Entwurfsmuster - Iterator & Composite

Alexander Rausch

Seminar Entwurfsmuster WS08/09

19. November 2008

Gliederung

Einführung

- Ein Beispiel zur Veranschaulichung
  - bestehende Hotelmanagement-Software soll erweitert werden
  - Software besitzt Module für Raum- und Veranstaltungsverwaltung

Problemstellung

Klassen benutzen Datenstrukturen (ArrayList, HashMap...), um Sammlungen von Objekten zu verwalten.

- solche Klassen werden auch als Aggregate bezeichnet
- Objekte einer Sammlung heißen Elemente

Problemstellung

- Client...
- möchte diese Sammlung durchlaufen
- möchte auf die einzelnen Elemente zugreifen
- arbeitet mit Aggregatklassen, die unterschiedliche Datenstrukturen nutzen

Beispiel für das Problem

Ein Lösungsansatz

```
public void showOverview() {
  Room[] allRooms = roomManagement.getRooms();
  ArrayList<Event> allEvents = eventCalendar.getEvents();

  /* Alle Räume anzeigen */
  for(int i=0; i < allRooms.length; i++) {
      if(allRooms[i] != null) {
          allRooms[i].printInfo();
      }
  }

  /* Alle Veranstaltungen anzeigen */
  for(int i=0; i < allEvents.size(); i++) {
          allEvents.get(i).printInfo();
  }
}
```

Nachteile

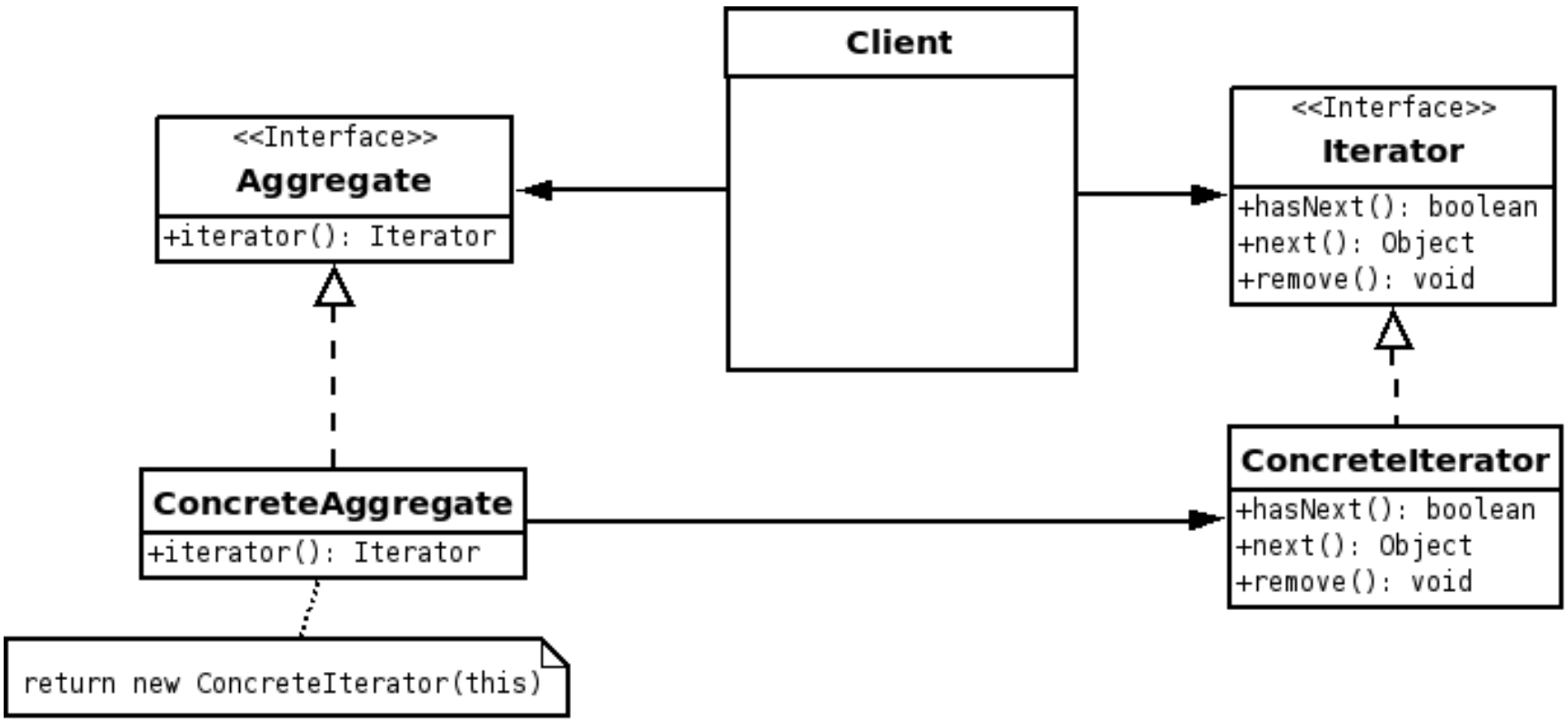
- Client muss sich um Traversierungslogik kümmern
- starke Kopplung zwischen Client und Aggregat
- Wartung wird erschwert
- doppelter Code für gleiche Problematik

Das Iterator Entwurfsmuster

- objektbasiertes Verhaltensmuster
- bietet die Möglichkeit auf Elemente in einer Aggregatklasse sequentiell zuzugreifen
- erlaubt es mehrere Aggregate mit dem gleichen Code zu durchlaufen
- Client muss nicht die Implementierung des Aggregats kennen
- entkoppelt Aggregat und Logik für die Iteration

Hohe Kohäsion

Iterator Struktur



Teilnehmer - Iterator

- legt Methoden fest, die ein Client für das Traversieren benötigt
- alle ConcreteIterator Klassen müssen dieses Interface implementieren

```
public interface Iterator {

   public boolean hasNext();

   public Object next();

   public void remove();
}
```

Teilnehmer - ConcreteIterator

- Konkreter Iterator für bestimmtes Aggregat
- enthält die Logik zum Durchlaufen der Collection
- implementiert eine konkrete Traversierungsart
- Beispiel:
  - Vorwärts / Rückwärts traversieren
  - Filter anwenden

Teilnehmer - ConcreteIterator

```
public class ConcreteIterator implements Iterator {
  private List[] collection;
  private int position;

  public ConcreteIterator(ConcreteAggregate aggregate) {
     /* Referenzen speichern */
  }
  public boolean hasNext() {
     /* Prüfen ob ein weiteres Element vorhanden ist */
  }
  public Object next() {
   /* Aktuelles Element ausgeben
    * und Position inkrementieren */
  }
  public void remove() {
  /* Optional: Aktuelles Element löschen */
  }
}
```

Teilnehmer - Aggregate

- Schnittstelle für den Client zu der Aggregatklasse(n)
- Minimum: Methode, welche passenden Iterator liefert
- Beispiel für eine Factory-Methode

```
public interface Aggregate {
  public Iterator iterator();
}
```

Teilnehmer - ConcreteAggregate

- enthält eine Collection (Sammlung) von Objekten
- implementiert iterator() Methode
- liefert passenden Iterator
- kümmert sich nur um das Verwalten der Collection

Teilnehmer - ConcreteAggregate

```
public class ConcreteAggregate implements Aggregate {
  private List<ClassOfElement> collection;

  /* Verwaltunsmethoden für die Collection */
  public void addElement(Object o) {
  . . .
  }

  public void removeElement(Object o) {
  . . .
  }

  public List<ClassOfElement> getElements() {
  . . .
  }

  /* Liefert den konkreten passenden Iterator für diese
      Collection */
  public Iterator iterator() {
    return new ConcreteIterator(this);
  }
}
```

Teilnehmer - Client

- muss nur Interfaces Aggregate und Iterator kennen
- ist sicher vor Änderungen am Aggregate-Code
- flexibel dank Iterator Interface

```
/* Aggregate-Klasse mit irgendeiner Liste von Objekten */
Aggregate aggregate = new ConcreteAggregate(alist);
/* Passenden Iterator holen */
Iterator myIterator = aggregate.iterator();

while(myIterator.hasNext()) {
  /* das aktuelle Element geben lassen */
  ClassOfElement element = (ClassOfElement) myIterator.
      next();
  /* oder man benutzt ein Interface */
  CommonInterface element = (CommonInterface) myIterator.
      next();
}
```

Interne und externe Iteratoren

Es gibt zwei Arten von Iteratoren

- externe Iteratoren
  - Client...
  - steuert Iteration
  - schaltet auf das nächste Element weiter
  - kann beliebige Funktionalität implementieren
- interne Iteratoren
  - Client...
  - teilt Iterator eine Operation mit
  - erhält fertiges Ergebnis der Operation zurück
  - Iterator traversiert intern eine Collection

Nulliterator

Ein Nulliterator

- liefert bei hasNext() immer false
- nützlich für Aggregate, die eigentlich keine Liste enthalten, der Client sie aber behandeln soll, als hätten sie eine
- Beispiel: Blätterklasse im Composite Muster

Anwendungen

- implementiert in fast allen OO-Sprachen
- Java Collection-Framework
  - jede Behälterklasse (ArrayList...) hat einen Iterator, der java.util.Iterator implementiert
- C#: System.Collections.IEnumerator
  - Klassen die GetEnumerator() implementieren lassen sich mit foreach-Konstrukt durchlaufen
- in Verbindung mit den Mustern
  - Composite (Durchlaufen der Kinder)
  - Memento (Iterator verwendet es, um Schritte einer Iteration zwischenzuspeichern)

Vor - und Nachteile

Vorteile

- Client kann die Elemente eines Aggregates sequentiell traversieren
- Implementierung des Aggregats muss dem Client nicht bekannt sein
- Entkopplung von Aggregatklasse und Logik zum Iterieren
- fördert Hohe Kohäsion

Nachteile

- erhöhte Laufzeit- und Speicherkosten

Beispiel

Es folgt ein Beispiel...

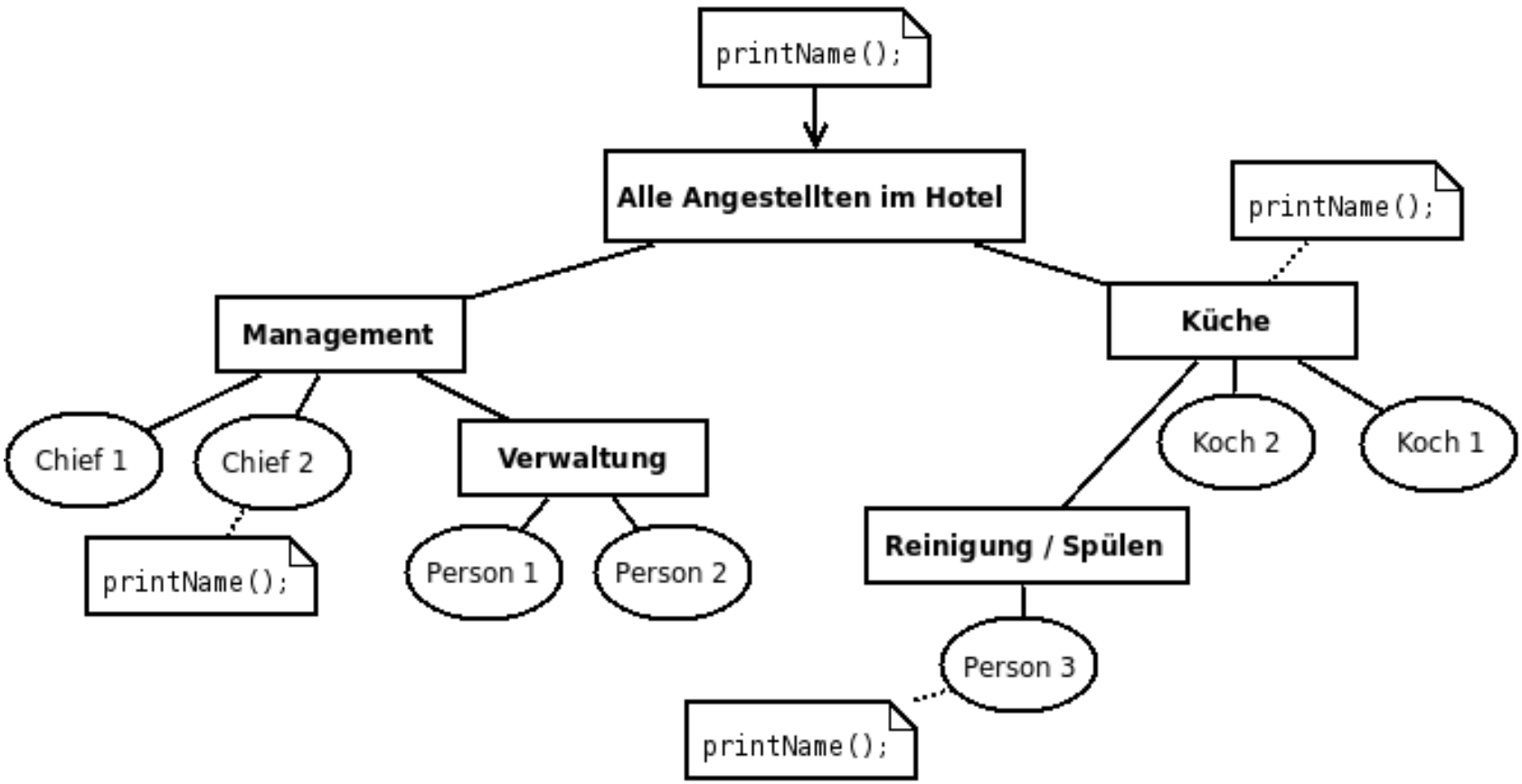
Baumstrukturen - Ein neues Problem

In der Informatik gibt es viele Konstrukte, die in einer Baumstruktur aufgebaut sind.

- Beispiele
  - Dateisysteme bestehen aus Verzeichnissen, Dateien
  - Grafiken setzten sich aus geometrischen Objekten zusammen
  - grafische Oberflächen in Java Swing fügen sich aus Frames, Panels, Buttons, Labels etc. zusammen
  - HTML Dokumente werden in einem DOM-Baum organisiert und setzten sich aus Tags zusammen

Gemeinsame Operationen

Was jetzt, wenn ein Client auch noch Operationen auf Blätter und Containerklassen ausführen möchte?

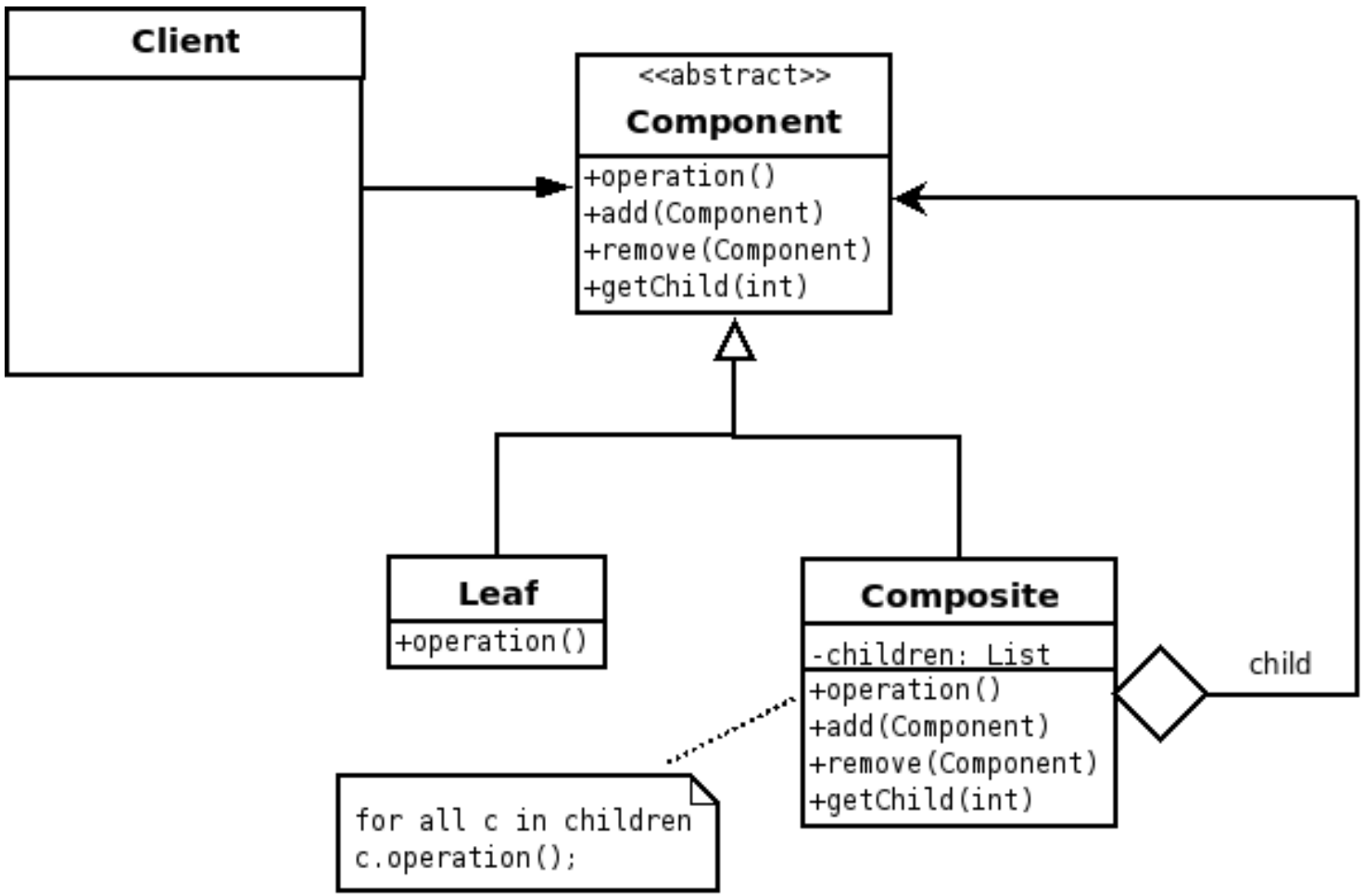


Das Composite Entwurfsmuster

Für solch einen Anwendungsfall gibt es das Composite Entwurfsmuster!

- objektbasiertes Strukturmuster
- erzeugt Baumstrukturen, welche Teil-Ganzes Hierarchien repräsentieren
- Client kann gemeinsame Operationen auf primitive und zusammengesetzte Objekte ausführen
- Client kann diese beiden gleich behandeln

Composite Struktur



Teilnehmer - Component

- eine abstrakte Klasse, welche primitive und zusammengesetzte Klassen repräsentiert
- dient dem Client als Schnittstelle zur Baumstruktur
- enthält Methoden, die für Leaf und Composite genutzt werden
- definiert Methoden zur Kindverwaltung
- bietet Defaultimplementierung
- lässt Methodenaufrufe defaultmäßig fehlschlagen (Exception)

Teilnehmer - Component

```
public abstract class Component {
  public void add(Component c) {
     throw new UnsupportedOperationException();
  }

  public void remove(Component c) {
     throw new UnsupportedOperationException();
  }

  public Component getChild(int i) throws {
     throw new UnsupportedOperationException();
  }

  public void operation() {
     throw new UnsupportedOperationException();
  }
}
```

Teilnehmer - Composite

- repräsentiert zusammengesetzte Objekte (Behälterklasse)
- enthält Referenzen auf Kind-Objekte (Leaf oder Composite)
- führt operation() aus und ruft sie rekursiv bei allen Kindern auf
- zum Traversieren der Kinder: Iterator Muster

Teilnehmer - Composite

```
public class Composite extends Component {
  private List<Component> children;

  public void add(Component c) {
  /* Blatt anhängen */
  }

  public void remove(Component c) {
  /* Blatt löschen */
  }

  public Component getChild(int i) {
   return children.get(i);
  }

  public void operation() {
  /* Eigene Instruktionen ausführen und
   * rekursiv bei allen Kindern aufrufen */
  }
}
```

Teilnehmer - Leaf

- repräsentiert ein primitives Blattobjekt
- kann keine Kinder enthalten
- implementiert das gewünschte Blattverhalten bei Aufruf von operation()

```
public class Leaf extends
    Component {
  public void operation() {
  /* gewünschtes Verhalten */
  }
}
```

Teilnehmer - Client

- baut mit der Schnittstelle der abstrakten Componentklasse den Baum auf
- nutzt Component als Typ für Leaf - und Composite-Objekte
- Leaf und Composite können gleich behandelt werden

```
Component c1 = new Composite();
Component c2 = new Leaf();
c1.add(c2);

c1.operation();
c2.operation();
```

Verantwortlichkeit von Component

- Composite Muster verstößt auf den ersten Blick gegen das objektorientierte Entwurfsprinzip der Hohen Kohäsion (Yoda)
- ein Kompromiss sollte eingegangen werden
- werden Blätter als Composite-Objekte, die keine Kinder enthalten, angesehen, wirds besser!
- Transparenz für den Client ist wichtiger als Sicherheit beim Methodenaufruf
- unsinnige Methodenaufrufe sind möglich!
- daher: Exceptions in Defaultimplementierung werfen

Implementierungstipps

- abstrakte Component Klasse als Schnittstelle arbeiten lassen
- Leaf und Composite Methoden in dieser Schnittstelle deklarieren
- Datenstruktur für Kinder in Composite Klasse nicht in Component

Anwendungen

- Java AWT/Swing Oberflächenkomponenten
- alle GUI Elemente erben von einer abstrakten Komponentenklasse namens Container
- Beziehung zu anderen Mustern
  - Visitor
  - Decorator
  - Command (Verschachtelung von Commandos)

Vor - und Nachteile

Vorteile

- Blätter- und Behälterklassen können gleich behandelt werden
- gemeinsame Operationen auf komplette Baumstruktur anwendbar
- neue Leaf und Composite Objekte lassen sich leicht hinzufügen

Nachteile

- Entwurf kann zu allgemein werden
- entstehende Komponenten eines Kompositums lassen sich nicht einschränken

Beispiel

Es folgt ein Beispiel...

Quellenangaben

Eric Freeman, Elisabeth Freemann u.a.
Entwurfsmuster von Kopf bis Fuß
O'Reilly 2006, ISBN 3-89721-421-0

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software
Addison Wesley 1996, ISBN 3-89319-950-0

Vielen Dank für die Aufmerksamkeit

- Haben Sie noch Fragen?
- Download der Präsentation:
- http://www.alex-rausch-online.de