5 Strategie-Muster

Um das Konzept des Strategie-Musters zu verdeutlichen wird ein Programm implementiert, das den Namen des aktuell verwendeten Betriebssystems ausgibt. Schreiben Sie eine Schnittstelle Strategie, die eine Methode `getOS()` deklariert.

```
public interface Strategie {
    public String getOS();
}
```

Schreiben Sie eine Klasse **`LinuxStrategie`**, welche die Schnittstelle **`Strategie`** implementiert.

```
public class LinuxStrategie implements Strategie {
    public String getOS() {
        return "Linux";
    }
}
```

Erstellen Sie analog eine Klasse **`WindowsStrategie`** und eine Klasse **`MacStrategie`**.

Schreiben Sie eine Klasse **`Betriebssystem`** als Kontext, die ein Attribut vom Typ **`Strategie`** besitzt. Implementieren Sie einen Konstruktor mit einem Übergabeparameter vom Typ Strategie. Überladen sie die Methode `toString()` der Klasse **`Betriebssystem`**, die die Methode `getOS()` der Strategie aufruft:

```
public String toString(){
    return strategie.getOS();
}
```

Testen Sie Ihr Programm, indem Sie eine Klasse **`BetriebbsystemTest`** schreiben. Um zu überprüfen welches Betriebssystem auf Ihrem Rechner läuft, verwenden Sie die Methode `System.getProperty("os.name")` der Java API:

```
boolean isWin =
    System.getProperty("os.name").startsWith("Windows");
boolean isMac = System.getProperty("os.name").startsWith("Mac");
boolean isLinux =
    System.getProperty("os.name").startsWith("Linux");
```

Erstellen Sie eine Strategie abhängig von dem Betriebssystem:

```
Strategie str;
if (isWin) {
    str =  new WindowsStrategie();
} else if (isMac) {
    str = new MacStrategie();
} else if (isLinux) {
    str = new LinuxStrategie();
} else {
    str = null;
    System.out.println("OS nicht feststellbar");
}
```

Überprüfen Sie das Ergebnis Ihres Programms indem Sie ein Objekt der Klasse **`Betriebssystem`** erstellen und die Methode `toString()` aufrufen:

```
Betriebssystem bs = new Betriebssystem(str);
System.out.println(bs);
```

Zeichnen Sie das zum Programm zugehörige Klassendiagramm. Orientieren Sie sich dabei an dem Beispiel aus der Vorlesung.

Aufgaben

1. Für ein Online-Shop sind verschiedene Zahlungsmöglichkeiten zu implementieren. Verwenden sie dazu das Strategie-Muster. Erstellen sie zunächst eine Schnittstelle **`Zahlungsstrategie`**.

2. Schreiben Sie zwei neue Klassen **`KreditkartenStrategie`** und **`PayPalStrategie`**, die die Schnittstelle **`Zahlungsstrategie`** implementieren. Orientieren sie sich dabei an dem UML-Klassendiagramm in Abb. 1. Die `zahle(preis:int)`-Methode soll je nach gewählter Strategie, den Preis, Namen und die Kreditkartennummer oder den Preis und die Email in der Konsole ausgeben.

3. Erstellen sie eine Klasse **`Warenkorb`**, die ein privates Attribut vom Typ Zahlungsstrategie definiert.

4. Testen sie die erstellten Klassen in der `main`-Methode einer neuen Klasse **`WarenkorbTest`**. Erstellen Sie dazu Warenkorb Objekte, die verschiedene Zahlungsstrategien als Übergabeparameter bekommen.

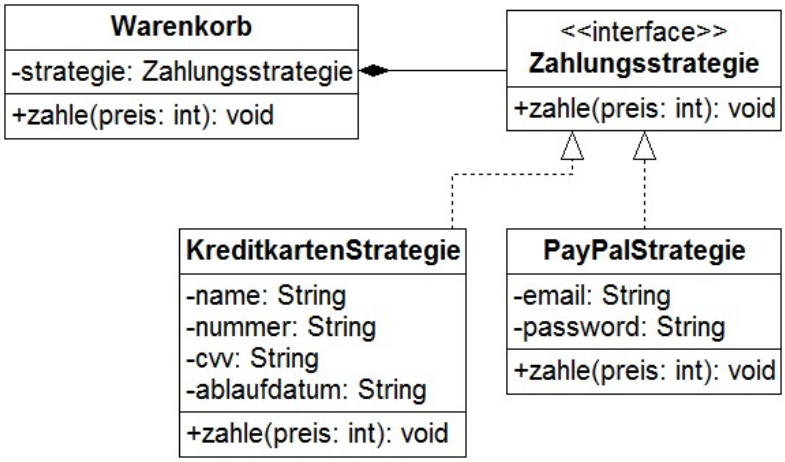

Abbildung 1: Klassendiagramm des Online-Shops

Lösung:

Warenkorb.java

```
package warenkorb;

public class Warenkorb {

    private Zahlungsstrategie strategie;

    public Warenkorb(Zahlungsstrategie strategie) {
        this.strategie = strategie;
    }

    public void zahle(int preis) {
        strategie.zahle(preis);
    }

}
```

Zahlungsstrategie.java

```
package warenkorb;

public interface Zahlungsstrategie {

    public void zahle(int preis);

}
```

KreditkartenStrategie.java

```java
package warenkorb;

public class KreditkartenStrategie implements Zahlungsstrategie {

    private String name;
    private String nummer;
    private String cvv;
    private String ablaufdatum;

    public KreditkartenStrategie(String name, String nummer, String
    ↪  cvv, String ablaufdatum) {
        this.name = name;
        this.nummer = nummer;
        this.cvv = cvv;
        this.ablaufdatum = ablaufdatum;
    }

    public void zahle(int preis) {
        System.out.println(preis + " bezahlt mit Kreditkarte " +
        ↪  nummer + " von " + name);
    }

}
```

PayPalStrategie.java

```java
package warenkorb;

public class PayPalStrategie implements Zahlungsstrategie {

    private String email;
    private String password;

    public PayPalStrategie(String email, String pwd) {
        this.email = email;
        this.password = pwd;
    }

    public void zahle(int preis) {
        System.out.println(preis + " bezahlt mit Paypal. Email
        ↪  Adresse: " + email);
    }

}
```

WarenkorbTest.java

```java
package warenkorb;

public class WarenkorbTest {

    public static void main(String[] args) {
        Warenkorb wk = new Warenkorb(new
        ↪  PayPalStrategie("myemail@example.com", "mypwd"));
        wk.zahle(500);

        Warenkorb wk2 = new Warenkorb(new KreditkartenStrategie("Max
        ↪  Mustermann", "1234567890123456", "786", "12/15"));
        wk2.zahle(1500);
    }

}
```

Iterator-Muster

Arbeiten Sie für die nächsten Aufgaben mit dem Binärbaum aus Übung 4 Aufgabe 3 weiter. Dazu können Sie einen neuen Branch ihres Git-Projektes erstellen oder mit ihren lokal gespeicherten Daten zu dieser Aufgabe weiterarbeiten.

Aufgaben

1. Erweitern Sie ihre Implementation passend zu dem UML-Klassendiagramm aus der Vorlesung und Abbildung 3, um das Iterator-Muster umzusetzen. Implementieren Sie einen Iterator für die Tiefensuche (DFS) mithilfe eines Stapels, und einen Iterator für die Breitensuche (BFS) mithilfe einer Warteschlange.

2. Die Standarditeration des Binärbaums soll die Tiefensuche sein. Erweitern Sie das UML-Diagramm um das `Iterable`-Interface und implementieren Sie es.

3. Erweitern Sie Ihren Iterator durch die Funktionen `peek()` und `previous()`. `peek()` soll das aktuelle Element des Iterators zurückgeben ohne die Iteration fortzuführen. `previous()` soll den Iterator "zurückspulen" und das vorherige Element zurückgeben.

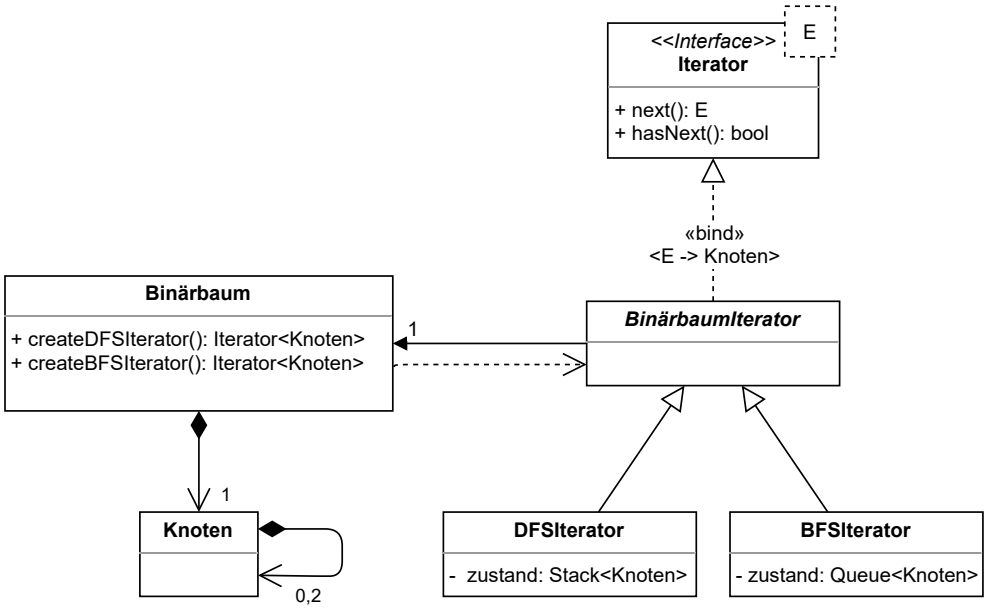

Abbildung 2: UML-Klassendiagramm für Binärbaum-Iteratoren

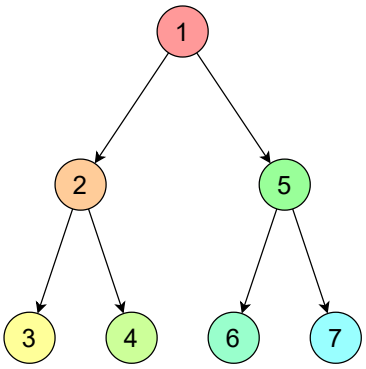

Abbildung 3: Tiefensuche

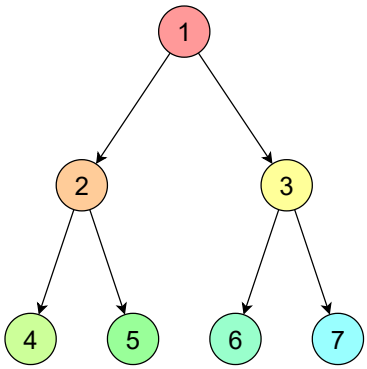

Abbildung 4: Breitensuche

Lösung:

BinaryTree.java

```
import java.util.Iterator;

public class Binaerbaum implements Iterable<Node>{
    private Node wurzelknoten;

    public void hinzufuegen(Person p) {
        if(wurzelknoten == null) {
            wurzelknoten = new Node(p);
        } else {
            wurzelknoten.hinzufuegenRekursiv(p);
        }
    }

    public Person finden(int id) {
        if(wurzelknoten == null)
            return null;
        else {
            return wurzelknoten.findenRekursiv(id);
        }
    }

    public Node getWurzelknoten() {
        return wurzelknoten;
    }

    public KnotenIterator createDFSIterator() {
        return new DFSIterator(this);
    }
    public KnotenIterator createBFSIterator() {
```

```
        return new BFSIterator(this);

    }

    @Override
    public Iterator<Node> iterator() {
        // DFS ist der Standarditerator
        return createDFSIterator();
    }
}
```

Knoten.java

```
public class Node {

    private Node links, rechts;
    private Person person;

    public Node(Person p) {
        this.person = p;
    }

    public Node getRechts() {
        return rechts;
    }

    public Node getLinks() {
        return links;
    }

    public void hinzufuegenRekursiv(Person einfuegePerson) {
        if(einfuegePerson.getId() <= person.getId()) {
            if(links != null) {
                links.hinzufuegenRekursiv(einfuegePerson);
            } else {
                links = new Node(einfuegePerson);
            }
        } else {
            if(rechts != null) {
                rechts.hinzufuegenRekursiv(einfuegePerson);
            } else {
                rechts = new Node(einfuegePerson);
            }
        }
    }
```

```
    @Override
    public String toString() {
        return person.getName();
    }

    public Person findenRekursiv(int id) {
        if(id == person.getId())
            return person;
        else if(id < person.getId()) {
            if(links != null)
                return links.findenRekursiv(id);
            else
                return null;
        } else {
            if(rechts != null)
                return rechts.findenRekursiv(id);
            else
                return null;
        }



    }
}
```

KnotenIterator.java

```
import java.util.Iterator;

public interface KnotenIterator extends Iterator<Node> {
    Node peek();
    Node previous();
}
```

BinaryTreeIterator.java

```
import java.util.Iterator;

public abstract class BinaryTreeIterator implements KnotenIterator {
    protected Binaerbaum tree;

    public BinaryTreeIterator(Binaerbaum tree) {
        this.tree = tree;
    }
}
```

DFSIterator.java

```java
import java.util.Stack;

public class DFSIterator extends BinaryTreeIterator {
    // Der Stack speichert die entdeckten Knoten, die noch besucht
    ↪ werden müssen
    private Stack<Node> knotenStack;

    // Der previousStack speichert bereits besuchte Knoten
    private Stack<Node> previousStack;

    public DFSIterator(Binaerbaum tree) {
        super(tree);
        knotenStack = new Stack<>();
        knotenStack.push(tree.getWurzelknoten());
        previousStack = new Stack<>();
    }

    @Override
    public boolean hasNext() {
        return !knotenStack.empty();
    }

    // Besucht den Knoten auf dem Stack, und falls kein Blatt,
    ↪ entdeckt neue Knoten
    @Override
    public Node next() {
        Node k = knotenStack.pop();
        if (k.getRechts() != null) {
            knotenStack.add(k.getRechts());
        }
        if (k.getLinks() != null){
            knotenStack.add(k.getLinks());
        }
        previousStack.add(k);
        return k;
    }

    @Override
    public Node peek() {
        return knotenStack.peek();
    }

    // Gibt den zuletzt besuchten Knoten zurück, und legt ihn wieder
    ↪ auf den Stack
    @Override
```

```
    public Node previous() {
        if(!previousStack.isEmpty()) {
            Node k = previousStack.pop();
            knotenStack.add(k);
            return k;
        }
        return null;
    }
}
```

BFSIterator.java

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class BFSIterator extends BinaryTreeIterator {
    // Die Queue speichert die entdeckten Knoten, die noch besucht
    ↪ werden müssen
    private Queue<Node> knotenQueue;

    // Der previousStack speichert bereits besuchte Knoten
    private Stack<Node> previousStack;

    public BFSIterator(Binaerbaum tree) {
        super(tree);
        knotenQueue = new LinkedList<>();
        knotenQueue.add(tree.getWurzelknoten());

        previousStack = new Stack<>();
    }

    @Override
    public boolean hasNext() {
        return !knotenQueue.isEmpty();
    }

    // Besucht den Knoten in der Queue, und falls kein Blatt, entdeckt
    ↪ neue Knoten
    @Override
    public Node next() {
        Node k = knotenQueue.poll();
        if (k.getLinks() != null) {
            knotenQueue.add(k.getLinks());
        }
        if (k.getRechts() != null){
```

```
            knotenQueue.add(k.getRechts());
        }
        previousStack.add(k);
        return k;
    }

    @Override
    public Node peek() {
        return knotenQueue.peek();
    }

    // Gibt den zuletzt besuchten Knoten zurück, und legt ihn wieder
    ↪  in die Queue
    @Override
    public Node previous() {
        if(!previousStack.isEmpty()) {
            Node k = previousStack.pop();
            knotenQueue.add(k);
            return k;
        }
        return null;
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        Person[] personen = {
                new Person("Alice", 102),
                new Person("Bob", 57),
                new Person("Chris", 451),
                new Person("Diane", 13),
                new Person("Esther", 78),
                new Person("Fritz", 255),
                new Person("Gianni", 900),
                new Person("Hannah", 66),
                new Person("Irene", 377)
        };

        Binaerbaum b = new Binaerbaum();
        for (Person p :
                personen) {
            b.hinzufuegen(p);
        }
        int id = 255;
```

```
        Person p = b.finden(id);
        if(p != null)
            System.out.println(p.getName());
        else
            System.out.println(id + " wurde nicht gefunden");

        KnotenIterator dfsIterator = b.createDFSIterator();

        System.out.println("Tiefensuche");
        while (dfsIterator.hasNext()) {
            Node k = dfsIterator.next();
            System.out.println(k);
        }

        System.out.println(dfsIterator.previous());
        System.out.println(dfsIterator.previous());
        System.out.println(dfsIterator.previous());

        System.out.println("Breitensuche");
        KnotenIterator bfsIterator = b.createBFSIterator();

        while (bfsIterator.hasNext()) {
            Node k = bfsIterator.next();
            System.out.println(k);
        }

        System.out.println(bfsIterator.previous());
        System.out.println(bfsIterator.previous());
        System.out.println(bfsIterator.previous());
    }
}
```