

Musterlösung Übung 8

Entwurfsmuster „Fabrik“ und „Einzelstück“

1. Entwerfen Sie nach dem **Factory**-Muster zwei Pizzerien („Pizza-Fabriken“), die die Bestellung einer Pizza unterschiedlich handhaben, z.B. `PizzaFabrikItalienisch` und `PizzaFabrikAmerikanisch`. Beide sollen Objekte vom Typ `Pizza` erzeugen (Sie können auch mehrere Varianten, z.B. `SalamiPizza`, `Calzone` usw. vorsehen), aber mit unterschiedlichen Saucen oder Teigvarianten (eine italienische Pizza hat üblicherweise einen dünneren Boden als eine amerikanische, dafür gibt es bei der amerikanischen Variante Käse im Rand).

Siehe die Klassen [uebung08-musterloesung/ItalienischePizzaFabrik.java](#) und [uebung08-musterloesung/AmerikanischePizzaFabrik.java](#), die jeweils unterschiedliche Parameter für die `zubereitung()` aufrufen:

_____ *Pizza.java:* _____

```
public abstract class Pizza {
    public abstract void zubereitung(String s);
}
```

_____ *PizzaFabrik.java:* _____

```
public abstract class PizzaFabrik {
    abstract Pizza bestelleSalamiPizza();
    abstract Pizza bestelleCalzone();
}
```

_____ *ItalienischePizzaFabrik.java:* _____

```
public class ItalienischePizzaFabrik extends PizzaFabrik {
    public Pizza bestelleCalzone() {
        Pizza calzone = new Calzone();
        calzone.zubereitung(", italienische Art");
        return calzone;
    }
}
```

```
public Pizza bestelleSalamiPizza() {
    Pizza salami = new SalamiPizza();
    salami.zubereitung(", italienische Art");
    return salami;
}
}
```

AmerikanischePizzaFabrik.java:

```
public class AmerikanischePizzaFabrik extends PizzaFabrik {
public Pizza bestelleCalzone() {
    Pizza calzone = new Calzone();
    calzone.zubereitung(", amerikanische Art");
    return calzone;
}

public Pizza bestelleSalamiPizza() {
    Pizza salami = new SalamiPizza();
    salami.zubereitung(", amerikanische Art");
    return salami;
}
}
```

Calzone.java:

```
public class Calzone extends Pizza {
public void zubereitung(String s) {
    System.out.println("Calzone wird zubereitet" + s);
}
}
```

SalamiPizza.java:

```
public class SalamiPizza extends Pizza {
public void zubereitung(String s) {
    System.out.println("Salami Pizza wird zubereitet" + s);
}
}
```

Main.java:

```
public class Main {
public static void main(String[] args) {
    PizzaFabrik italia = new ItalienischePizzaFabrik();
    PizzaFabrik california = new AmerikanischePizzaFabrik();

    Pizza calzone1 = italia.bestelleCalzone();
    Pizza calzone2 = california.bestelleCalzone();

    Pizza salami1 = italia.bestelleSalamiPizza();
    Pizza salami2 = california.bestelleSalamiPizza();
}
}
```

Prinzipiell sollte es unterschiedliche Implementationen von SalamiPizza, je nach Herstellungsart, geben. Die Musterlösung wurde jedoch sehr kurz gehalten.

2. Verwenden Sie das Entwurfsmuster **Singleton**, um einen „Ereignis-Protokollierer“ zu generieren, der
 - (a) bei seiner Instanzierung eine Datei zum Schreiben öffnet,
 - (b) Nachrichten von Objekten über eine Methode `log(String nachricht)` entgegennimmt und in die Datei schreibt.

In der Lösung wurde `FileWriter` verwendet. Damit jede neu eingetragene Zeile der Ausgabedatei sofort sichtbar ist, und die Datei auch jederzeit umbenannt oder kopiert werden kann, ohne dass der „Protokollierer“ durcheinanderkommt, wurde der Algorithmus dahingehend modifiziert, dass die Logdatei bei jedem Schreibvorgang im „Anfügen“-Modus geöffnet, und nach dem Schreibvorgang sofort wieder geschlossen wird.

_____ *Logger.java:* _____

```
import java.io.*;    // FileWriter()
import java.util.*; // Date()

public final class Logger {
    private static final Logger instance = new Logger();

    private Logger() {} /* Konstruktor ist privat. */

    public static Logger getInstance() { return instance; }

    public synchronized void log(String message) {
        try { // Datei zum Anhängen öffnen, schreiben, schließen.
            FileWriter fw = new FileWriter("logfile.txt", true);
            fw.write(new Date() + ": " + message + "\n");
            fw.close();
        } catch ( IOException e ) {
            System.out.println("Konnte nicht in Datei schreiben.");
        }
    }
}
```

Process.java:

```
import java.awt.*;

public class Process extends Thread {

    public void run() { // Prozess läuft los.
        Logger l = Logger.getInstance(); // (Einzige) Logger-Instanz holen

        // Damit man auch was sieht: Jeder Prozess bekommt ein Fenster
        Frame frame = new Frame(this.toString()); // Ein Fenster
        frame.setSize(200,100); // Größe ändern (breite, höhe)
        frame.move((int) (Math.random()*800), (int) (Math.random()*600));
        frame.setVisible(true);

        // In Log-Datei schreiben (synchronized: Immer nur einer)
        l.log("Neuer Prozess " + this.toString() + " gestartet.");

        // "ungefähr" eine Sekunde warten
        try { sleep((int) (10000.0 * Math.random())); }
            catch(InterruptedException ie) {} // Nichts tun, falls Unterbrechung

        frame.dispose(); // Fenster (für immer) schließen
        l.log("Prozess " + this.toString() + " beendet.");
    }

    public Process() {
        // start() aus Basisklasse Thread aufrufen, startet auch run()
        start();
    }
}
```

Main.java:

```
public class Main {
    public static void main(String[] args) {
        Logger l = Logger.getInstance();

        System.out.println("main() gestartet.");
        l.log("main() gestartet.");

        for(int i=0; i<100; i++) new Process();

        System.out.println("main() endet.");
        l.log("main() endet.");
    }
}
```

Warum ist für solche Aufgaben der Einsatz von **Singletons** sinnvoll?

Der Einsatz eines Singletons ist hier sinnvoll, um Chaos beim Schreiben in die Log-Datei zu vermeiden. Beispiel-Szenario:

- (a) Prozess Nr. 1 öffnet eine Logdatei.
- (b) Prozess Nr. 2 öffnet die gleiche Logdatei.
- (c) Prozess Nr. 1 schreibt in die Logdatei.
- (d) Prozess Nr. 2 schreibt in die Logdatei und schließt diese.
- (e) Prozess Nr. 1 schließt die Logdatei.

Frage: Welche Daten stehen nur in der Logdatei? Nur die von Prozess 1? Nur die von 2? Beide? Keine? → Antwort ist implementationsabhängig und nicht in jedem Fall deterministisch.

Generell ist der Einsatz von Singletons sinnvoll, wenn ein „Koordinationsobjekt“ benötigt wird, das gleichzeitige Zugriffe in einer Art Warteschlange sammeln und Zugriffe regeln kann („Atomizität“ von Operationen).