Hochschule für angewandte Wissenschaften
Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

Schwerpunktseminar

Design Pattern

Erarbeitet im Seminar der Vertiefungsrichtung Technische Informatik des Studiengangs Informatik

Tobias Hahn

Eingereicht: Oktober 2012

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Factory

Unter den Design Patterns gibt es zwei ähnlich lautende Patterns. Zum einen die Abstract Factory, zum anderen die Factory Method. Beide sind ähnlich aufgebaut und doch haben sie ein wenig andere Einsatzgebiete.

1.1 Abstract Factory

Es wird eine Schnittstelle bereitgestellt, um Familien verbundener oder abhängiger Objekte zu erstellen, ohne die konkreten Klassen zu spezifizieren.[KE11]

1.1.1 Motivation

Stellen sie sich eine Klassenbibliothek für Benutzungsschnitstellen vor, die mehrere Look-and-Feel-Standards wie Motif oder den Presentation-Manager unterstützt. Unterschiedliche Look-and-Feel-Standards definieren unterschiedliches Aussehen und Verhalten von Widgets, den Interaktionselementen einer Benutzerschnittstelle, wie Scrollbars, Fenster und Knöpfen. Um zwischen verschiedenen Look-and-Feel-Standards portierbar zu sein, sollte sich eine Anwendung nicht auf einen Widget spezifischen Standard festlegen. Die Erzeugung von Look-and-Feel spezifischen Widgetklassen über die ganze Anwendung zu verteilen macht es schwer, das Look-and-Feel später zu ändern.

Man kann das Problem durch Einführung einer abstrakten WidgetFabrik lösen, die eine Schnittstelle zum Erzeugen jeder grundlegenden Art von Widget deklariert, siehe Abbildung 1.1. Weiterhin gibt es eine abstrakte Klasse für jede Widgetart sowie konkrete Unterklassen, welche die Widgets für den jeweiligen Look-and-Feel-Standard implementieren. Die Schnittstelle der WidgetFabrik besitzt für jede abstrakte Widgetklasse eine Operation, die ein neues Widget zurückgibt. Klienten rufen diese Operationen auf, um Exemplare von Widgets zu erzeugen, ohne dabei die konkrete Klassen zu kennen, die sie benutzen. Somit bleiben sie unabhängig vom aktuellen Look-and-Feel.

Für jeden Lock-and-Feel-Standard gibt es eine konkrete Unterklasse von WidgetFabrik. Jede Unterklasse implementiert die Operationen zum Erzeugen des passenden Widgets für den Look-and-Feel. Die ErzeugeScrolbar-Operation der Motif-WidgetFabrik zum Beispiel erzeugt eine Scrolbar für Motif und gibt sie zurück, währen die entsprechende Operation der PMWidgetFabrik eine Scrollbar für den Presentation-Manager zurückliefert. Klienten erzeugen die Widgets ausschließlich über die Schnittstelle der WidgetFabrik und kennen die Klassen nicht, welche das Widget für ein bestimmtes Look-and-Feel implementieren. Mit anderen Worten, Klienten stützen sich immer nur auf eine durch eine abstrakte Klasse definierte Schnittstelle, nicht aber auf eine konkrete Klasse. Eine Widgetfabrik sichert zudem

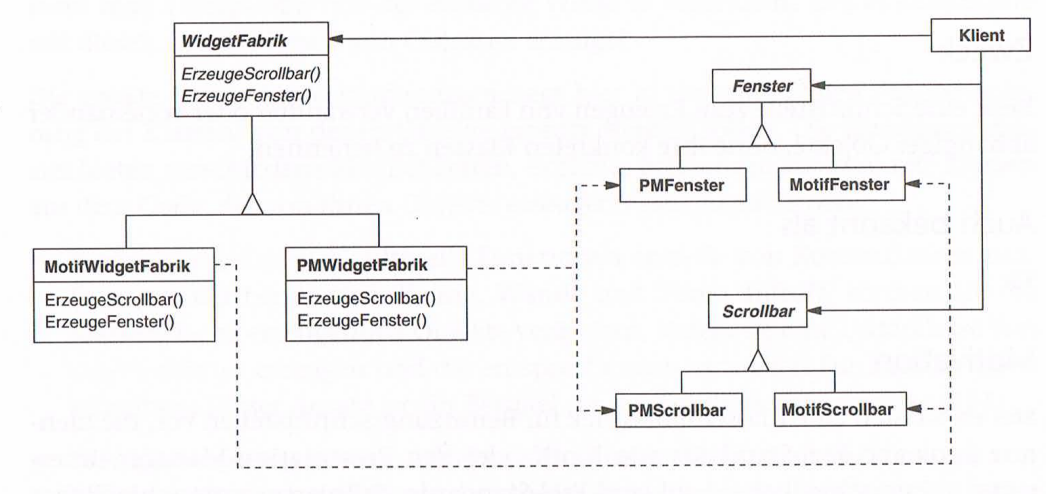

Abb. 1.1: Abstrakte Fabrik Beispiel[GHJV11]

Abhängigkeiten zwischen konkreten Widgetklassen ab. Eine Motif-Srollbar sollte nur mit einem Motif-Knopf und einem Motif-Texteditor zusammen verwendet werden. Diese Konsitenzbedingungen wird automatisch als Konsequenz des Einsatzes der MotifWidgetFabrik sichergestellt.[GHJV11]

1.1.2 Beschreibung / Lösung

Das Abstrakte Fabrik Muster kann verwendet werden, wenn ein System unabhängig davon sein soll, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden. Es ist auch anzuwenden wenn, ein System mit einer von mehreren Produktfamilien konfiguriert werden soll oder, wenn eine Familie von verwanden Produktobjekten entworfen wurde, zusammen verwendet zu werden, und Sie diese Konsistenzbeziehung sicherstellen müssen.
Die Abbildung 1.2 zeigt die Struktur des abstrakte Fabrik Musters.

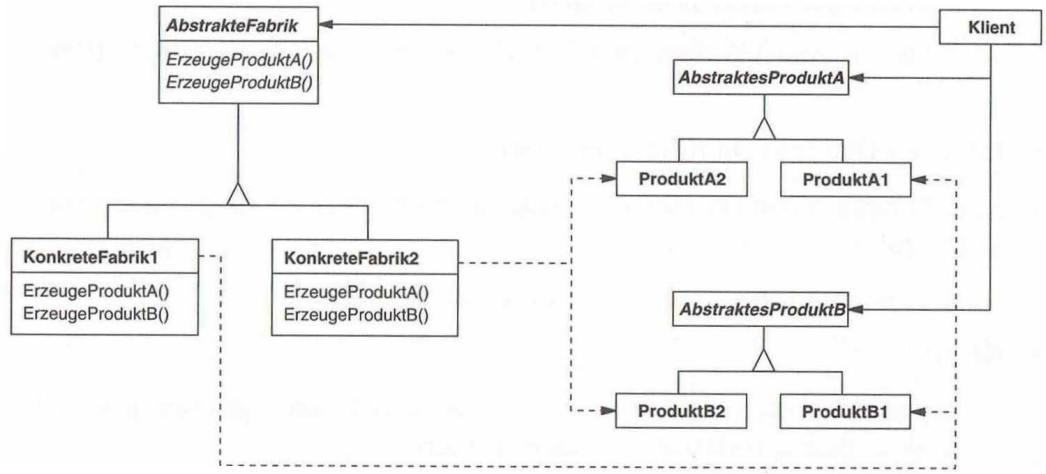

Abb. 1.2: Abstrakte Fabrik [GHJV11]

AbstrakteFabrik deklariert eine abstrakte Schnittstelle für Operationen, die konkrete Produktobjekte erzeugen.

KonkreteFabrik implementiert die Operationen zur Erzeugung konkreter Produktobjekte

AbstraktesProdukt deklariert eine Schnittstelle für einen bestimmten Typ von Produktobjekten.

KonkretesProdukt definiert ein von der entsprechenden konkreten Fabrik zu erzeugendes Produktobjekt und implementiert die AbstraktesProdukt-Schnittstelle.

Klient verwendet nur die Schnittstellen, welche von den AbstrakteFabrik und AbstaktesProdukt-Klassen deklariert werden.
[GHJV11]

Normalerweise wird ein einzelnes Exemplar der KonkretenFabrik-Klasse zur Laufzeit erzeugt. Diese konkrete Fabrik erzeugt Produktobjekte, welche spezifische Implementierungen haben. Um verschiedene Produktobjekte zu erzeugen, sollten Klienten unterschiedlich konkrete Fabriken haben. Eine AbstrakteFabrik verlagert die Erzeugung von Produktobjekten auf ihre KonkreteFabrik-Unterklasen.

1.1.3 Implementierung

Die folgende Implementierung soll die Funktionsweise der abstrakten Fabrik verdeutlichen. Hier wird die Verwendung einer abstrakten Fabrik im Zusammenhang mit der Erzeugung von verschiedenen Buttons gezeigt.

```
1  // Abstraktes Produkt
2  abstract class Button {
3      public abstract void paint();
4  }
5
6  //Abstrakte Fabrik
7  abstract class GUIFactory {
8      public static GUIFactory getFactory() {
9          int sys = readFromConfigFile("OS_TYPE");
10         if (sys == 0){
11                 return new WinFactory();
12         }
13         else{
14                 return new OSXFactory();
15         }
16     }
17
18     public abstract Button createButton();
19 }
20
21 //Konkrete Fabrik
22 class OSXFactory extends GUIFactory {
23     public Button createButton() {
24         return new OSXButton();
25     }
26 }
27
```

```
28 //Konkrete Fabrik
29 class WinFactory extends GUIFactory {
30     public Button createButton() {
31         return new WinButton();
32     }
33 }
34
35 //Konkretes Produkt
36 class OSXButton extends Button {
37     public void paint() {
38         System.out.println("I'm an OSXButton: ");
39     }
40 }
41
42 //Konkretes Produkt
43 class WinButton extends Button {
44     public void paint() {
45         System.out.println("I'm a WinButton: ");
46     }
47 }
48
49 //Klient
50 public class Application {
51     /**
52     * Es wird über die Abstrakte Fabrik eine konkrete
53     * Fabrik abgefragt und bei dieser über ein Abstraktes
54     * Produkt das konkrete Produkt, hier der Button.
55     */
56     public static void main(String[] args) {
57         GUIFactory factory = GUIFactory.getFactory();
58         Button button = factory.createButton();
59         button.paint();
60     }
61 }
```

Listing 1.1: Beispiel einer Abstrakten Fabrik Implementierung

1.1.4 Bewertung

Vorteile:

- Konkrete Klassen werden isoliert, somit erscheinen sie nicht im Klientcode
- Austausch einer Produktfamilie ist durch Austausch einer Fabrik vereinfacht
- Stellt Konsistenz unter Produkten sicher

Nachteile:

- Erzeugung neuer Produktarten ist schwierig, da alle Fabriken geändert werden müssen.

1.2 Factory Method

Es wird eine Schnittstelle für die Erzeugung von Objekten definiert. Die Entscheidung, welche konkrete Klasse zu instanziieren, zu konfigurieren und schließlich zurückzugeben ist, wird konkreten (Unter-) Klassen überlassen, die diese Schnittstelle implementieren.[KE11]

1.2.1 Motivation

Frameworks verwenden abstrakte Klassen, um die Beziehung zwischen Objekten zu definieren und zu verwalten. Ein Framework ist oft auch für die Erzeugung dieser Objekte zuständig.

Stellen sie sich ein Framework für Anwendungen vor, die dem Benutzer mehrere Dokumente auf einmal präsentieren können. Zwei zentrale Abstraktionen dieses Frameworks sind die Klassen Anwendung und Dokument. Beide Klassen sind abstrakt, und Klienten müssen Unterklassen von ihnen bilden um ihre Anwendungsspeziefische Implementierung einzubringen. Um beispielsweise eine Zeichenanwendung zu erstellen, definieren wir die Klasse ZeichenAnwendung und ZeichenDokument. Die Klasse Anwendung ist für die Verwaltung von Dokumenten zuständig und erzeugt sie auf Verlangen - beispielsweise wenn der Benutzer Öffnen oder Neu in einem Menü auswählt.

Da die jeweilige Dokumentenunterklasse, von der Objekte zu erzeugen sind, anwendungsspezifisch ist, kann die Anwendung diese Unterklasse nicht vorhersagen - sie weiß lediglich, wann ein neues Dokument erzeugt werden soll, nicht aber welche Art von Dokument zu erzeugen ist. Dies führt zu einem Dilemma: Das Framework muss Objekte erzeugen, kennt aber nur ihre abstrakten Oberklassen, von denen es keine Objekte erzeugen kann.

Das Fabrikmethodenmuster bietet die Lösung. Es kapselt das Wissen um die zu erzeugende Dokument-Unterklasse und lagert es aus dem Framework aus. Siehe Abbildung 1.3 Anwendungs-Unterklassen überschreiben eine abstrakte ErzeugeDokument-Operation und

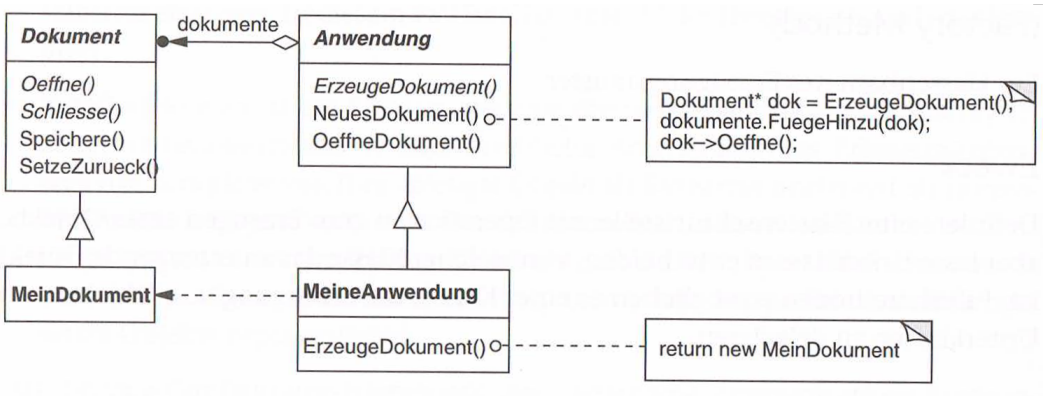

Abb. 1.3: Fabrikmethode Beispiel[GHJV11]

Anwendung, sodass die ein Exemplar der passenden Dokument-Unterlasse zurückgibt. Sobald einmal ein Objekt einer Unterklasse von der Anwendung erzeugt ist, kann sie anwendungsspeziefische Dokumente erzeugen, ohne deren exakte Klasse zu kennen. Wir nennen ErzeugeDokument eine Fabrikmethode, weil sie für die „Herstellung“ eines Objektes zuständig ist.[GHJV11]

1.2.2 Beschreibung / Lösung

Das Fabrikmethodenmuster findet Anwendung, wenn eine Klasse die Klassen von Objekten, die sie erzeugen soll im voraus nicht kennt.
Die Abbildung 1.4 zeigt die Musterstruktur.

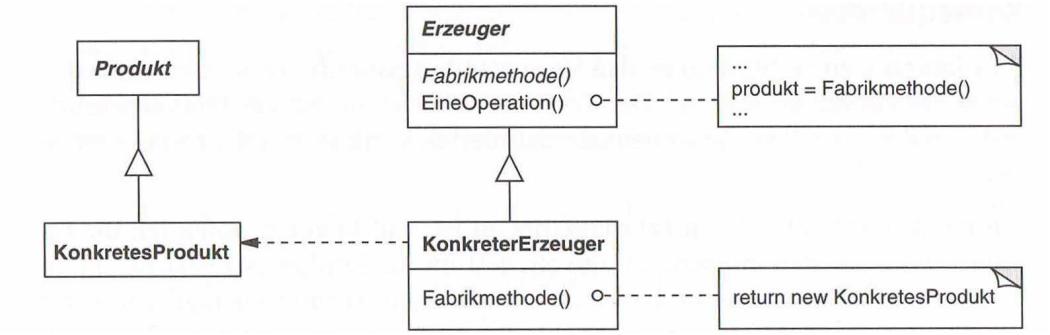

Abb. 1.4: Fabrikmethode [GHJV11]

Produkt definiert die Klasse des von der Fabrikmethode erzeugten Objekts.

KonkretesProdukt implementiert die Produktschnittstelle

Erzeuger deklariert die Fabrikmethode, die ein Objekt des Typs Produkt zurückgibt. Der Erzeuger kann möglicherweise eine Defaultimplementierung der Fabrikmethode definieren, die ein vordefiniertes KonkteteProduktObjekt erzeugt. Er kann die Fabrikmethode aufrufen, um ein Produktobjekt zu erzeugen.

KonkreterErzeuger überschreibt die Fabrikmethode, sodass sie ein Exemplar von KonkretesProdukt zurückgibt.
[GHJV11]

Der Erzeuger verlässt sich darauf, dass Unterklassen die Fabrikmethode definieren, so dass sie ein Exemplar der passenden konkreten Produktklasse zurückgeben.

1.2.3 Implementierung

Die folgende Implementierung soll die Funktionsweise der Fabrikmethode verdeutlichen. Hier wird die Verwendung einer Fabrikmethode im Zusammenhang mit der Erzeugung von verschiedenen grafischen Objekten gezeigt.

```
1 // die Figur-Schnittstelle
2 abstract public class Figur {
3         abstract public double umfang();
4         abstract public double flaeche();
5 }
6
7 // die nicht public Klasse fuer Rechtecke
8 class Rechteck extends Figur {
```

```
 9
10          private double breite, hoehe;
11
12          Rechteck(double breite, double hoehe) {
13                  this.breite = breite;
14                  this.hoehe  = hoehe;
15          }
16
17          public double umfang() {
18                  return 2 * (breite + hoehe);
19          }
20
21          public double flaeche() {
22                  return breite * hoehe;
23          }
24  }
25
26  // die nicht public Klasse fuer Kreise
27  class Kreis extends Figur {
28
29          private double radius;
30
31          Kreis(double radius) {
32                  this.radius = radius;
33          }
34
35          public double umfang() {
36                  return 2 * Math.PI * radius;
37          }
38
39          public double flaeche() {
40                  return Math.PI * radius * radius;
41          }
42  }
43
44  // die default Figur-Fabrik
45  public class FigurErzeuger {
46
47          public Figur erzeugeRechteck(double breite, double hoehe) {
48                  return new Rechteck(breite,hoehe);
49          }
50
51          public Figur erzeugeKreis(double radius) {
52          return new Kreis(radius);
53          }
54  }
55
56  // die Erweiterung fuer
57  // die Sonderbehandlung von Quadraten
58  // Quadrat erbt nicht von Rechteck !!!
59  class Quadrat extends Figur {
60
61          private double breite;
62
63          Quadrat(double breite) {
64                  this.breite = breite;
65          }
66
67          public double umfang() {
68                  return 4 * breite;
69          }
70
71          public double flaeche() {
72                  return breite * breite;
73          }
74  }
75  // die erweiterte Figur-Fabrik
76  public class ErweiterterFigurErzeuger extends FigurErzeuger {
77
```

```
78          public Figur erzeugeRechteck(double breite, double hoehe) {
79
80          // die alten Produkte werden mit der alten Fabrik erzeugt
81                  return (breite == hoehe) ?
82                          (Figur)new Quadrat(breite) :
83                          super.erzeugeRechteck(breite,hoehe); }
84  }
85
86  public class Klient{
87
88          static FigurErzeuger f;
89
90          public static void main(String [] argv) {
91                  if (argv[0] == "xxx") {
92                          f = new ErweiterterFigurErzeuger();
93                  } else {
94                          f = new FigurErzeuger();
95                  }
96
97                  Figur q = f.erzeugeRechteck(1.0,1.0);
98          }
99  }
```

Listing 1.2: Beispiel einer Factorymethode Implementierung

1.2.4 Bewertung

Vorteile:

- Parallele Hierarchien strukturierbar
- Leichter Austausch einer Fabrik, die leicht andere Produkte erzeugen kann
- Entkoppelt Objekterzeugung von der Logik, daher einfache Erweiterung

Nachteile:

- Neue Fabriken müssen explizit programmiert werden
- Der Aufwand um einfache Elemente zu erzeugen ist hoch
- Durch die vielen Subklassen steigt die Komplexität des Programmes

2 Singleton

Das Singleton Pattern auch „Einzelstück“ genannt gehört zu Klasse der Erzeugungsmuster. Es stellt sicher, dass nur genau eine Klasse einer Instanz erzeugt wird.[KE11]

2.1 Motivation

Bei manchen Klassen ist es wichtig, dass es genau ein Exemplar gibt. Obwohl es in einem System viele Drucker geben kann, sollte es nur einen Druckerpool geben. Es sollte nur ein Dateisystem und nur eine Fensterverwaltung geben. Ein digital Filter besitzt einen A/D-Konvertierer. Ein Buchhaltungssystem dient während es arbeitet genau einer Firma.

Wie stellen wir sicher, dass eine Klasse über genau ein Exemplar verfügt und dass einfach auf dieses Exemplar zugegriffen werden kann? Eine globale Variable ermöglicht den Zugriff auf ein Objekt, verhindert aber nicht das erzeugen mehrerer Exemplare.

Es ist besser, die Klasse selbst für die Verwaltung ihres einzigen Exemplars zuständig zu machen. Die Klasse kann durch Abfragen von Befehlen zur Erzeugung neuer Objekte sicherstellen, dass kein weiteres Exemplar erzeugt wird, und sie kann die Zugriffsmöglichkeit auf das Exemplar anbieten. Dies ist die Essenz des Singletonmusters.[GHJV11]

2.2 Beschreibung / Lösung

Das Singleton Muster findet Anwendung, wenn es genau ein Exemplar einer Klasse geben darf und es für einen Klienten an einem wohl definiertem Punkt zugreifbar sein soll.

Die Abbildung 2.1 zeigt die Struktur des Singletonmusters.

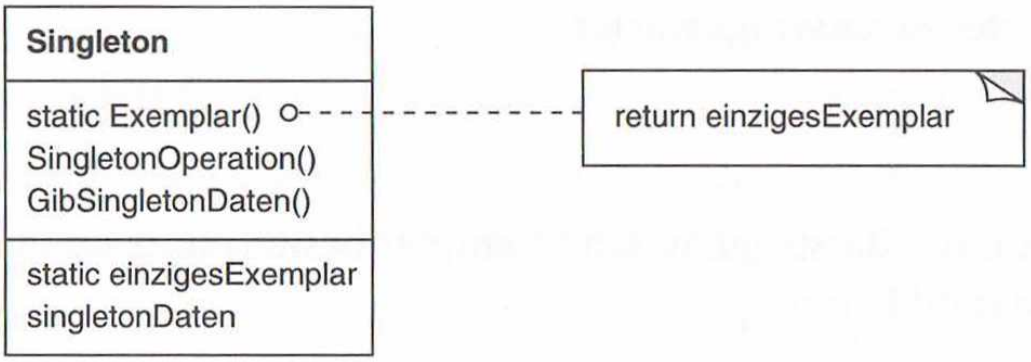

Abb. 2.1: Singleton [GHJV11]

Singleton definiert eine Exemplaroperation, die es dem Klienten ermöglicht, auf sein einziges Exemplar zuzugreifen. Es ist potenziell für die Erzeugung seines einzigen Exemplares zuständig.
[GHJV11]

Ein Klient kann ausschließlich auf ein Singletonexemplar durch die statische Exemplar-Operation der Singletonklasse zugreifen.

2.3 Implementierung

Die folgende Implementierung soll die Funktionsweise des Singletonmusters verdeutlichen. Hier wird die Verwendung eines Singleton im Zusammenhang mit einer Protikolldatei gezeigt.

```
1  public final class Log {
2
3          /**
4          * Privates Klassenattribut,
5          * wird beim erstmaligen Gebrauch (nicht beim Laden) der Klasse erzeugt
6          */
7          private static Log instance;
8
9          //Konstruktor ist privat, die Klasse darf nicht von außen instanziiert werden.
10         private Log() {}
11
12         /**
13         * Statische Methode "getInstance()" liefert die einzige Instanz
14         * der Klasse zurück.
15         * Ist synchronisiert und somit thread-sicher.
16         */
17         public synchronized static Log getInstance(){
18                 if (instance == null) {
19                         instance = new Log();
20                 }
21                 return instance;
22         }
23
24         public void openLogFile(){
25                 //Hier kann die Protokolldatei geöffnet werden.
26         }
27
28         public void closeLogFile(){
29                 //Hier kann die Protokolldatei geschlossen werden.
```

```
30          }
31
32          public void addLogEntry(String entry){
33                  //Hier kann die Protokolldatei um einen Eintrag erweitert werden.
34          }
35  }
36
37  public class Klient
38  {
39          /**
40          * Die Methode "run()"  verwendet das Singletonexemplar
41          *und führt Methoden dieser Klasse aus.
42          */
43          public void run(){
44                  Log.getInstance().openLogFile();
45                  Log.getInstance().addLogEntry("run wird ausgefuehrt");
46                  Log.getInstance().closeLogFile();
47          }
48  }
```

Listing 2.1: Beispiel einer Singleton Implementierung

2.4 Bewertung

Vorteile:

- Einfach zu implementieren
- Es kann eine Zugriffskontrolle realisiert werden

Nachteile:

- Kann zu Problemen bei multithreading führen, ohne die entsprechende Implementierung
- Es besteht die Gefahr einer übertriebenen Nutzung des Musters

Literaturverzeichnis

[GHJV11] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON JOHN VLISSIDES: *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*. Pearson Deutschland GmbH, München, 6. Auflage , 2011.

[KE11] KARL EILBRECHT, GERNOT STARKE: *Patterns kompakt - Entwurfsmuster für effektive Software-Entwicklung*. Spektrum Verlag, Heidelberg, 3. Auflage , 2011.