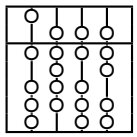
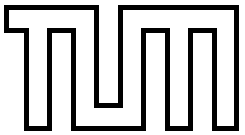


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK



Besuchermuster[1]

Das Besuchermuster ist ein spezielles Entwurfsmuster (*engl. design pattern*) Entwurfsmuster sind Lösungen für ständig wiederkehrende Problemstellungen der objektorientierten Programmierung.

Die hier betrachtete Problemstellung kann wie folgt dargestellt werden: In einer Objektstruktur sind Objekte vieler verschiedener Klassen enthalten. Wir wollen nun auf diesen Objekten Operationen durchführen, die von der konkreten Klasse der Objekte abhängen.

In der Praxis wird das Besuchermuster zur typabhängigen Abarbeitung von Datenstrukturen (Listen, Bäume, Ströme, ...) verwendet, die Objekte mit verschiedenen Klassen enthalten.

Im Folgenden finden Sie eine Schritt-Für-Schritt Anleitung für die Entwicklung eines Besuchers, erläutert an einem Beispiel aus dem Gartenbau.

1 Entwicklung

1.1 Behandlungsroutinen

Der Ausgangspunkt für die Entwicklung eines Besuchers ist immer eine bereits vorhandene Datenstruktur. Für unser Beispiel hier erstellen wir einzelne Blumenklassen, die wir in unserem Garten verwalten wollen:

```
1 class Rose implements Blume {
    public void stutzen() {
      /* stutzt die Rose */
    }
  }
6
  class Unkraut implements Blume {
    public void rupfen() {
      /* rupft das Unkraut */
    }
11 }
```

Ein Rosenzüchter möchte nun jede Blume verschieden behandeln, und stellt deswegen für jede Blumenklasse verschiedene Behandlungsroutinen bereit:

```
  class Rosenzuechter {
    public void visit(Rose r) {
      r.stutzen();
4   }
    public void visit(Unkraut u) {
      u.rupfen();
    }
  }
```

Zur Verwaltung unserer Blumen in unserem Garten können wir mehrere Blumen zu einem Beet (`Blume[] beet`), also einem Array zusammenfassen. Im Frühling bearbeitet ein Rosenzüchter sein Beet, indem er in einer Schleife alle Blumen dieses Beets behandelt. Das sollte eigentlich so aussehen:

```
  public void fruehling() {
```

1.2 Anpassungsschnittstelle

Die Lösung für dieses Problem besteht darin, dass wir über die Blume dem Rosenzüchter (*Besucher*) einen Tipp geben, wie er sie zu behandeln hat. Das sieht so aus:

```
interface Blume {
  public void accept(Rosenzuechter r);
}

class Rose implements Blume {
  public void stutzen() {
    /* stutzt die Rose */
  }
  public void accept(Rosenzuechter r){
    r.visit(this);
  }
}

class Unkraut implements Blume {
  public void rupfen() {
    /* rupft das Unkraut */
  }
  public void accept(Rosenzuechter r){
    r.visit(this);
  }
}
```

Damit das funktioniert muss die Frühlingsroutine so abgeändert werden, dass die Behandlung der Blumen nun von den Blumen selbst ausgelöst wird, anstatt vom Rosenzüchter. Folglich sieht die Frühlingsmethode wie folgt aus:

```
public void fruehling() {
  Blume[] beet = Garten.getBeet();
  Rosenzuechter manni = new Rosenzuechter();
  for (int i = 0; i < beet.length; i++){
    beet[i].accept(manni);
  }
}
```

Nun liefert der Übersetzer keinen Fehler mehr, da die jeweiligen Blumen ihren konkreten Typ innerhalb der `accept()`-Methoden kennen und ihn dem Rosenzüchter mitteilen können.

1.3 Verallgemeinerung

Manni ist nicht der einzige Gärtner in unserem Garten. Es sind noch weitere Gärtner denkbar, die die Blumen anders behandeln als ein Rosenzüchter. In der obigen Fassung sind Blumen allerdings nur darauf eingerichtet von Rosenzüchtern behandelt zu werden. Um also auch auf zukünftige Problemstellungen reagieren zu können (und damit eine flexible Datenstruktur bereitzustellen), sollten Blumen mit beliebigen Gärtnern zusammenarbeiten. Daher führt man eine allgemeine Schnittstelle `Gaertner` für alle Personen, die Blumen behandeln möchten, ein. Die nötigen Anpassungen dazu sieht man im abschließenden Beispiel.

1.4 Komplettes Beispiel

Zum besseren Verständnis folgt noch das komplette Beispiel aus der Botanik.

```
public interface Blume {
    public void accept(Gaertner g); // on the whole: c.visit(this);
}

public interface Gaertner {
    public void visit(Rose r);
    public void visit(Unkraut u);
}

public class Rose implements Blume {
    public void stutzen() {
        /* stutzt die Rose */
    }
    public void accept(Gaertner g) {
        g.visit(this);
    }
}

public class Unkraut implements Blume {
    public void rupfen() {
        /* rupft das Unkraut */
    }
    public void accept(Gaertner g) {
        g.visit(this);
    }
}

public class Rosenzuechter implements Gaertner {
    public void visit(Rose r) {
        r.stutzen();
    }
    public void visit(Unkraut u) {
        u.rupfen();
    }
}

public class Main {
    public static void main(String[] args) {
        Blume[] beet = Garten.getBeet();
        Gaertner manni = new Rosenzuechter();
        for (int i = 0; i < beet.length(); i++) {
            beet[i].accept(manni);
        }
    }
}
```

2 Anhang

UML

Das folgende UML-Diagramm veranschaulicht noch einmal allgemein das Zusammenwirken der einzelnen Komponenten, die an Besuchermustern beteiligt sind.