Analyse und Modellierung von Informationssystemen

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2014/2015

Das Iterator-Muster

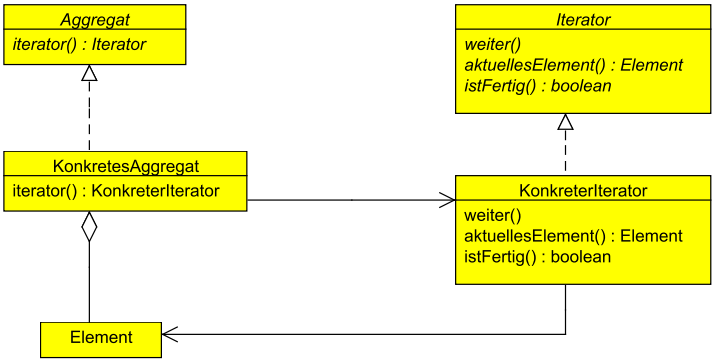
Iteratoren dienen dazu, über eine Menge von Elementen zu iterieren, wobei hier i. A. folgende Operationen existieren:

- Iteration ein Schritt vor
- Iteration ein Schritt zurück
- Zugriff auf das aktuelle Element der Iteration
- Abfrage, ob die Iteration am Ende angelangt ist.

Hierbei bleibt die interne Struktur der Elemente (sequenziell, baumartig) verborgen.

In Java existiert das Interface `Iterator<E>` in abgewandelter Form (mit Erweiterungen z. B. beim `ListIterator<E>`) zur Iteration über Collections (meist mit der erweiterten `for`-Schleife).

Schaubild



Beispiel

Der Iterator ist oft mit einer Elementmenge verknüpft, die durch ein Kompositum repräsentiert wird.

Im folgenden Beispiel wird basierend auf dem Kompositum aus der fünften Vorlesung, bei dem ein Dokument mit einer Baumstruktur von ggfs. dekorierten Textfragmenten dargestellt wurde.

Dieses Kompositum wird nun so erweitert, dass das Interface `Iterable<Node>` implementiert wird, wobei es sich um XML-artige Knoten handelt, die analog zum Parsen mit SAX entweder Text-Knoten oder Knoten sind, die den Start bzw. das Ende eines dekorierenden Elementes sind.

Vorkommende Knotenarten

Interface Node:

```
public interface Node {
}
```

Textknoten:

```
public class TextNode implements Node {
    private String text;

    public TextNode(String text) {
        this.text = text;
    }

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

Vorkommende Knotenarten (Forts.)

Start eines Elementes:

```
public class ElementNode implements Node {
    private String element;

    public ElementNode(String element) {
        this.element = element;
    }

    @Override
    public String toString() {
        return String.format("<%s>", element);
    }
}
```


Vorkommende Knotenarten (Forts.)

Ende eines Elementes:

```
public class EndElementNode implements Node {
    private String element;

    public EndElementNode(String element) {
        this.element = element;
    }

    @Override
    public String toString() {
        return String.format("</%s>", element);
    }
}
```


Erweiterung des Interface Component

Das Interface `Component` erbt nun vom Interface `Iterable<Node>`, daher muss nun jede implementierende Klasse über die Methode `iterator()` ein Objekt zurückliefert, das das Interface `Iterator<Node>` implementiert.

```
public interface Component extends Iterable<Node> {
    public void add(Component c);
    public boolean hasChildren();
    public Iterable<Component> getChildren();
}
```

Abstrakte Basis des Iterators

Iteratoren müssen in Java auch die Methode `remove()` zum Entfernen des aktuellen Elementes aus der Elementmenge implementieren. Da diese im Beispiel nicht benötigt wird und funktionslos bleiben soll, wird hier eine abstrakte Basisklasse geschaffen, die die Methode leer definiert (vergl. Template-Pattern).

```
public abstract class NodeIterator implements Iterator<Node> {
    @Override
    public void remove() {
    }
}
```


Iterator für einen reinen Text

```
public class Text extends AbstractComponentWithoutChildren {
    private String text;

    public Text(String text) {
        this.text = text;
    }

    @Override
    public Iterator<Node> iterator() {
        return new NodeIterator() {
            private Iterator<Node> state = new NodeIterator() {
                @Override
                public boolean hasNext() {
                    return true;
                }
```


Iterator für einen reinen Text (Forts.)

```
                @Override
                public Node next() {
                    state = new NodeIterator() {
                        @Override
                        public boolean hasNext() {
                            return false;
                        }

                        @Override
                        public Node next() {
                            throw new NoSuchElementException();
                        }
                    };
                    return new TextNode(text);
                }
            };
```

Iterator für einen reinen Text (Forts.)

```
            @Override
            public boolean hasNext() {
                return state.hasNext();
            }

            @Override
            public Node next() {
                return state.next();
            }
        };
    }
}
```


Iterator für dekorierten Text

```
public class StyleDecorator extends AbstractComponentWithoutChildren {
    private Component component;
    private Style style;

    public StyleDecorator(Component component, Style style) {
        this.component = component;
        this.style = style;
    }

    @Override
    public Iterator<Node> iterator() {
        return new NodeIterator() {
            private Iterator<Node> state = new NodeIterator() {
                @Override
                public boolean hasNext() {
                    return true;
                }
```


Iterator für dekorierten Text (Forts.)

```
                @Override
                public Node next() {
                    state = new NodeIterator() {
                        private Iterator<Node> iter = component.iterator()
                        @Override
                        public boolean hasNext() {
                            return true;
                        }

                        @Override
                        public Node next() {
                            if (iter.hasNext()) return iter.next();
                            state = new NodeIterator() {
                                @Override
                                public boolean hasNext() {
                                    return false;
                                }

                                @Override
                                public Node next() {
                                    throw new NoSuchElementException();
                                }
                            };
```

Iterator für dekorierten Text (Forts.)

```
                        return new EndElementNode(style.toString());
                    }
                };
                return new ElementNode(style.toString());
            }
        };

        @Override
        public boolean hasNext() {
            return state.hasNext();
        }

        @Override
        public Node next() {
            return state.next();
        }
    };
  }
}
```


Iterator für Compositions

```
public class Composition implements Component {
    class NodeIterator implements Iterator<Node> {
        private Iterator<Component> outer = components.iterator();
        private Iterator<Node> inner = null;

        public NodeIterator() {
            if (outer.hasNext()) {
                inner = outer.next().iterator();
            }
        }

        @Override
        public boolean hasNext() {
            if (inner==null) {
                return false;
            }
```


Iterator für Compositions (Forts.)

```
            while (true) {
                if (inner.hasNext()) {
                    return true;
                }
                if (outer.hasNext()) {
                    inner=outer.next().iterator();
                } else {
                    return false;
                }
            }
        }

        @Override
        public Node next() {
            if (hasNext()) {
                return inner.next();
            } else
                throw new NoSuchElementException();
        }
```

Iterator für Compositions (Forts.)

```
            @Override
            public void remove() {
            }
        }

        private List<Component> components = new ArrayList<Component>();

        @Override
        public void add(Component c) {
            components.add(c);
        }

        ...

        @Override
        public Iterator<Node> iterator() {
            return new NodeIterator();
        }

    }
```


Multi-Tier-Architekturen

Die *Schichtenarchitektur* wird häufig zur Strukturierung von Software-Architekturen verwendet. Hierbei werden die einzelnen Aspekte des Systems verschiednen Schichten (engl. *tier* oder *layer*) zugeordnet.

Dies dient der Entkopplung von Systemteilen und reduziert so die Komplexität des Problems. Weitere Vorteile sind bessere Wartbarkeit und die Möglichkeit, Teilkomponenten einfacher auszutauschen.

Grundprinzip:
Komponenten höherer Schichten dürfen nur die Funktionalität und Aspekte tieferer Schichten verwenden.

Die Weiterleitung von Daten zwischen den Schichten (ggfs. mit Transformation) kann sich nachteilig auf die Performance auswirken.

Zwei-Schichten-Architektur

Bei der Zwei-Schichten-Architektur (*two tier architecture*) greift die höhere Schicht auf die tiefere zu. Die höhere Schicht ist somit ein *Client*, der einen darunter liegenden *Server* steuert, der als Diensteanbieter funktioniert. Daher spricht man hier von einer *Client-Server-Architektur*.

Typischerweise sind die Clients leistungsstarke Applikationen, die z. B. auf einen Datenbank-Server zugreifen. Solche Clients werden als *Rich* bzw. *Fat Clients* bezeichnet.

Drei-Schichten-Architektur

In der Drei-Schichten-Architektur (*three tier architecture*) werden typischerweise folgende Schichten getrennt:

Präsentationsschicht (*client* oder *presentation tier*), die als *Front End* der Präsentation der Daten und der Entgegennahme von Benutzereingaben dient.

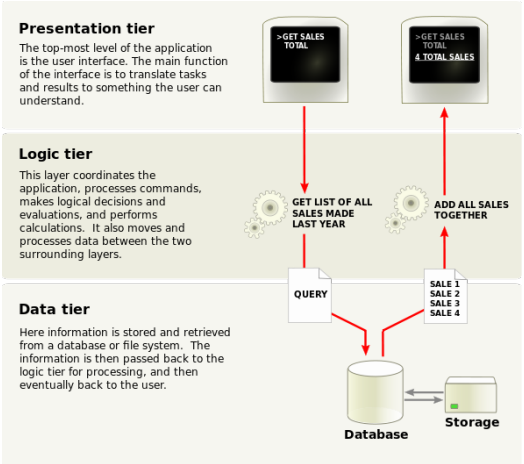
Logikschicht (*business*, *middle* oder *enterprise tier*), in der Eingaben verarbeitet und die Ergebnisse zu Präsentation aufbereitet und an die Präsentationsschicht geliefert werden.

Datenhaltungsschicht (*data-server* oder *back tier*) die i. A. die Datenbank enthält und die *Persistenz* der Daten sicherstellt.

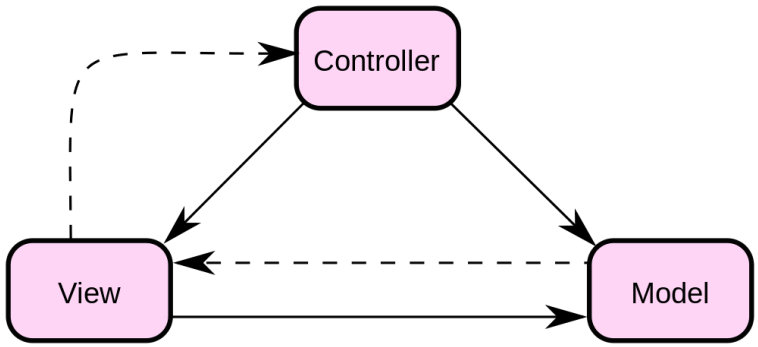
Drei-Schichten-Architektur (Forts.)

In verteilten Systemen dient die Drei-Schichten-Architektur einer besseren Skalierbarkeit des Systems: Als Datenhaltungsschicht wird eine (schnelle) Datenbank verwendet (ggfs. auch verteilt und redundant, wie z. B. bei Oracle RAC), die Geschäftslogik läuft in der Businessschicht auf Workstations verteilt, während die Präsentationsschicht durch *Thin Clients* (z. B. einfach Web-Browser) auf vielen Bedienterminals realisiert wird.

Innerhalb eines Softwaremoduls wird dieses Drei-Schicht-Model häufig durch das *Model-View-Controller*-Architekturmuster beschrieben.



Kommunikation innerhalb einer MVC-Architektur



Durchgezogene Pfeile beschreiben direkte Assoziation,
gestrichelte Pfeile indirekte Assoziation, z. B. über Beobachter.