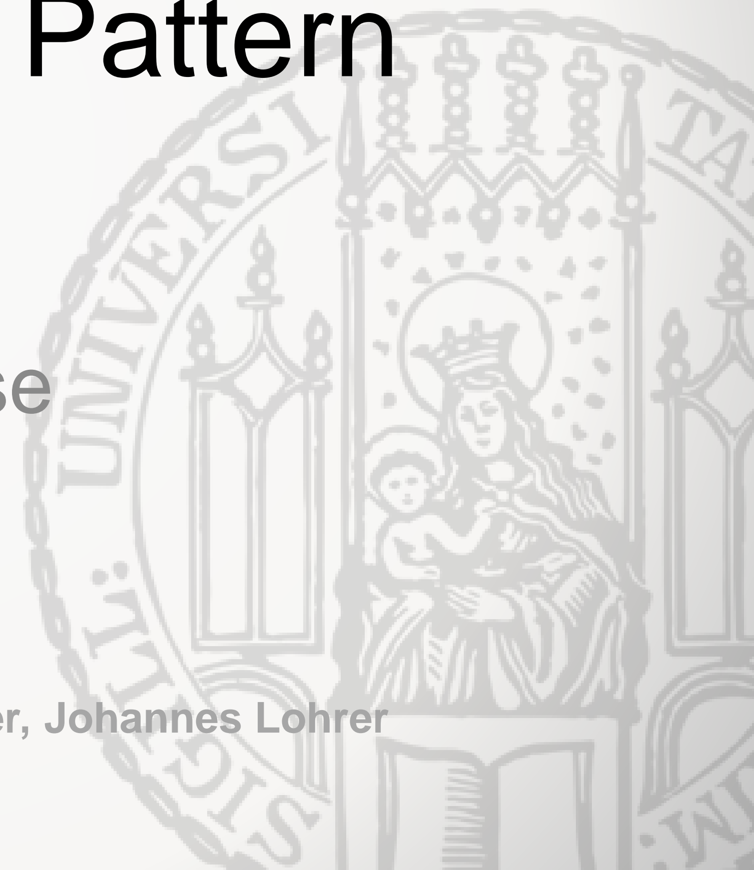


# Weitere Design Pattern

SEP

Emanuel Böse

13.11.2017

Wissenschaftlicher Betreuer:
**Janina Sontheim, Daniel Kaltenthaler, Johannes Lohrer**
Verantwortlicher Professor:
**Prof. Dr. Peer Kröger**

DBS

# Einführung

- Design Pattern als Lösungsmuster für oft auftretende Programmierprobleme
- Entwürfe sollen flexibel, wiederverwendbar, erweiterbar, sowie leicht veränderbar sein
- 1995 “Design Patterns. Elements of Reusable Object-Oriented Software” von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides

# Anforderungen

- Eines oder mehrere Probleme lösen
- Auf realen Designs basieren
- Ein erprobtes Konzept bieten
- Über das rein Offensichtliche hinausgehen
- Den Benutzer in den Entwurfsprozess einbinden
- Beziehungen aufzeigen, die tiefergehende Strukturen und Mechanismen eines Systems umfassen

*Quelle: https://de.wikipedia.org/wiki/Entwurfsmuster#Anforderungen_und_Nutzen*

# Nachteile

- Design Pattern werden oft als „Wunderwaffe“ für ein gutes Design gesehen
- Unerfahrene Entwickler könnten geneigt sein, möglichst viele Design Pattern zu verwenden
- Sind kein Garant für einen guten Entwurf

# Immutable Design Pattern

- „Unveränderlich“
- Bietet Schutz vor Manipulation
- Verbessert die Performance
- Nebenläufigkeit ohne Synchronisation möglich

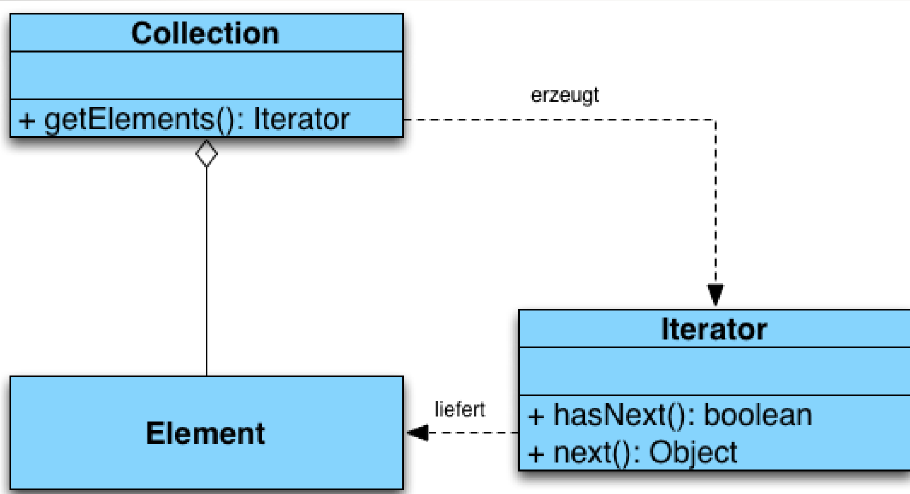
# Anforderungen

- Keine „setter“-Methoden zur Verfügung stellen
- Alle Instanzvariablen als „final“ und „private“ deklarieren
- Keine Vererbung zulassen (z.B. die Klasse als „final“ deklarieren)
- Vorsicht bei Instanzvariablen, welche auf „mutable“ Objekte zeigen

# Iterator Design Pattern

- Elemente einer Sammlung können der Reihe nach durchlaufen werden
- Es können gemischte Objekte in einer Sammlung gelagert werden
- Beispiele in Java:
  - List
  - Set
  - Map

# Aufbau eins Iterators

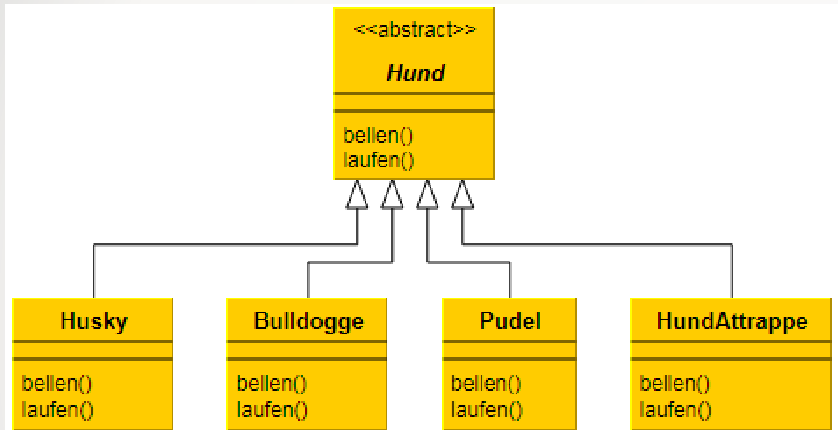



- Besteht mindestens aus den Methoden hasNext() und next()
- hasNext(): überprüft ob noch ein weiteres Element folgt
- next(): liefert das nächste Element zurück

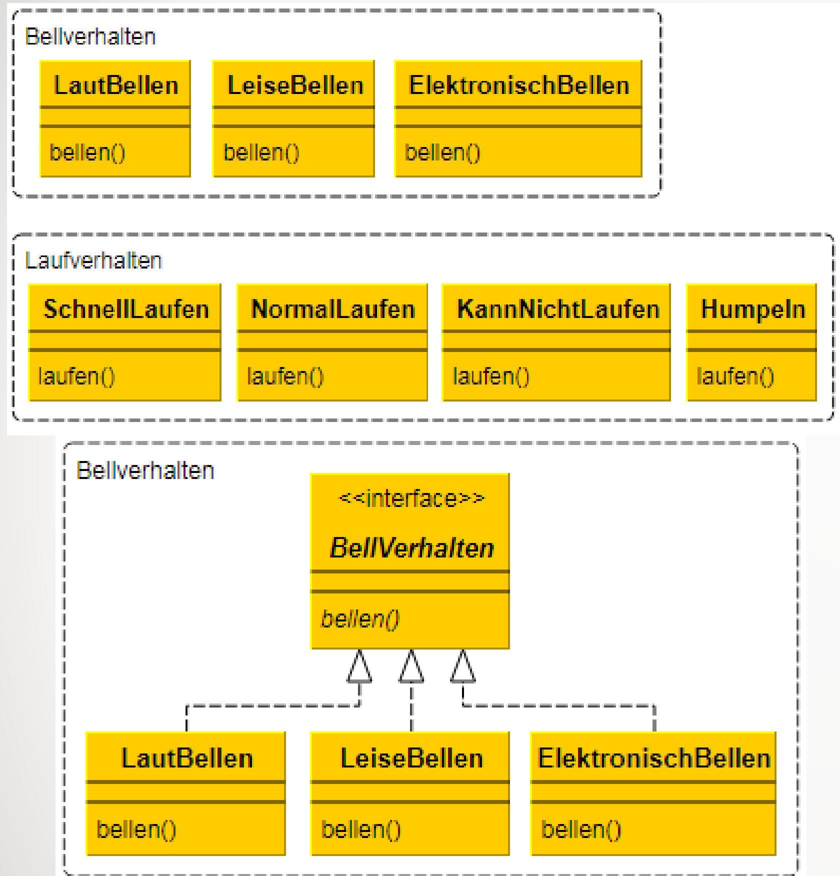
# Strategy Design Pattern

- Wird durch eine bestimmte Schnittstelle implementiert
- Verwendung bietet sich an, wenn:
  - Verwandte Klassen sich nur in ihrem Verhalten unterscheiden
  - Unterschiedliche Varianten eines Algorithmus benötigt werden
  - Daten innerhalb eines Algorithmus verborgen werden sollen

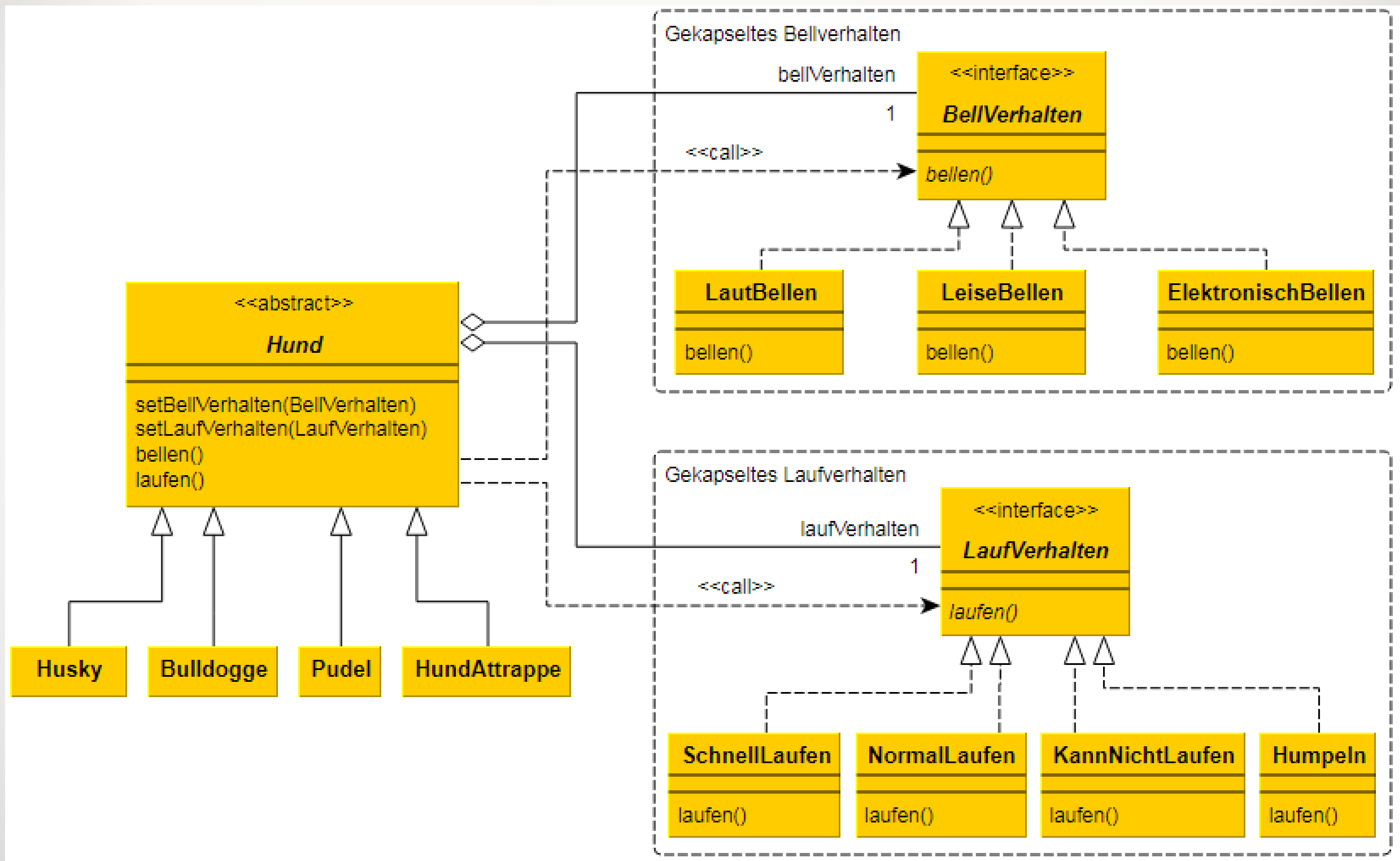
# Beispiel: Strategy Design Pattern




- Probleme:
  - Bellen() und laufen() werden immer vererbt, auch wenn HundAttrappe z.B. nicht laufen kann
  - Code Redundanz (ähnliches Verhalten muss doppelt geschrieben werden, Wartungsprobleme)
  - Verhalten der Hunde kann nicht zur Laufzeit geändert werden
  - Keine allgemeinen Aussagen über das Verhalten der Hunde
  - Keine Wiederverwendbarkeit (z.B. neue Klasse Schäferhund)




- Eine Schnittstelle für Verhaltensmuster erstellen (Bellverhalten)
- Es ist nur noch nötig, dass jeder Hund sein „Verhaltensobjekt“ kennt
- Hund „weiß“ nicht mehr welches Verhalten er besitzt, kann es jedoch nutzen
- Kann während der Laufzeit geändert werden



Gekapseltes Bellverhalten
bellVerhalten
1
<<call>>
<<interface>>
BellVerhalten
bellen()
LautBellen
bellen()
LeiseBellen
bellen()
ElektronischBellen
bellen()
<<abstract>>
Hund
setBellVerhalten(BellVerhalten)
setLaufVerhalten(LaufVerhalten)
bellen()
laufen()
Husky
Bulldogge
Pudel
HundAttrappe
Gekapseltes Laufverhalten
laufVerhalten
1
<<call>>
<<interface>>
LaufVerhalten
laufen()
SchnellLaufen
laufen()
NormalLaufen
laufen()
KannNichtLaufen
laufen()
Humpeln
laufen()

- Neue Hundearten können durch Kombination oder Erstellung neuer Verhaltensklassen kreiert werden
- Keine Code Redundanz bzw. doppelte Implementierung identischer Verhaltensmuster
- Wiederverwendbarkeit
- Wartungsfreundlich
- Nicht nur zur Designzeit, sondern auch zur Laufzeit können Verhalten geändert werden
- Verallgemeinerung von Verhalten