Hochschule für angewandte Wissenschaften
Fakultät Informatik und Wirtschaftsinformatik

Seminararbeit

Design Patterns

Decorator und Facade

Lorenz Hrobarsch und Umut Ikibas

19. November 2012

Inhaltsverzeichnis

1 Decorator

Im Folgenden wird das Dekorator Design Pattern beschrieben. Es handelt sich dabei um einen Vertreter der Struktur Patterns und dient dazu, Objekte flexibel um Funktionalitäten zu erweitern. Die sogenannte Gang of Four (GoF) fasst das Dekorator Pattern folgendermaßen zusammen:

> *"Dekorierer bieten eine flexible Alternative zur Unterklassenbildung, um die Funktionalität einer Klasse zu erweitern."* [EG10, S.199]

1.1 Ziel

Wie sich der Zusammenfassung der GoF entnehmen lässt, liegt das Ziel der Nutzung des Dekorator Pattern darin, Klassen um Funktionalitäten zu erweitern. Hierfür gibt es unterschiedliche Ansätze, wobei einer davon die Vererbung ist. Die Erweiterung von Klassen über Vererbung ist jedoch nicht sehr flexibel, da sie ausschließlich zur Kompilierzeit vorgenommen werden kann. [Gam94, S.177] Der Ansatz des Dekorator Patterns liegt darin, die zu erweiternde Klasse mit weiteren Klassen, die zusätzliche Funktionalitäten enthalten, zu "dekorieren", so dass die Funktionalitäten auch zur Laufzeit hinzugefügt oder entfernt werden können. [Gam94, S.177] Wie diese "Dekoration" einer Klasse konkret im Sourcecode aussieht, wird im Abschnitt 1.3 ausgeführt.

1.2 Beschreibung

Um die Funktionsweise des Dekorator Pattern verständlich zu erklären, wird nach der Erklärung der allgemeinen Funktionsweise eine Analogie aus dem Alltag herangezogen. Die Zusammenhänge werden anhand der Zusammenstellung einer Pizza verdeutlicht. Zuerst wird jedoch die theoretische Funktionsweise des Dekorator Patterns anhand des Klassendiagramms in Abbildung 1.1 erklärt. Diese bietet einen Überblick über die beteiligten Klassen und in welchem Verhältnis diese zueinander stehen. Das Interface "Komponente" muss von allen konkreten Komponenten und Dekoratoren (in Abbildung 1.1 "Dekorierer" genannt) implementiert werden. Die konkreten Komponenten stellen die Objekte dar, die mit den Dekoratoren erweitert werden können und implementieren die Methoden des Interfaces "Komponente". Der abstrakte "Dekorierer" aus Abbildung 1.1 dient als "Vorlage" für die konkreten Dekoratoren, die die zusätzlichen Funktionalitäten enthalten (in Abbildung 1.1 "zusatzVerhalten()" und "zusatzZustand"). Da der Dekorator dieselbe Schnittstelle implementiert, wie die konkreten Komponenten, ist er für Klassen, die von "außen" auf Funktionen der konkreten Komponente zugreifen wollen "transparent". Der Dekorator schließt die konkrete Komponente also ein, erlaubt es aber weiterhin ihre Funktionen nutzen zu können. [Gam94, S.175]

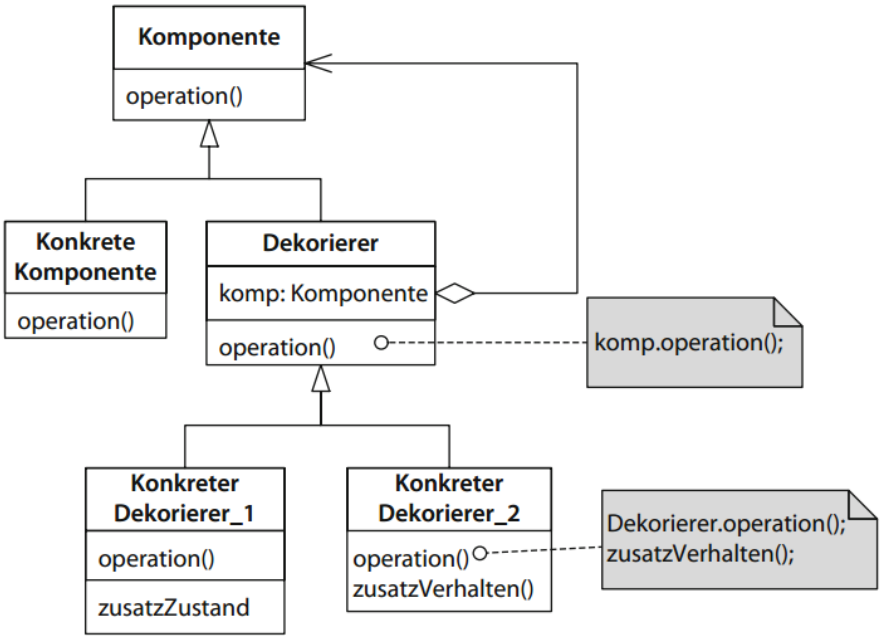

Abb. 1.1: Struktur des Dekorator Pattern (Klassendiagramm)[Eil10]

Abbildung 1.2 zeigt die eben beschriebene Funktionsweise übertragen, auf das bereits erwähnte Beispiel der Zusammenstellung einer Pizza. Die “Komponente“ aus Abbildung 1.1 wird durch “Pizza“ ersetzt. Es wird angenommen, dass sich eine Pizza aus zwei “Teilen“ zusammensetzt: dem Boden (“konkrete Komponente“) und dem Belag (“konkreter Dekorator“). Im Beispiel gibt es zwei Arten von Böden, “Italian Style“ und “American Style“. Die Böden können (wie die konkreten Komponenten mit Dekoratoren) mit Belägen “dekoriert“ werden. Der Übersichtlichkeit wegen wird der “Dekorierer“ aus Abbildung 1.1 hier “Decorator“ genannt, anstatt die Bezeichnung “Belag“ zu verwenden. Der “Decorator“ stellt also die “Vorlage“ für die Beläge der Pizza dar. Diese Vorlage wird durch mehrere “Beläge“ implementiert, stellvertretend dargestellt durch “BelagSalami“.

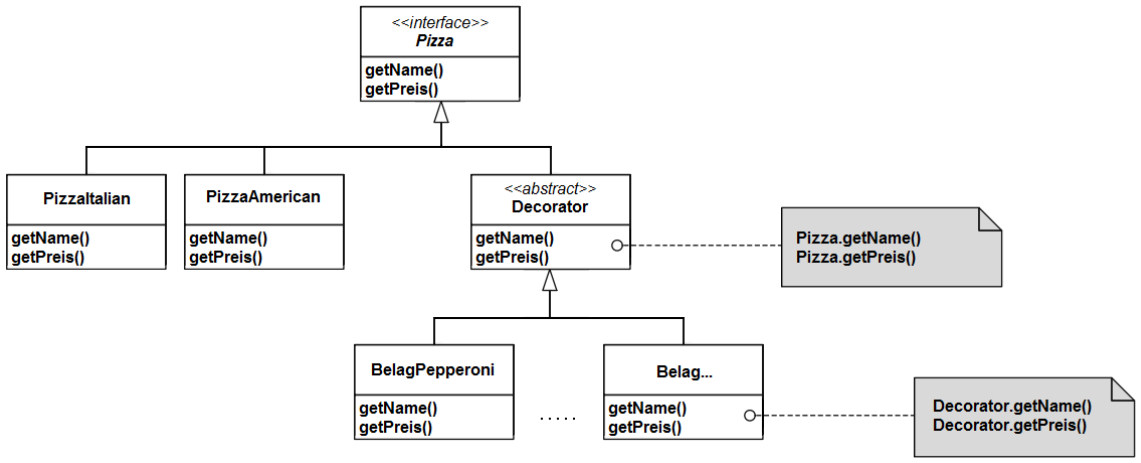

Abb. 1.2: Für das Pizza Beispiel angepasstes Klassendiagramm

1.3 Implementierung

Nachdem die grundlegende Funktionsweise des Decorator Patterns erklärt wurde, folgt die konkrete Implementierung anhand des Pizza Beispiels. Zuerst wird das Interface “Pizza“ wie in Listing 1.1 gezeigt implementiert . Dort werden die beiden Methoden getName() und getPreis() definiert.

```
1 public interface Pizza
2 {
3   public String getName();
4   public double getPreis();
5 }
```

Listing 1.1: Das Interface “Pizza“

Da die Implementierung der beiden Pizzaböden weitgehend gleich ist, zeigt Listing 1.2 stellvertretend nur die Implementierung der konkreten Komponente “PizzaAmerican“. Der einzige Unterschied zwischen beiden Klassen sind der zurückgegebene Name und der unterschiedliche Preis (“PizzaAmerican“ 4,50 Euro und “PizzaItalian“ 4,00 Euro). Beide implementieren die Methoden aus dem Interface “Pizza“.

```
1 public class PizzaAmerican implements Pizza{
2   public String getName()
3   {
4     return "Pizza (American): Boden mit Käse im Rand,
          Tomatensoße, Käse";
5   }
6
7   public double getPreis()
8   {
9     return 4.50;
10  }
11 }
```

Listing 1.2: Die Klasse “PizzaAmerican“

Um die Pizzaböden mit Belägen zu “dekorieren“, benötigt man zuerst die “Vorlage“ für die Beläge, also den abstrakten Dekorator. Dieser wird, wie in Listing 1.3 gezeigt, implementiert.

```
1 public abstract class Decorator implements Pizza
2 {
3   public abstract String getName();
4   public abstract double getPreis();
5 }
```

Listing 1.3: Der abstrakte Dekorator “Decorator“

Nachdem die “Vorlage“ für die konrketen Dekoratoren festgelegt wurde, folgt in Listing 1.4 die Implementierung eines konkreten Dekorators. Da der Einfachheit wegen allen Belägen der Preis von 0,50 Euro zugewiesen wurde, unterscheiden sich die Beläge ausschließlich bei

den zurückgegebenen Namen. Daher wird in Listing 1.4 stellvertretend nur der Sourcecode des Belags “BelagPepperoni“ betrachtet. Um in der Testandwenung eine Auswahl an Belägen für die Pizza-Objekte zu haben, stehen die Dekoratoren “Mozzarella”, “Champignons”, “Schinken”, “Pepperoni“, “Salami“, “Oliven”, “Pilze”, “Parmesan” und “Thunfisch” zur Verfügung.

```
1  public class BelagPepperoni extends Decorator
2  {
3    private Pizza pizza;
4
5    public BelagPepperoni (Pizza p)
6    {
7      this.pizza = p;
8    }
9
10   public String getName()
11   {
12     return this.pizza.getName() + ", Pepperoni";
13   }
14
15   public double getPreis()
16   {
17     return this.pizza.getPreis()+0.50;
18   }
19 }
```

Listing 1.4: Der konkrete Dekorator “BelagPepperoni“

Um die verschiedenen Dekoratoren und Kombinationen zu testen, wird die Testanwendung “Main“ genutzt. Wie man Listing 1.5 entnehmen kann, werden mehrere Objekte vom Typ “Pizza” erstellt, die mit verschiedenen Belägen dekoriert werden. Anschließend werden auf der Kommandozeile die zusammengestellten Pizzen und der jeweilige Gesamtpreis ausgegeben.

```
1  public class Main
2  {
3    public static void main(String[] args)
4    {
5      Pizza pizza = new PizzaItalian();
6      Pizza pizza2 = new PizzaAmerican();
7      Pizza pizza3 = new PizzaItalian();
8      Pizza pizza4 = new PizzaAmerican();
9
10     // Pizza mit Pepperoni
11     pizza = new BelagPepperoni(pizza);
12
13     // Pizza mit Oliven, Schinken, Salami und Champignons
14     pizza2 = new BelagOliven (pizza2);
```

```
15      pizza2 = new BelagSchinken (pizza2);
16      pizza2 = new BelagSalami (pizza2);
17      pizza2 = new BelagChampignons (pizza2);
18
19      // Pizza mit Mozzarella, Champignons,
20      // Schinken, Oliven, Pilze, Parmesan und Thunfisch
21      pizza3 = new BelagMozzarella (pizza3);
22      pizza3 = new BelagChampignons (pizza3);
23      pizza3 = new BelagSchinken (pizza3);
24      pizza3 = new BelagOliven (pizza3);
25      pizza3 = new BelagPilze (pizza3);
26      pizza3 = new BelagParmesan (pizza3);
27      pizza3 = new BelagThunfisch (pizza3);
28
29      // Pizza mit Salami und Pepperoni
30      // alternative Schreibweise
31      pizza4 = new BelagSalami (new BelagPepperoni (pizza4));
32
33      System.out.println(pizza.getName() + ": " + pizza.
            getPreis() + " Euro");
34      System.out.println(pizza2.getName() + ": " + pizza2.
            getPreis() + " Euro");
35      System.out.println(pizza3.getName() + ": " + pizza3.
            getPreis() + " Euro");
36      System.out.println(pizza4.getName() + ": " + pizza4.
            getPreis() + " Euro");
37
38    }
39  }
```

Listing 1.5: Die Klasse “Main“

Die Ausgabe der berechneten Preise und in der Anwendung zusammengestellten Pizza Objekte werden in Abbildung 1.3 dargestellt. Wie man sieht, wurden neben den aneinandergefügten Namens-Strings auch die Preise der verschiedenen Beläge zu den Pizzaböden aufaddiert.

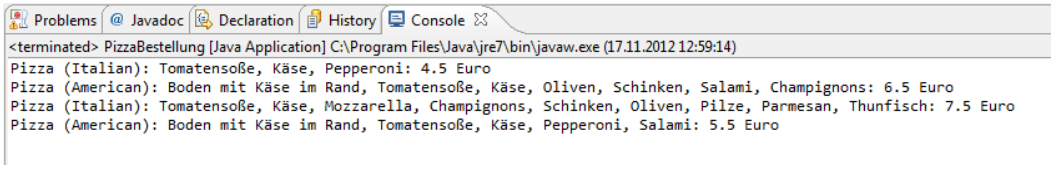

Abb. 1.3: Ausgabe der Testklasse Main auf der Konsole

1.4 Bewertung

Abschließend werden die Vor- und Nachteile der Nutzung des Decorator Patterns dargelegt.

1.4.1 Vorteile

- Flexible und funktionale Erweiterung von Klassen zur Kompilierzeit und zur Laufzeit. [Gam94, S.175]

  Pizza Beispiel: Es können jederzeit Beläge hinzugefügt oder entfernt werden.

- Gleichzeitige Nutzung mehrerer Dekoratoren (was bei einer Vererbung nicht möglich ist). [Gam94, S.178]

  Pizza Beispiel: Statt einer Klasse für jede Kombination von Belägen und Böden können die Böden beliebig mit Belägen “dekoriert“ werden.

- Bestehender Code muss zur Erweiterung nicht angepasst werden. [Hau09b]

  Pizza Beispiel: Neue Beläge ohne großen Aufwand möglich.

1.4.2 Nachteile

- Es kann sich eine große Zahl von Objekten ergeben, die sich sehr ähnlich sind, wodurch die Fehlerfindung, sowie die Einarbeitung erschwert wird. [Gam94, S.178]

2 Facade

Facade ist ein Struktur-Pattern, das dazu verwendet wird, den Zugriff auf komplexe Sub-Systeme durch Reduktion der zugänglichen Schnittstellen zu vereinfachen. Die GoF definiert das Muster wie folgt:

> *"Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use."* [Gam94, S.199]

2.1 Ziel

Oft bestehen Software-Systeme aus vielen einzelnen Klassen mit vielen Methoden, wobei die meisten ausschließlich für die interne Informationsverarbeitung benötigt werden. Sollen nun Klienten mit diesem System arbeiten, benötigen diese nur bestimmte Methoden und nutzen diese oft auch in einer bestimmten Reihenfolge. Hier kann nun die Fassade zum Einsatz kommen, um den Umgang mit einem solchen System vereinfachen. Nach außen bietet die Klasse der Fassade nur eine einfache Schnittstelle, welche die wichtigsten und von außen benötigten Funktionen zur Verfügung stellt. Alle anderen werden von der Fassade verborgen. So können zum Beispiel bei komplexeren Bibliotheken nur die relevanten Funktionen von außen zugänglich gemacht werden, um Anwender nicht zu verwirren. Ebenso können einzelne Programmabläufe zusammengefasst werden, was den Umgang vereinfacht und die Fehleranfälligkeit verringert (siehe Abschnitt 2.3). Auch zur Entkopplung kann dieses Entwurfsmuster genutzt werden: Werden abhängige Klassen (Klienten) nicht gegen ein komplettes System entwickelt, sondern nur gegen eine Schnittstelle (die Fassade), so ist das darunterliegende System ohne weiteres änder- bzw. austauschbar, ohne Änderungen an den Klienten vornehmen zu müssen. Dieses Prinzip lässt sich auch nutzen, um ein Programm in mehrere Schichten zu gliedern. Jede Schicht ist für bestimmte Aufgaben zuständig und ist für die anderen Schichten nur über die Fassade sichtbar. Das fördert wiederum die unabhänige Programmierung und mindert die Abhängigkeiten innerhalb des Programm-Codes.

2.2 Beschreibung

Abbildung 2.2 zeigt den Aufbau eines Systems, einmal ohne und einmal mit Fassade . Auf der linken Seite ist zu sehen, dass alle Klienten direkt auf die Subsystem-Klassen und deren Funktionen zugreifen können. Werden verschiedene Funktionen aufgerufen, muss jedem Klienten jede einzelne Klasse bekannt fsein, deren Funktion er nutzen möchte. Auf der rechten hingegen kommt eine Fassade zum Einsatz. Sie bietet alle Funktionen, die von Klienten benötigt werden, und delegiert alle Aufrufe an die entsprechenden Objekte im Subsystem. Die gesamte Funktionalität des Systems bleibt erhalten. Die Subsystemklassen wissen dabei von der Fassade nichts.

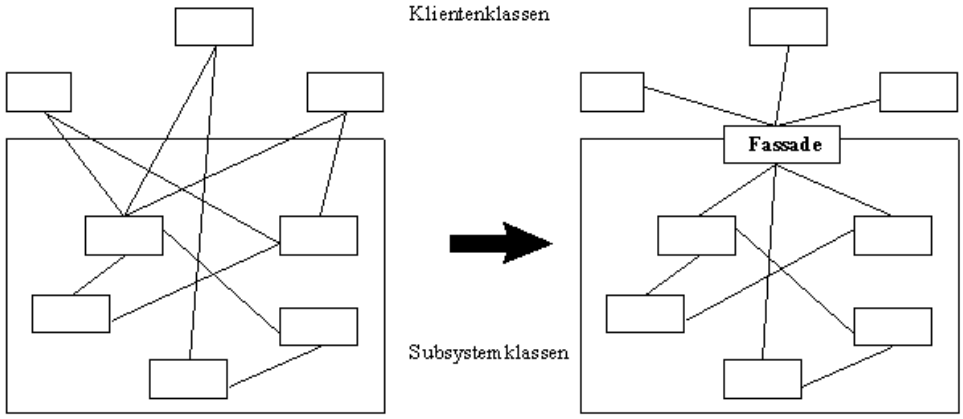

Abb. 2.1: System mit und ohne Fassade [EG10, S.185]

2.3 Implementierung

Das Entwurfsmuster soll anhand einer kleinen Firmenverwaltung erläutert werden. Diese Verwaltung bietet verschiedene Klassen zum Erstellen und Versenden von Texten welche von Clients (also den Nutzern der Verwaltung) verwendet werden können. Alle Klassen bieten spezielle Methoden , um zum Beispiel Texte zu schreiben oder diese auszudrucken:

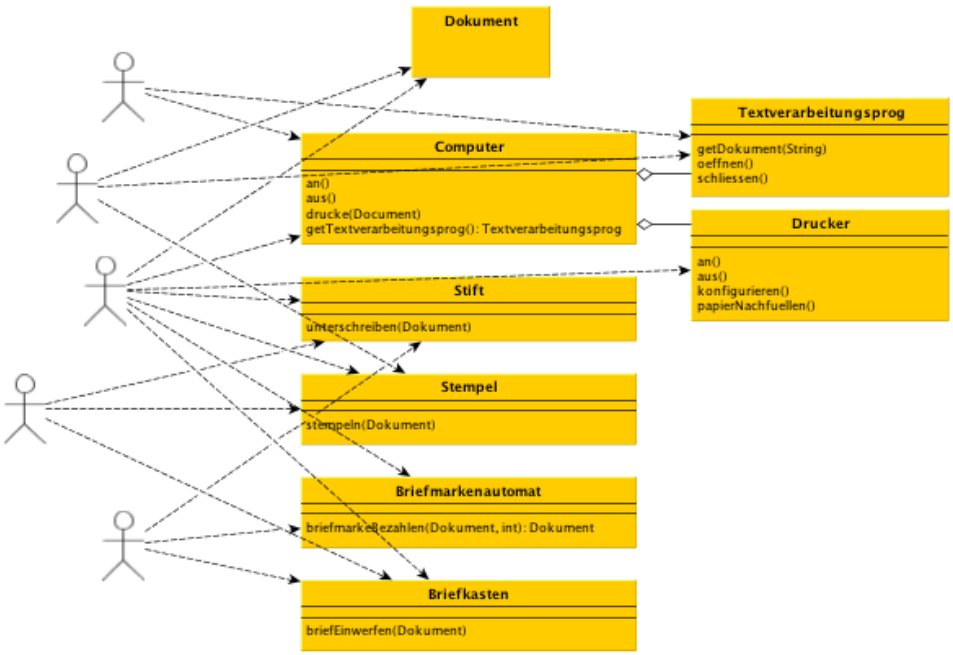

Abb. 2.2: Verwaltungs-System ohne Fassade

Jeder Client kann jede einzelne Funktion nach Belieben nutzen, allerdings will der Großteil der Benutzer diese Funktionen nutzen, um einen Text zu schreiben, zu drucken und anschließend zu verschicken. Dabei werden alle Klassen des Systems in immer wieder der gleichen Reihenfolge genutzt:

- Anschalten des Computers
- Starten des Textverarbeitungsprogramms
- Erzeugen des Dokumentes im Textverarbeitungsprogramm
- Anschalten und konfigurieren des Druckers
- Befülen des Druckers mit Papier
- Drucken des Dokumentes
- Ausschalten des Druckers
- Ausschalten des Computers
- Unterschreiben des Dokumentes
- Stempeln des Dokumentes
- Frankieren des Dokumentes
- Versenden des Dokumentes

Angenommen alle benötigten Klassen sind bereits instanziiert, sieht der Ablauf zum Versenden eines Textes wie folgt aus:

```
1  String text = "Dies ist der Text zum versenden";
2  //Computer einschalten
3  computer.an();
4  //Neues Textverarbeitungsprogramm erzeugen
5  Textverarbeitungsprog textverarbeitungsprog = computer.
       getTextverarbeitungsprog();
6  //Textverarbeitungsprogramm öffnen
7  textverarbeitungsprog.oeffnen();
8  //Neues Dokument aus Text erzeugen
9  Dokument dokument = textverarbeitungsprog.getDokument(text);
10 //Drucker einschalten
11 drucker.an();
12 //Drucker einrichten
13 drucker.konfigurieren();
14 //Drucker mit Papier befüllen
15 drucker.papierNachfuellen();
16 //Dokument drucken
17 computer.drucke(dokument);
18 //Drucker ausschalten
19 drucker.aus();
20 //Computer ausschalten
21 computer.aus();
22 //Dokument unterschreiben
```

```
23 stift.unterschreiben(dokument);
24 //Dokument stempeln
25 stempel.stempel(dokument);
26 //Dokument mit Briefmarke versehen
27 briefmarkenautomat.briefmarkeBezahlen(dokument);
28 //Dokument in Briefkasten einwerfen
29 briefkasten.briefEinwerfen(dokument);
```

Listing 2.1: Ablauf zum Versenden eines Textes

Diese einzelnen Schritte sind von jedem Client, der ein Dokument erstellen und versenden will, auszuführen. Das führt natürlich zu redundantem Code, was die Wartbarkeit stark verschlechtert, denn bei einer Änderung der verwendeten Klassen müsste jeder einzelne Client wieder angepasst werden. Auch könnten einzelne Schritte vergessen werden, was zu einem Fehler führen würde (z.B.: Briefmarke nicht aufgeklebt). Hier kommt nun das Entwurfsmuster Facade ins Spiel: Anstatt jeden Client sämtliche Schritte durchführen zu lassen, wird die Klasse des Büroangestellten eingeführt, welche diese Aufgaben übernimmt. Dadurch bietet das gesamte System nach außen eine Schnittstelle (Fassade), welche von den Benutzern anstelle der einzelnen Klassen verwendet werden kann.

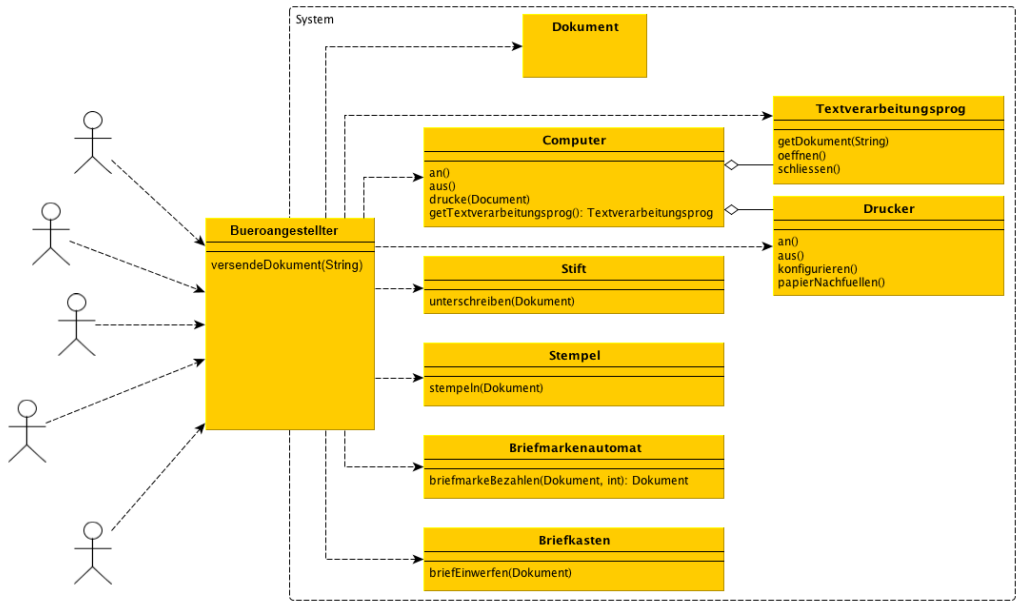

Abb. 2.3: Verwaltungs-System mit Fassade [Hau09a, S.185]

Der Code dieser Klasse könnte dann wie folgt aussehen:

```
1 class Bueroangestellter {
2
3   public Bueroangestellter(){
4
5     drucker = new Drucker();
6     computer = new Computer();
7     stift = new Stift();
```

```
8       stempel = new Stempel();
9       briefmarkenautmat = new Briefmarkenautomat();
10      briefkasten = new Briefkasten();
11    }
12
13    public void versendeDokument(String pString){
14
15      String text = pString;
16      computer.an();
17      Textverarbeitungsprog textverarbeitungsprog = computer.
            getTextverarbeitungsprog();
18      textverarbeitungsprog.oeffnen();
19      Dokument dokument = textverarbeitungsprog.getDokument(
            text);
20      drucker.an();
21      drucker.konfigurieren();
22      drucker.papierNachfuellen();
23      computer.drucke(dokument);
24      drucker.aus();
25      computer.aus();
26      stift.unterschreiben(dokument);
27      stempel.stempel(dokument);
28      briefmarkenautomat.briefmarkeBezahlen(dokument);
29      briefkasten.briefEinwerfen(dokument);
30    }
31
32    Drucker drucker;
33    Computer computer;
34    Stift stift;
35    Stempel stempel;
36    Briefmarkenautomat briefmarkenautomat;
37    Briefkasten briefkasten;
38  }
```

Listing 2.2: Die Klasse *Bueroangestellter*

Möchten Clients nun einen Text versenden, muss nur noch die entsprechende Methode der Büroangestellten-Klasse aufgerufen werden:

```
1 Bueroangestellter bueroAng = new Bueroangestellter();
2 bueroAng.versendeDokument("Das ist der Text zum versenden");
```

Listing 2.3: Versenden eines Dokumentes mit Hilfe der Fassade

Für die Benutzer wird der Vorgang dadurch wesentlich vereinfacht, da er sich nicht mehr mit den einzelnen Klassen auseinandersetzen muss. Sollen dennoch einzelne Funktionen verwendet werden, wie zum Beispiel das Versenden eines bereits frankierten Briefes, kann entweder die Büroangestellten-Klasse um diese Funktionalität erweitert werden oder es kann direkt auf die Briefkasten-Klasse zugegriffen werden.

2.4 Bewertung

Die Anwendung des Facade-Patterns bringt mehrere Vorteile mit sich.

1. Vereinfachung: Klienten kennen nur noch die Fassade und deren Methoden. Das Programmieren wird vereinfacht und der Umgang mit dem komplexen Subsystem wird vereinfacht.[Gam94, S.187]

2. Die Fassade fördert die schwache Bindung zwischen den Klienten und dem Subsystem. Das unter der Fassade liegende System kann verändert oder komplett ausgetauscht werden, ohne die Klienten zu beeinflussen. Dadurch werden Abhängigkeiten zwischen Klient und System minimiert, die getrennte Entwicklung dieser beiden Komponenten erleichtert.[Gam94, S.187].

3. Bei Bedarf können die Klienten dennoch die einzelnen Subsystem-Klassen verwenden und auf deren Funktionen zugreifen (Wahl zwischen einfacher und komplexer Bedieunung/Programmierung). [Gam94, S.188]

Literaturverzeichnis

[EG10] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON JOHN VLISSIDES: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*, 2010.

[Eil10] EILEBRECHT, STARKE: *Patterns kompakt: Entwurfsmuster fÃ¼r effektive Software-Entwicklung*, 2010.

[Gam94] GAMMA, HELM, JOHNSON VLISSIDES: *Design Patterns. Elements of Reusable Object-Oriented Software*, 1994.

[Hau09a] HAUER, PHILIPP: *Das Decorator Design Pattern*. `http://www.philipphauer.de/study/se/design-pattern/decorator.php`, 2009.

[Hau09b] HAUER, PHILIPP: *Das Facade Design Pattern*. `http://www.philipphauer.de/study/se/design-pattern/facade.php`, 2009.