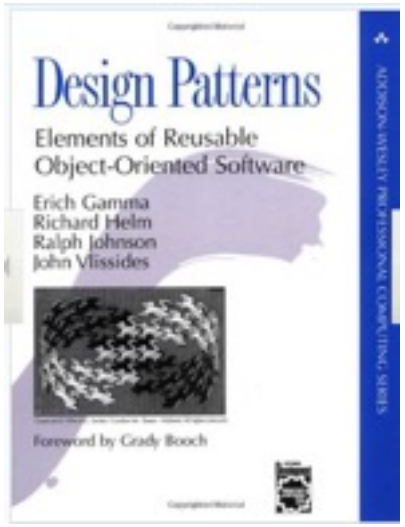
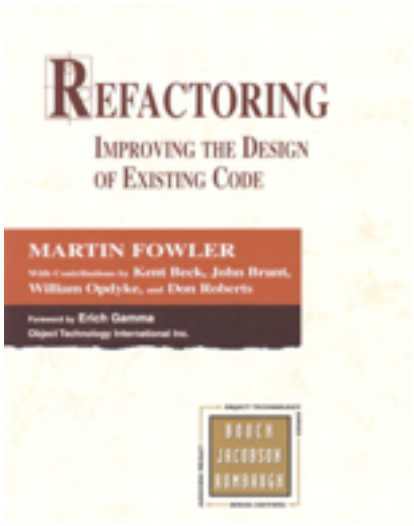
Entwurfsmuster

Programmiermethodik 2

Zum Nachlesen

- Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra: *Head First Design Patterns*, O'Reilly Media 2004
- E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*, Addison-Wesley
- M. Fowler: Refactoring, Addison-Wesley





Wiederholung

- Kritischer Abschnitt
- Monitor-Mechanismus
- Reihenfolge-Beschränkungen
- Deadlocks

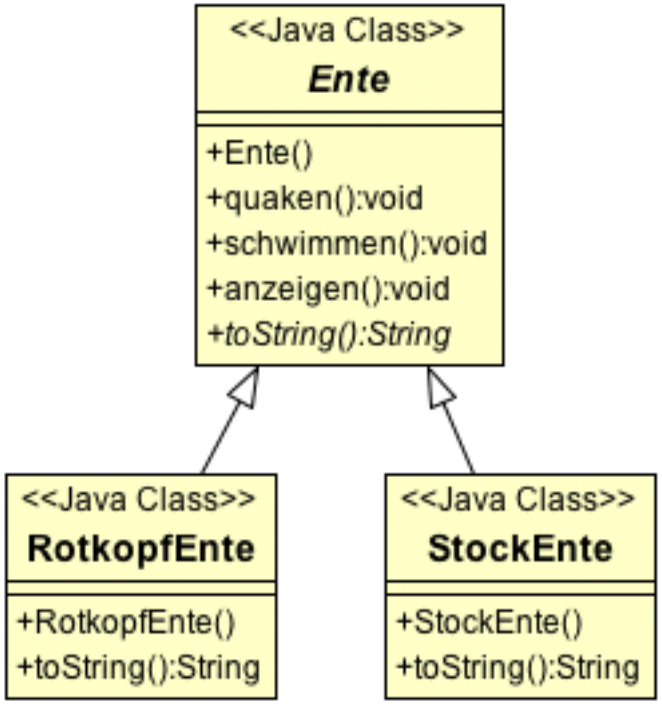
Ausblick

Agenda

- Einführung
- Strategie
- Beobachter
- Model-View-Controller

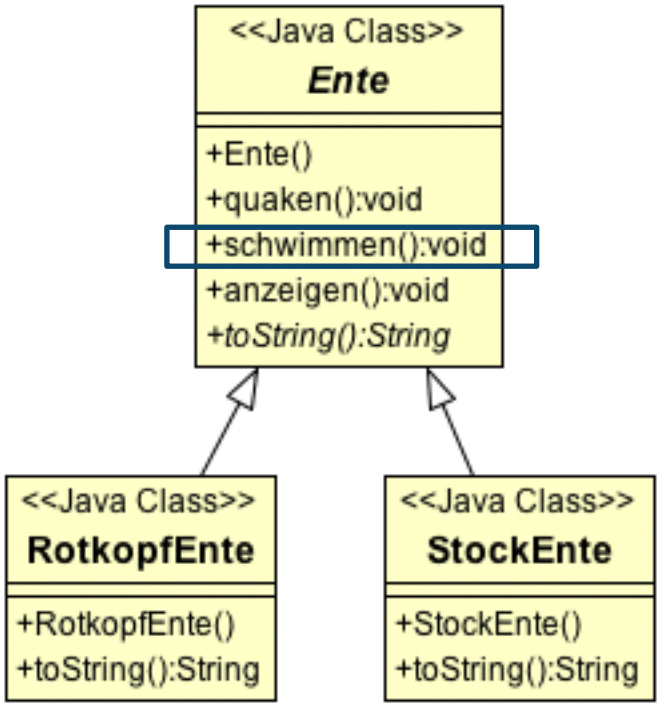
Einführung

- Erfolgreiche Simulation für Ententeich: SimDuck
  - mehrere Enten
  - Enten können schwimmen und quaken
- Objektorientiertes Design mit Enten-Superklasse



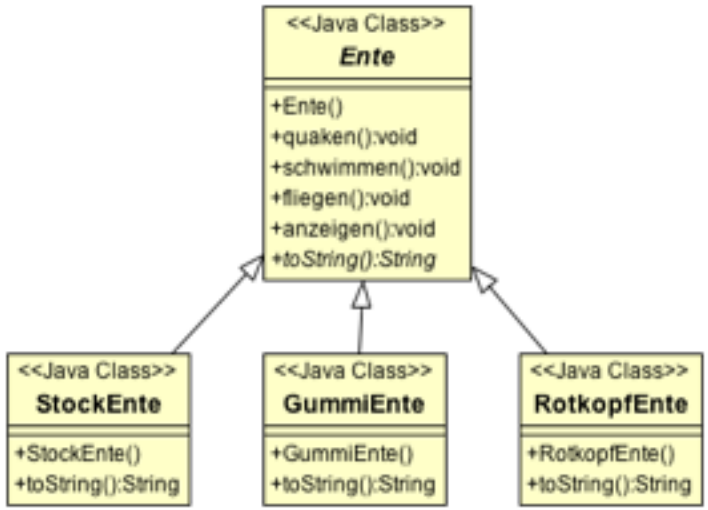
Einführung

- aber Konkurrenzdruck ➔ Zeit für Innovation
- Unternehmensleitung mit Idee: Enten sollen fliegen!
- Softwaretechnisch kein Problem
    - "Haben ja objektorientiert entwickelt!"



Einführung

- Aktionärsversammlung
    - Fliegende Gummienten!
- Was ist passiert
    - Annahme, dass ALLE Enten fliegen können war falsch
    - Methode fliegen() für manche Unterklassen nicht geeignet
- Was wie eine großartige Verwendung von Vererbung aussah, hat sich als weniger gut für Wartung erwiesen.



Einführung

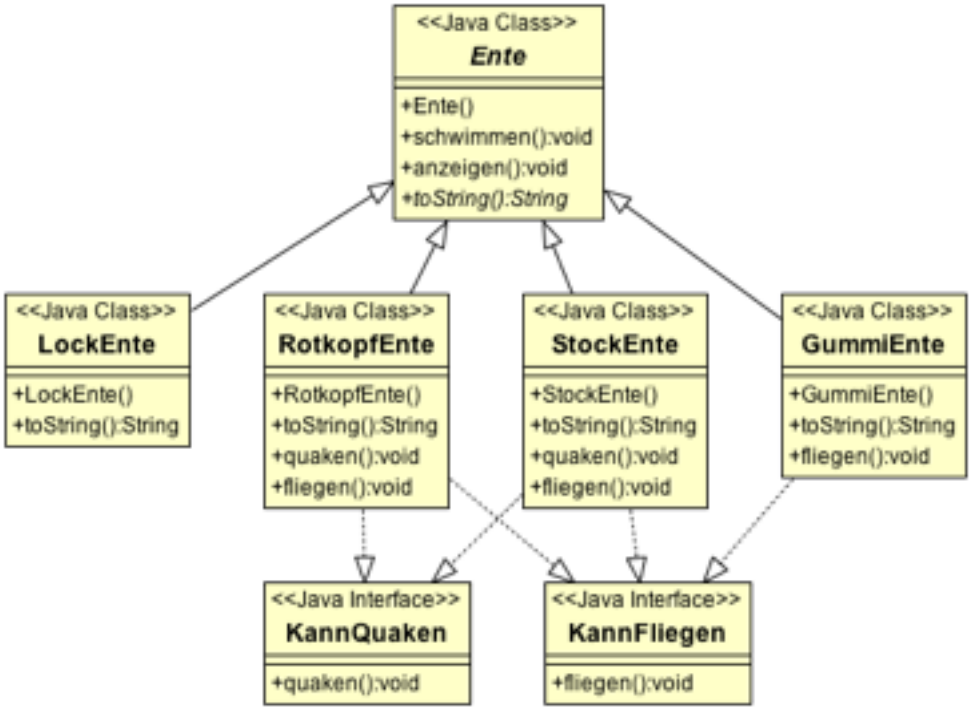
- Wie kann man das Design anpassen, damit es mit Gummienten korrekt umgeht?
- Überschreiben der Methode fliegen() in der Klasse GummiEnte?
- Aber was passiert dann, wenn Lockenten eingefügt werden
    - können weder fliegen noch quaken

Murmelgruppe: Vererbung

- Welche der folgenden Punkte sind Nachteile, wenn Sie Vererbung für das Entenverhalten einsetzen
  - Code wird über Unterklassen verdoppelt
  - Verhaltensänderungen zur Laufzeit sind schwierig
  - Wir können die Enten nicht zum "tanzen" bringen
  - Es ist nicht einfach Erkenntnisse über das Verhalten aller Enten zu erlagen
  - Enten können nicht gleichzeitig fliegen und quaken
  - Änderungen können sich unbeabsichtigt auf andere Enten auswirken

Interfaces?

- Irgendwie ist Vererbung hier nicht die Lösung
- Unternehmensleitung: Ab sofort ein Update alle 6 Monate
- Wie wäre es mit Interfaces?



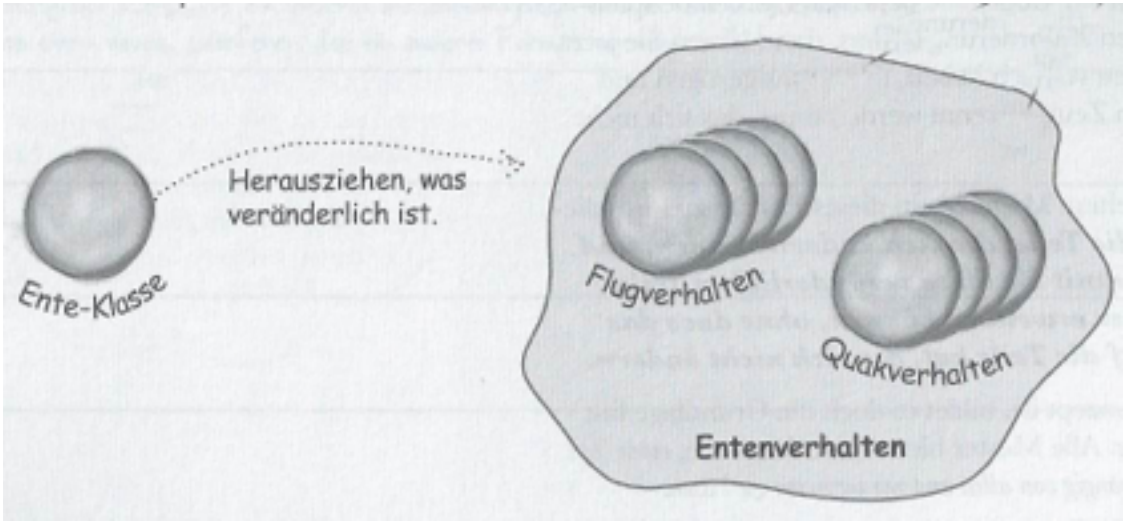
- Ist das die Lösung?
  - Nein! DRY-Prinzip (Don't repeat yourself), Code-Verdopplung!

Was Nun?

- Vererbung ist nicht die Lösung, weil sich manches Verhalten nicht für alle Unterklassen anbietet
- Interfaces gehen in die richtige Richtung, aber: DRY
- Daher: Entwurfsprinzip

Extraktion von Verhalten

- Was bedeutet das für das Design der Enten-Simulation?
  - Ente scheint generell gut zu funktionieren, nur Fliegen und Quaken machen Probleme.
  - Daher: Abtrennen der sich ändernden Funktionalitäten
  - Heraustrennen und Repräsentation in neuen Klassen



Einschub: Programmieren gegen Schnittstelle

- Konzept: Programmieren gegen Schnittstelle
    - Verwendung von Interfaces (nur öffentliche Schnittstelle)
    - dynamischer Typ: unterschiedliche Implementierungen
- Programmieren gegen Interface – Was bedeutet das für die Enten?
- Interfaces für verschiedene Verhaltensweisen
- neue Klasse je Verhalten
- vorher: auf Implementierung gestützt
    - entweder in Super-Klasse oder in abgeleiteter Klasse
- jetzt: Repräsentation als Interface

Einschub: Programmieren gegen Schnittstelle

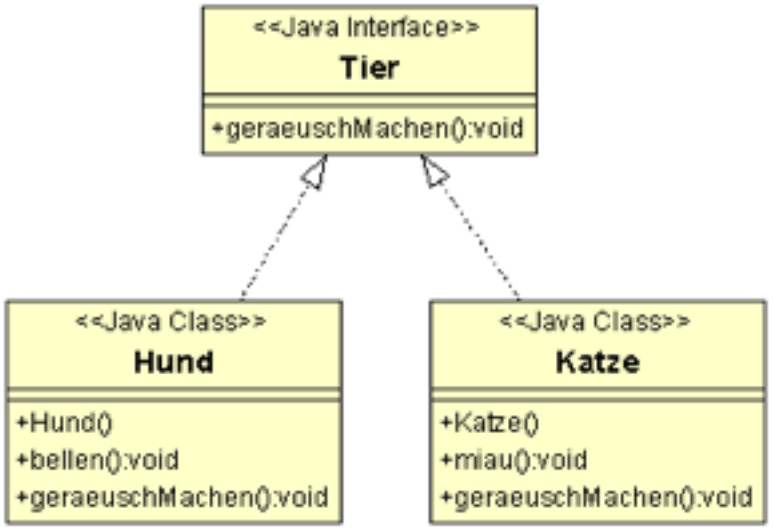
- Programmieren gegen Interface
  - Kann ich dann auch eine abstrakte Klasse verwenden?
  - Ja. Interface = Java-Interface oder Abstrakte Klasse

- Programmieren gegen Implementierung (nicht gut!)

```
Hund hund = new Hund();
hund.bellen();
```

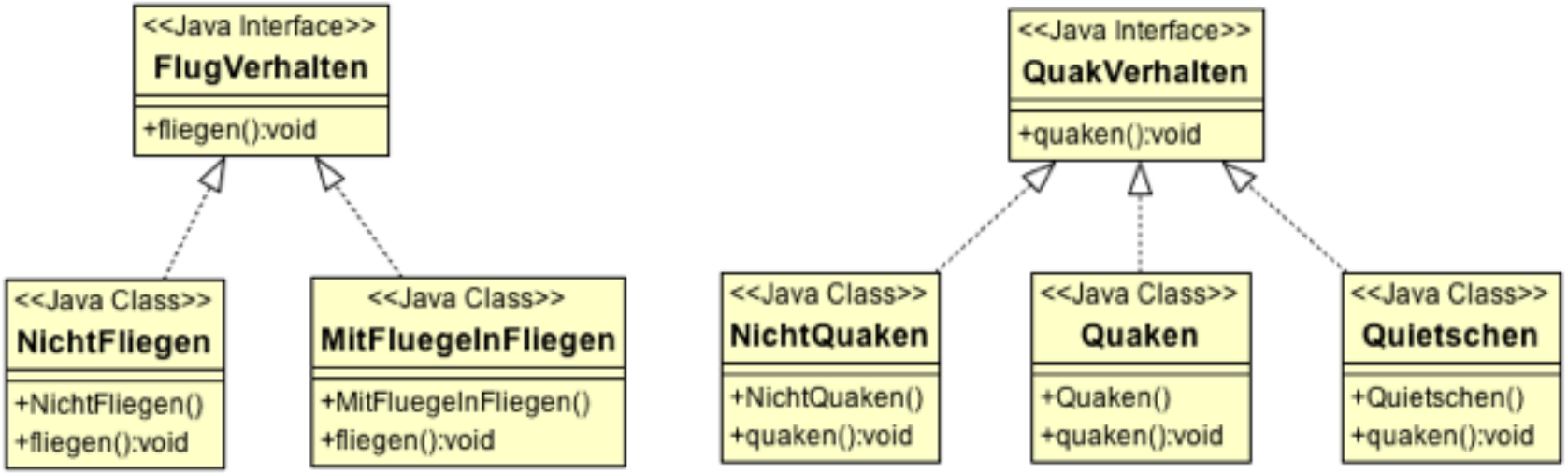
- Programmieren gegen Interface (gut!)

```
Tier tier = new Hund();
tier.geraeuschMachen();
```



Verhaltensinterfaces

- Idee: Kapselung in Verhaltens-Interfaces



- Verhalten kann bei neuen Ententypen wiederverwendet werden
- neue Verhaltensweisen können eingebaut werden, ohne bestehende Implementierungen zu verändern
- alle Vorteile der Wiederverwendung ohne den Ballast der Vererbung!

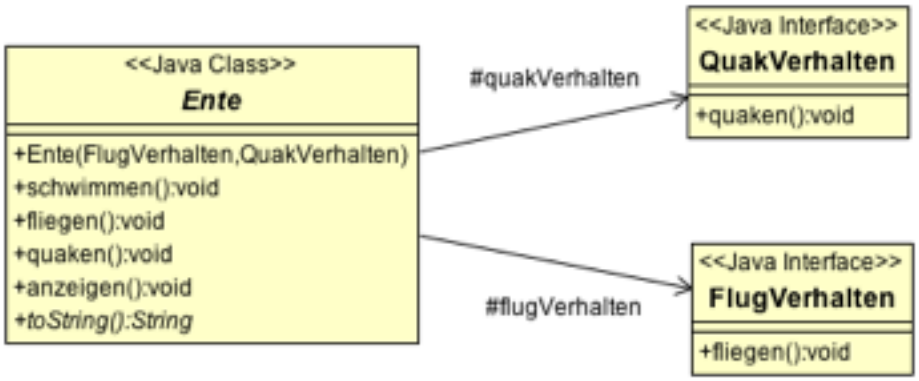
Murmelgruppe: Raketenantrieb

- Frage 1: Was machen Sie, wenn Sie der SimDuck-Anwendung unter Verwendung unsers neuen Entwurfs raketengetriebenes Fliegen hinzufügen müssten?

- Frage 2: Fällt Ihnen eine Klasse ein, die Quakverhalten nutzen könnte, ohne eine Ente zu sein?

Verhaltensobjekte

- Letzter Schritte: Entenverhalten mit den Enten zusammenbringen
- Ente bekommt zwei Objektvariablen:
    - flugVerhalten
    - quakVerhalten



- Verhalten werden in fliegen() und quaken() aufgerufen
- Abgeleitete Klassen setzen Verhaltensobjekte im Konstruktor

Beispiele

```
Ente rotKopfEnte = new
RotkopfEnte();
Ente gummiEnte = new GummiEnte();
rotKopfEnte.fliegen();
rotKopfEnte.quaken();
rotKopfEnte.anzeigen();
gummiEnte.fliegen();
gummiEnte.quaken();
gummiEnte.anzeigen();
```

```
public class RotkopfEnte extends
Ente {
  public RotkopfEnte( ) {
    super(new MitFluegelnFliegen( ),
        new Quaken( ));
  }

  @Override
  public String toString( ) {
    return "Rotkopfente";
  }
}
```

```
public class GummiEnte extends
    Ente {
  public GummiEnte( ) {
    super(new NichtFliegen( ),
        new Quietschen( ));
  }
  @Override
  public String toString( ) {
    return "Gummiente";
  }
}
```

Ich fliege!

Quak!

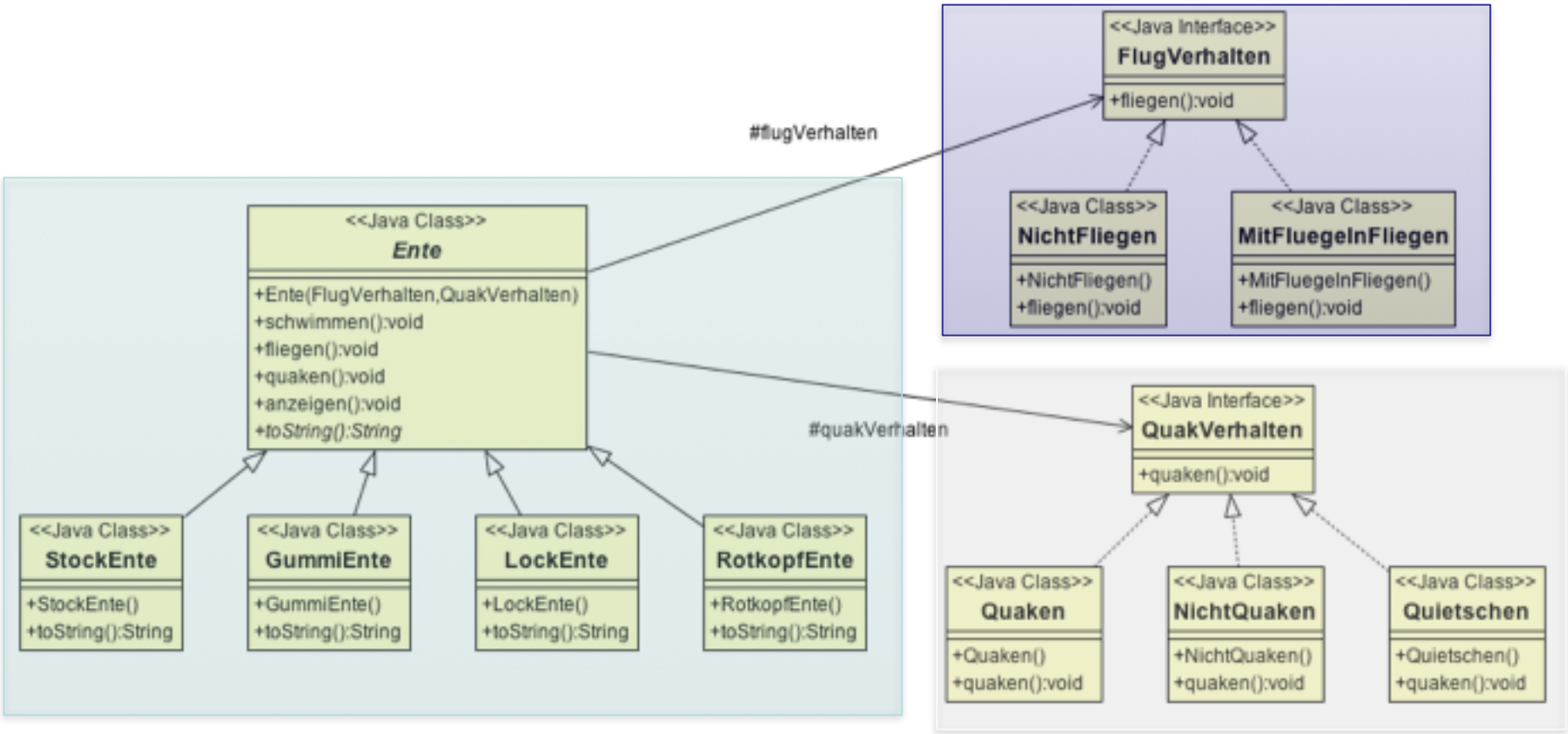
Rotkopfente

Ich kann nicht fliegen!

Quietsch!

Gummiente

Ein Blick zurück



Was haben wir gemacht?

- Relationen haben sich verändert
- vorher: viele "IST-EIN"-Beziehungen
- jetzt: viele "HAT-EIN"-Beziehungen

Entwurfsmuster Strategie

- Nebenbei haben Sie Ihr erstes Entwurfsmuster kennengelernt: Strategie (engl. *Strategy*)
- Definition

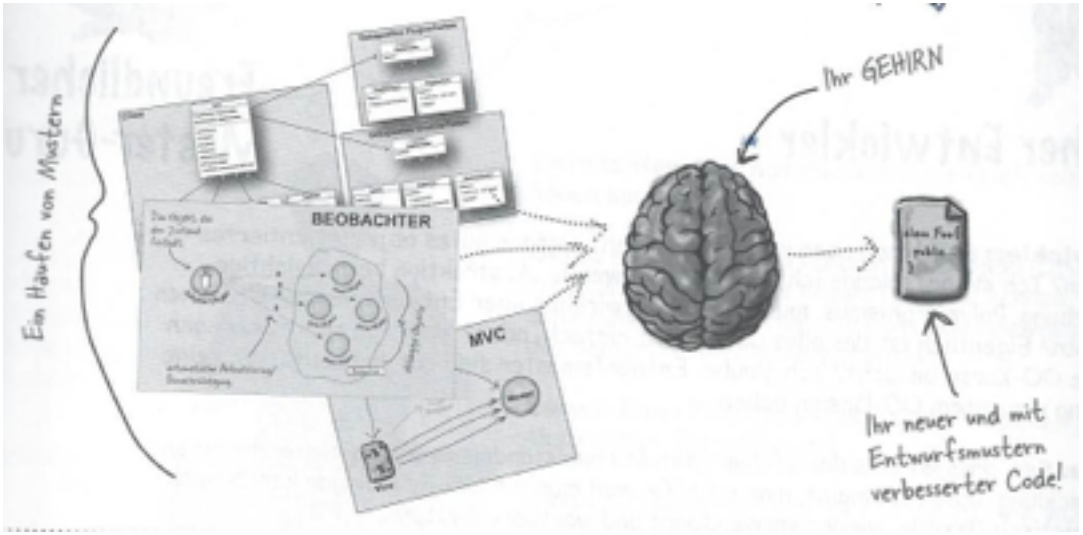
> *Das Strategie-Muster definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Das Strategie-Muster ermöglicht es, den Algorithmus unabhängig von den Clients, die ihn einsetzen (hier: konkrete Enten-Typen), variieren zu lassen.*

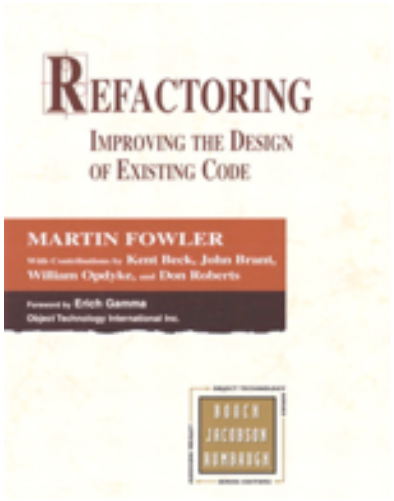
Sprechen über Entwurfsmuster

- Verwenden Sie immer die Namen der Muster, wenn Sie sie einsetzen und über Ihre Architektur sprechen
- Verwenden Sie die Namen wenn passend auch in den Bezeichnern für Klassen, Interfaces, Variablen ...
    - Beispiel: interface QuakenVerhaltenStrategie
- Warum?
    - Muster sind mächtig
    - Weniger Worte notwendig
    - Es wird über das "Wichtige" gesprochen (Architektur statt "Klein-Klein")
    - Weiterentwicklung über gemeinsames Vokabular

Wann werden Entwurfsmuster eingesetzt?

- Muster müssen sich in Ihrem Gehirn verfestigen
- Dann finden Sie automatisch die richtigen Stellen für den Einsatz
- Dann können Muster auch verwendet werden, um bestehenden Code zu verbessern/überarbeiten
    - dies nennt man Refactoring





Literaturempfehlung zum Selbststudium

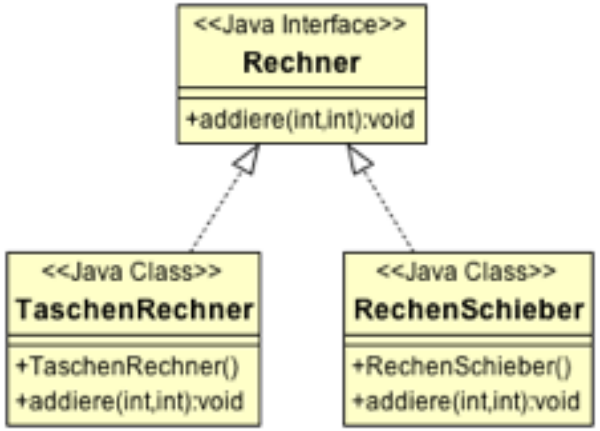
Anwendungsfälle

- Möglichkeit, dass neue Typen hinzukommen
    - die ähnliche Schnittstelle haben
    - die aus ähnlicher Kategorie (Basisklasse) kommen
    - unterschiedliches Verhalten haben
- Vermeidung von Quellcode-Verdopplung
- dennoch: Wiederverwendbarkeit von "Verhalten"

- Hinweis:
    - Schwester-Muster: Zustand (engl. State)
    - ähnliche Umsetzung
    - dynamische Verwaltung von Zustand (anstelle von Verhalten)
    - oder: zustandsabhängige Veränderung von Verhalten

Übung: Rechner

- Gegeben ist folgende Klassenhierarchie:



addiere(2,3):
Ausgabe (TaschenRechner):
Tippen, tippen, anzeigen: 5
Ausgabe (RechenSchieber):
*Schieb, schieb: ******

- Entwickeln Sie eine alternative Architektur, die das Strategie-Muster einsetzt.

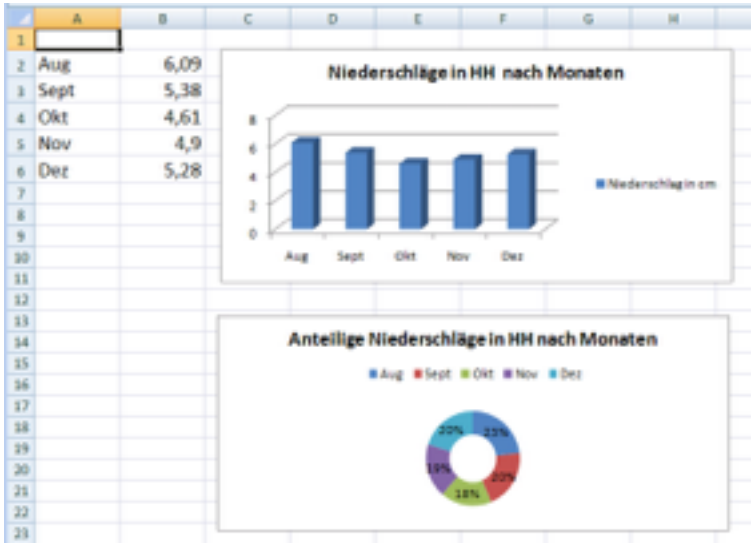
Beobachter (Observer)

Beobachter (Observer)

- Problem
  - Änderungen eines Objektes
  - alle abhängigen Objekten müssen informiert werden
  - ggf. Anpassen des Zustands
- Lösung
  - Registrierung von abhängigen Objekten als "Abonnenten"

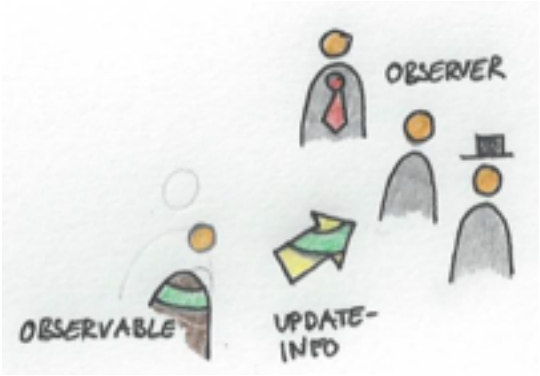
Beispiel

- drei Darstellungen von Niederschlagsmengen
  - Tabelle
  - Säulendiagramm
  - Kuchendiagramm
- Änderung der Werte
  - Aktualisieren aller abhängigen Grafiken
- Idee Observer-Muster
  - Grafiken beobachten die Tabellendaten
  - werden über jedes Änderungsereignis informiert



Entwurfsmuster

- Registrierung mehrerer Beobachter bei einem zu beobachtendem Objekt
- Änderungs-Ereignis bei beobachtetem Objekt
    - Informationen aller registrierten Beobachter durch Aufruf einer Methode
- Benennung
    - Beobachter: Observer
    - beobachtetes Objekte: Observable

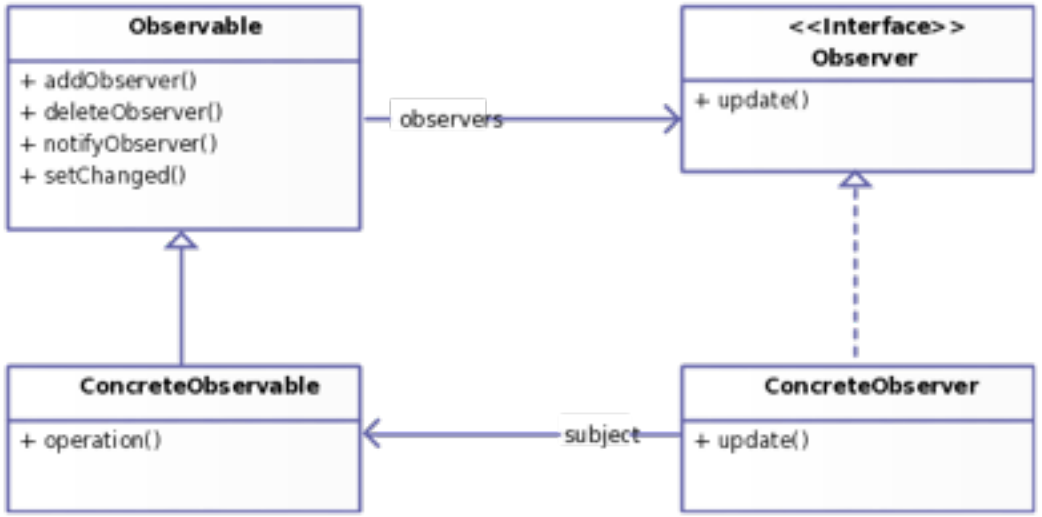


Anwendungsbeispiele

- Entwicklung Grafischer Benutzeroberflächen
  - Observer registriert sich bei einer GUI-Komponente (Observable)
  - Änderung der GUI-Komponente
  - jeder registrierte Beobachter erfährt per Methodenaufruf von dem Ereignis

Umsetzung in Java

- Interface Observer
- Klasse Observable
- Registrierung
  - Observable-Methode: addObserver()
- Abmeldung
  - Observable-Methode: deleteObserver()
- Update-Info
  - Observable-Methode notifyObservers()
  - zuvor: Anzeigen der Veränderung durch setChanged()
  - Observer implementieren die Interface-Methode update()
    - bekommt Referenz auf Observable-Objekt



Umsetzung in Java

- Klasse <Klassen-Bezeichner> extends Observable
    - repräsentiert einen (beobachteten) <Klassen-Bezeichner>
- Klasse <Klassen-Bezeichner> implements Observer
        - reagiert auf Änderungen eines Observables

Beispiel: Daten-Ausgabe

- Daten sollen auf zwei unterschiedliche Formen dargestellt werden.
    - später vielleicht mehr
- Daten sollen nichts über Darstellungen wissen.
- Darstellung soll sich aktualisieren, wenn sich die Daten ändern.
- Daten: eine Ganzzahl

```
public class Daten extends Observable {

  private int zahl = 0;

  public int getZahl() {
    return zahl;
  }

  public void setZahl(int zahl) {
    this.zahl = zahl;
    setChanged();
    notifyObservers();
  }
}
```

Beispiel: Daten-Ausgabe

- Ausgabe 1: arabische Zahlen

```
public class DatenArabischeZahlAusgabe
    implements Observer {

  public final Daten daten;

  public DatenArabischeZahlAusgabe(
      Daten daten) {
    this.daten = daten;
    daten.addObserver(this);
  }

  @Override
  public void update(Observable o,
      Object arg) {
    System.out.println("Daten: "
        + daten.getZahl());
  }
}

Daten daten = new Daten();
new DatenArabischeZahlAusgabe(daten);
DatenSymbolischeAusgabe symbolischeAusgabe = new
DatenSymbolischeAusgabe();
daten.addObserver(symbolischeAusgabe);
```

- Ausgabe 2: Symbole

```
public class DatenSymbolischeAusgabe
    implements Observer {

  @Override
  public void update(Observable o,
      Object arg) {
    if (o instanceof Daten) {
      Daten daten = (Daten) o;
      System.out.print("Daten: " +
          ((daten.getZahl() < 0) ? "-" : ""));
      for (int i = 0; i <
          Math.abs(daten.getZahl()); i++) {
        System.out.print((char) 9760);
      }
      System.out.println();
    }
  }
}
```

Anwendungsfälle

- Entkopplung von Beobachter und Beobachtetem
  - Beobachteter weiß nichts über Beobachter
  - Anzahl Beobachter kann sich jederzeit ändern
  - Hinzufügen eines neuen Beobachters erfordert keine Veränderung der bestehenden Implementierung
- Beobachter (nicht Beobachteter) entscheidet,
  - wie mit einer Veränderung umgegangen wird
  - ob mit einer Veränderung umgegangen wird

Übung: Zähler-Thread-Beobachten

- Schreiben Sie zwei Klassen mit folgendes Funktionalität:
  - Klasse 1 als Beobachteter: Thread, in dem von 1-10 gezählt wird, je 100 ms Pause
  - Klasse 2 als Beobachter: Ausgabe des Zählerstandes auf der Konsole
- Geben Sie auch den Code für die Registrierung des Beobachters beim Beobachteten an.

- Hinweise
  - Klasse Observable: Methoden
  - setChanged(), notifyObservers(), addObserver(Observer)
  - Interface Observer: Methode
    - public void update(Observable o, Object arg)

Model-View-Controller (MVC)

Model-View-Controller (MVC)

- Entwurfsmuster für die Entwicklung von graphischen Oberflächen
    - war bereits in Smalltalk realisiert
- zugrunde liegendes Konzept bei der Entwicklung mobiler Anwendungen (Apps)
- Trennung zwischen
    - Datenhaltung: Modell (engl. model)
    - Darstellung (engl. view)
    - Interaktion mit den der Daten (engl. controller)

Model-View-Controller (MVC)

Modell

- funktionaler Anwendungskern
- Kapselung in Daten und Methoden
- Methoden werden vom Controller aufgerufen
- Daten werden über die Darstellung abgefragt

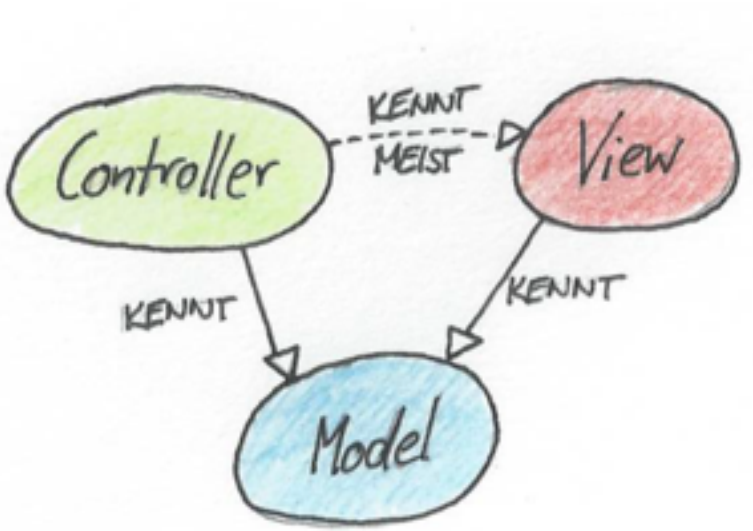
Darstellung

- Präsentiert die Daten des Modells
- häufig: update()- oder paint()-Methode
  - Aufruf bei Änderungen
- Verbindung zum Modell meist bei Initialisierung
  - ggf. über den Controller

Model-View-Controller (MVC)

Controller

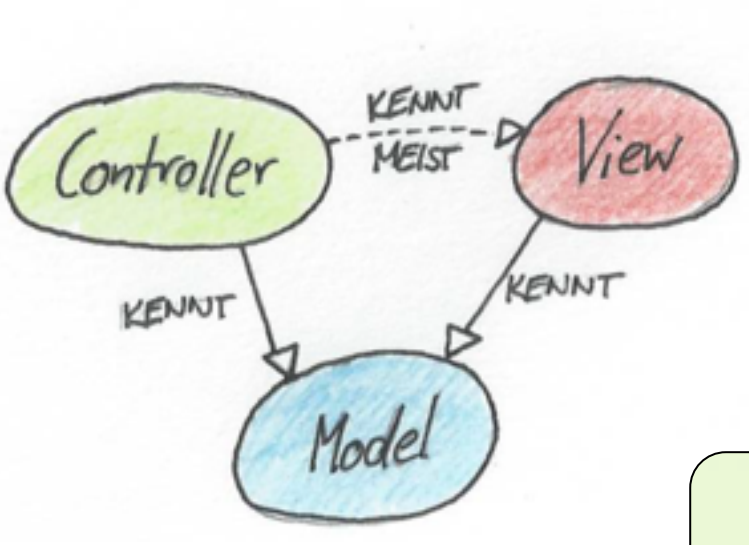
- Interpretiert Benutzereingaben
- Bewirkt Änderungen des Modells
    - und damit meist Anpassung der Darstellung



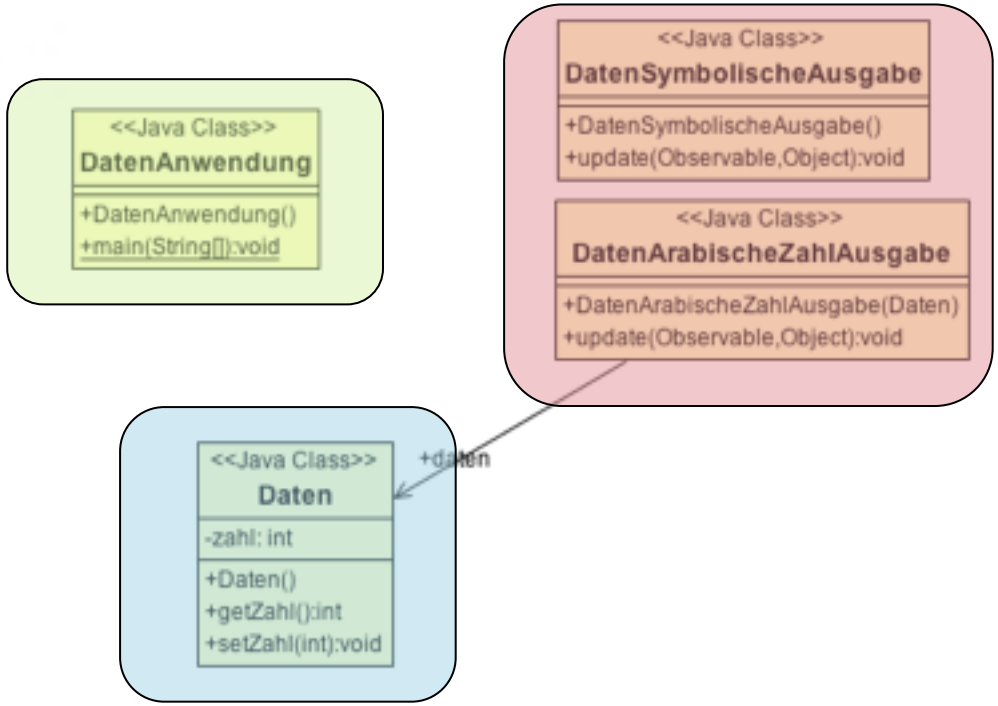
Model-View-Controller (MVC)

- Wir erkennen: MVC enthält weitere Muster
- insbesondere Beobachter!

Beispiel



- Anwendung bereits im Daten-Beispiel bei Beobachter!



Anwendungsfälle

- im Prinzip bei jeder größeren Anwendung
- Anwendungskern (Modell)
- verschiedene Formen der Ausgabe (Darstellung)
    - grafische Benutzerschnittstelle
    - Tabellen, Diagramme
    - Webseite
    - Ausgabe-Stream ...
- Interaktionsmöglichkeit (Controller)
    - oft verzahnt mit Darstellung
    - Konsoleneingabe
    - Web-Schnittstelle
    - ...

Zusammenfassung

Zusammenfassung

- Strategie (Strategy)
- Beobachter (Observer)
- Model-View-Controller