Entwurfsmuster

Ausarbeitung von
Sebastian Pötzsch
8. Semester

Wegweisende Arbeiten in der Softwaretechnik
Lehrstuhl für Angewandte Telematik / e-Business
Universität Leipzig

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Eine Stärke der objektorientierten Programmierung liegt in der Wiederverwendung und die damit verbundene Reduzierung der Entwicklungs- und Wartungskosten. Ein Entwurf muss deshalb nicht nur die vorliegenden Anforderungen erfüllen, sondern auch flexibel und allgemein genug sein, um spätere Bedingungen und Funktionalität zu erfüllen. Gleichzeitig lässt die objektorientierte Programmierung Entwicklern viel Spielraum bei ihren Entwürfen. Entwickler müssen viele Entscheidungen treffen. Sie müssen die richtige Granularität bestimmen, passende Schnittstellen definieren und die Beziehungen zwischen den Klassen festlegen. Dabei ist es oft nicht sinnvoll, die reale Welt 1:1 in einem Modell abzubilden. Die objektorientierte Programmierung lässt Entwicklern bei diesen Entscheidungen viele Möglichkeiten offen und dabei natürlich auch die Möglichkeit sich für einen schlechten Entwurf zu entscheiden. Anfänger fühlen sich oft überfordert mit dieser hohen Flexibilität und es dauert oft lange bis sie zu Experten werden. Um diesen Lernprozess zu verkürzen, muss man verstehen, was einen Experten ausmacht.

Experten haben einen sehr großen Erfahrungsschatz. Mit jedem Problem, das sie einmal erfolgreich gelöst haben, haben sie Strategien gelernt, die funktionieren, aber auch solche, die nicht den erwünschten Erfolg brachten. Aus diesem Vorrat an Lösungen können sie immer wieder schöpfen, wenn sie auf bekannte Probleme treffen. Sie müssen deshalb nicht jedes Problem von neuem lösen. Stattdessen verwenden sie ihre Lösungen wieder und wieder. Dieses Wissen fehlt Anfängern.

Viele Probleme ähneln sich, deshalb lassen sie sich häufig auf ein Grundproblem reduzieren, für welches meistens eine Standardlösung existiert. Bei diesen Lösungen lassen sich bestimmte Muster erkennen. Die Verbreitung dieser Muster hilft Expertenwissen auszutauschen und ermöglicht es Anfängern schneller gute Entwürfe zu machen. Das bedeutet, man kann auf die Erfahrung anderer zugreifen und sich ihrer bedienen. Solche Muster helfen Entwicklern einen Entwurf schneller “richtig“ zu machen.

Gute Entwürfe mittels Entwurfsmuster zu beschreiben hat viele Vorteile. Sie beschreiben ein

wiederkehrendes Problem, geben ihm einen Namen, beinhalten Informationen über dessen Lösung und Vor- und Nachteile der Anwendung. Sie veranschaulichen die beteiligten Objekte, ihre Rollen und ihre Zusammenarbeit. Entwurfsmuster sind kaum noch aus der objektorientierten Entwicklung wegzudenken.

- Entwurfsmuster bieten Entwicklern ein Vokabular, das es ihnen ermöglicht über Entwürfe zu sprechen, sie zu dokumentieren und über Alternativen zu diskutieren.

- Sie konzentrieren das Wissen von Experten. Man kann Entwurfsmuster wie Bausteine einsetzen, um komplexere Entwürfe zu realisieren.

- Entwurfsmuster ermöglichen es Anfängern schneller Klassenbibliotheken zu erlernen. Sie helfen Erfahrungen zu teilen und wieder zu verwenden. Anfänger werden mit ihrer Hilfe schneller zu Experten.

- Entwurfsmuster helfen richtige Klassenhierarchien zu bestimmen.

Erich Gamma und seine Mitstreiter haben viele dieser Entwurfsmuster gefunden, definiert, eingeordnet, bewertet und veröffentlicht und damit beigetragen Expertenwissen schneller zu verbreiten. Sie propagierten gutes Design und halfen Anfängern sich wie Experten zu benehmen.

1.2 Geschichte

Die eigentliche Grundidee der Entwurfsmuster stammt von dem Architekten Christopher Alexander. Er suchte nach den grundlegenden Eigenschaften guter Architektur, wie Freiheit, Komfort, Vollständigkeit, Harmonie, Haltbarkeit, Anpassungsfähigkeit, Flexibilität etc. Dabei stieß er auf wiederkehrende Muster, die aufeinander aufbauen und miteinander agieren. Alexander fand 253 Muster, von der Region über die Stadt, Stadtteile, Gebäude, Räume bis hin zur Innenausstattung, die er 1977 in seinem Buch “A Pattern Language“ [2] beschrieb. Diese Muster helfen komplexe Architektur zu beschreiben und in ihre Einzelteile zu zerlegen. Alexanders Ideen waren sehr theoretisch und fanden bei anderen Architekten wenig Anerkennung.

Jahre später erkannten Ward Cunningham und Kent Beck das Potential dieser Theorie für die Softwaretechnik. Sie entwickelten User Interfaces mit Smalltalk und benutzen Alexander's Ideen, um mittels Entwurfsmuster Smalltalk Anfänger schneller einzuarbeiten. Sie begannen 1987 ihre Ideen der objektorientierten Gemeinde näher zu bringen [1], doch vieles war noch zu theoretisch und praktische Beispiele fehlten.

Später fing Jim Coplien an einen Katalog von C++ Idiome zusammenzustellen. Er veröffentlichte sie 1992 in seinem Buch "Advanced C++ Programming Styles and Idioms" [3]. Idiome sind spezielle Muster für C++.

Erich Gamma fand das Muster sprachunabhängig sein müssten. Bei seiner Arbeit mit ET++, einem Framework für interaktive grafische Anwendungen, fing er an seine Entwurfsmuster zu dokumentieren und schnell hatte er ein Dutzend zusammen. Erich Gamma fand in Richard Helm, John Vlissides und Ralph Johnson Mitstreiter für seine Ideen und sie beschlossen einen Katalog mit Entwurfsmustern herauszubringen. Sie nannten sich die "Gang of Four", oder kurz "GoF". In ihrem Buch "Design Patterns: Elements of Reusable Object-Oriented Software" [5] stellten sie 23 Muster vor, zeigten anhand von Beispielen ihren Nutzen und führten ein einheitliches Schema zur Beschreibung von Entwurfsmustern ein. Das Buch sorgte für Aufsehen, und es begann sich schnell eine Entwurfsmuster-Gemeinde zu bilden.

Heute existieren zahlreiche Entwurfsmuster und es gibt zahlreiche Bücher über Anwendung und Nutzen von Entwurfsmustern.

2 Entwurfsmuster

2.1 Definition

Christopher Alexander definiert ein Muster folgender Maßen: “Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen.“ [2].

Erich Gamma beschreibt das ein Entwurfsmuster drei Eigenschaften erfüllen muss. Es muss nützlich, anwendbar und benutzt sein (Useful, Useable and Used), was soviel bedeutet, dass es ein häufig vorkommendes Problem sinnvoll und einfach lösen kann.

Entwurfsmuster bestehen aus vier grundlegenden Teilen.

1. Der Mustername, er benennt kurz das Problem, die Lösung und die Auswirkungen. Der Name ermöglicht es dem Programmierer über Muster und Entwürfe zu sprechen. Das Finden kurzer prägnanter Namen war mit die schwierigste Aufgabe der “GoF“.

2. Der Problemabschnitt, er beschreibt den Kontext, in dem das Muster angewendet werden kann und welches Problem damit gelöst wird. Die Problembeschreibung kann Bedingungen aufführen, wann es sinnvoll ist das Entwurfsmuster einzusetzen.

3. Der Lösungsabschnitt, er beschreibt die Objekte und Klassen und ihre Zusammenarbeit, um zu zeigen, wie das Problem gelöst wird. Er gibt keine konkrete Implementierung an.

4. Die Konsequenzen, beschreiben die Vor- und Nachteile der Musteranwendung. Sie dienen zum Vergleich und zur Bewertung und tragen zum Verständnis des Musters bei.

2.2 Organisation der Entwurfsmuster

Damit man einen besseren Überblick über die vielen Entwurfsmuster gewinnt, hat man sie nach verschiedenen Kriterien eingeteilt. Diese Organisation vereinfacht das Erkennen von verwandte Muster, das Lernen und das Finden neuer Muster. Die Tabelle 2.1 zeigt die klassische Einteilung. Die Entwurfsmuster sind nach zwei Hauptkriterien eingeteilt wurden, der Aufga-

Gültigkeitsbereich	**Aufgabe**		
	Erzeugungsmuster	**Strukturmuster**	**Verhaltensmuster**
klassenbasiert	Factory Method	Adapter (class) Bridge (class)	Template Method
objektbasiert	Abstract Factory Prototype Singleton	Adapter (object) Bridge (object) Flyweight Facade Proxy	Chain of Responsibility Command Iterator (object) Mediator Memento Observer State Strategy
kompositionsbasiert	Builder	Composite Decorator	Interpreter Iterator (compound) Visitor

Tabelle 2.1: Einteilung von Entwurfsmustern

be eines Muster und seinem Gültigkeitsbereich. Der Gültigkeitsbereich beschreibt ob sich das Muster auf Klassen, Objekte oder zusammengesetzte Objekte bezieht. Klassenbasierte Muster beschreiben Beziehungen, die statisch, das heißt zur Laufzeit, festgelegt sind. Sie beziehen sich auf die Beziehungen zwischen Klassen und ihren Unterklassen. Objektbasierte Muster hingegen beschreiben dynamische Beziehungen, die zur Laufzeit geändert werden können. Kompositionsbasierte Muster beschreiben rekursive Beziehungen. Es gibt auch Muster, die sowohl objektbasiert als auch klassenbasiert sind, aber die meisten Entwurfsmuster sind objektbasiert.

Das zweite Kriterium, die Aufgabe eines Musters, verdeutlicht, was ein Muster macht. Die Aufgaben eines Musters werden nochmals unterteilt, ob ein Muster erzeugende, strukturorientierte oder verhaltensorientierte Aufgaben hat.

Erzeugungsmuster befassen sich mit Problemen die beim Erzeugen von Objekten entstehen. Sie helfen dabei spezifische Details der Erzeugung eines Objektes zu verstecken. Gibt man beispielsweise bei der Instanzierung den Namen eine Klasse an, so legt man sich auf eine bestimmte Implementierung fest, was zukünftige Änderungen erschwert. Erzeugungsmuster kapseln Informationen über die Art und Weise, wie Objekte erzeugt und zusammengefügt werden.

Klassenbasierte Erzeugungsmuster benutzen dazu Vererbung, während objektbasierte Erzeugungsmuster die Erzeugung in Unterklassen verlagern. Kompostionsbasierte Erzeugungsmuster erzeugen rekursive Objektstrukturen.

Strukturmuster helfen bei Problemen, die die Zusammensetzung von Objekten betrifft. Klassenbasierte Strukturmuster lösen diese Art von Problemen mittels Vererbung. Sie können zum Beispiel helfen, zwei verschieden Klassen zu einander kompatibel zu machen. Ein Beispiel hierfür ist das Adapter Muster. Objektbasierte Strukturmuster ermöglichen die Zusammensetzung unterschiedlicher Objekt mittels Objektkomposition, um zusätzliche Funktionalität der Objekte zu erreichen.

Verhaltensmuster beschreiben wie die Objekte miteinander interagieren. Sie konzentrieren sich auf das Objektverhalten. Klassenbasierte Verhaltensmuster verwenden hierfür wieder Vererbung, während objektbasierte Verhaltensmuster Komposition benutzen.

2.3 Beschreibung von Entwurfsmustern

Es gibt viele unterschiedliche Entwurfsmuster, die sich in ihrem Abstraktionsgrad und ihrer Granularität unterscheiden. Grafische Notationen reichen da nicht aus, um ein Entwurfsmuster zu beschreiben. Die „GoF“ hat deshalb ein einheitliches Schema zum Beschreiben von Entwurfsmustern eingeführt [4]. Es soll helfen Entwurfsmuster zu verstehen, zu vergleichen, zu verwenden und ihre Vor- und Nachteile aufzuzeigen.

Musternname Der Name sollte die wesentliche Funktion des Entwurfsmusters beschreiben. Ein guter Name ist dabei sehr wichtig, da er Teil des Entwurfsvokabular werden wird. Er sollte kurz und leicht zu merken sein.

Zweck Was macht das Muster? Was ist sein Prinzip und sein Zweck? Welches Problem kann es lösen?

Motivation Zeigt ein Szenario, indem das Muster angewendet werden kann, ein Entwurfsproblem, an das sich das Muster richtet und die Klassen- und Objektstrukturen, die das Problem lösen.

Anwendbarkeit In welchen Situationen kann das Muster angewendet werden? Woran erkenne ich solche Situationen?

Teilnehmer Beschreibt die beteiligten Klassen, Objekte und ihre Zuständigkeiten.

Interaktion Beschreibt, wie die einzelnen Teilnehmer zusammenarbeiten.

Struktur Eine grafische Darstellung des Musters basierend auf der Object-Modeling-Technique (OMT).

Konsequenzen Welche Vor- und Nachteile hat das Muster? Was sind Alternativen? Wie versucht das Muster seine Ziele zu erreichen?

Implementierung Welche Fallen, Tips oder Techniken muss man beachten? Gibts es sprachspezifische Aspekte?

Beispiele Zeigt existierende Beispiele der realen Welt in denen das Muster angewandt wurde.

Siehe Auch Was sind verwandte Muster? Was sind Unterschiede zu ihnen? Zusammen mit welchen Mustern sollte ich es verwenden?

3 Der Katalog

Hier werden nun einige Entwurfsmuster aus dem Originalkatalog [5] von der „GoF“ genauer beschrieben. Ich beschränke mich hier auf die bekanntesten Entwurfsmuster. Aus Platzgründen wird die “GoF“-Form zur Beschreibung von Entwurfsmustern, wie siehe im Abschnitt 2.3 auf Seite 11 vorgestellt wurde, etwas kompakter dargestellt.

3.1 Strategy

Stategy ist ein objektbasiertes Verhaltensmuster.

Name

Strategy

Zweck

Kapsle einzeln die verschiedene Algorithmen für das selbe Problem, um sie besser optimieren, variieren oder austauschen zu können.

Motivation

Es gibt unterschiedliche Arten Dateien zu komprimieren. Codiert man die zugehörigen Algorithmen fest in die Klienten, wird es schwierig sie wieder zu verwenden, zu ändern oder auszutauschen. Die Klienten können dadurch zu komplex und unflexibel werden. Es kommt auch auf den Kontext an, welchen Algorithmus man vorzieht. Den schnelleren oder den, der die

stärkste Kompression liefert? Das Strategy Muster hilft uns hier die verschiedenen Algorithmen zu kapseln und aus den Klienten auszulagern. Für unser Beispiel würde das bedeuten, dass es eine Schnittstelle Komprimierer gibt, die die unterschiedlichen Komprimierer implementieren, beispielsweise ein ZIPKomprimierer, oder ein HuffmanCodeKompimierer. Eine Klient hält eine Referenz auf einen Komprimierer und überträgt ihm die Aufgabe des Komprimierens.

Anwendbarkeit

Das Strategy Muster bietet sich an, wenn

- sie unterschiedliche Varianten eines Algorithmus verwenden.
- dem Klienten Details eines Algorithmus verborgen bleiben sollen.
- sich ihre Klassen nur im Verhalten unterscheiden.
- ihre Klassen mehrfache Bedingungsanweisung enthalten.

Struktur

Die Struktur des Strategy Musters ist in Abbildung 3.1 dargestellt.

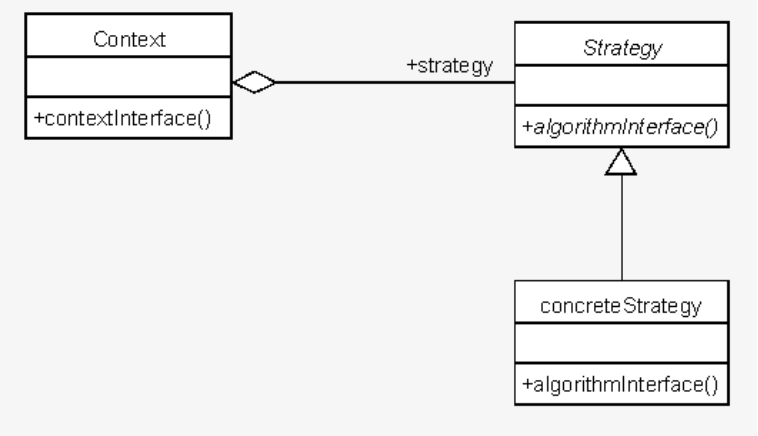

Abbildung 3.1: Strategy

Beispielcode

Die Klasse `FileHandler` bietet Funktionen zum Öffnen, Speichern, Anhängen und komprimieren von verschiedenen Dateien. Das Komprimieren wird an einen `Compressor` übergeben, je nach dem mit welchem konkreten `Compressor` man den `FileHandler` initialisiert hat, wird der Algorithmus ausgeführt.

```
// Unser Context
public class FileHandler {
        // der Komprimierer
        private Compressor compressor;
        private File file;

        public FileHandler(Compressor compressor) {
         this.compressor = compressor;
        }

        void saveFile() {
        //...
        }

        // uebergib dem Komprimierer die Aufgabe
        File compress(File file) {
         return compressor.compress(file);
        }
        //...
}
```

Die Schnittstelle `Compressor` bietet die Methode `compress` an. Es gibt verschieden Möglichkeiten, wie die vom Algorithmus benötigten Daten, vom Klienten übergeben werden. Entweder man übergibt der Funktion nur die notwendigen Daten, was weise Vorausicht erfordert, da man bei neuen Algorithmen nicht jedesmal die Schnittstelle ändern will, oder der Klient übergibt sich selbst und der Algorithmus entscheidet, welche Daten er benötigt. In unserem Beispiel wird die erste Methode benutzt.

```
// Die Strategy Schnittstelle
public interface Compressor {

        // Methode zum komprimieren
        File compress(File file);
}
```

Eine Variante zur Komprimierung wäre der `ZIPCompressor`.

```
// Eine konkrete Strategie zum komprimieren
public class ZIPCompressor implements Compressor {
```

```
        // ZIP-Verfahren zum Komprimieren
        public File compress(File file) {
         // komprimiere mittels ZIP-Verfahren
         //...
         return file;
        }
}
```

Der `HuffamCodeCompressor`.

```
// eine andere Strategie
public class HuffamCodeCompressor implements Compressor {

        // Huffman-Code-Verfahren
        public File compress(File file) {
         // komprimiere mit Huffman-Code-Verfahren
         //...
         return file;
        }
}
```

Abhängig vom Kontext könnte man jetzt bequem die verschiedenen Strategien benutzen.

```
FileHandler ZIPFileHandler = new FileHandler(new ZIPCompressor());
// oder
FileHandler HCFileHandler = new FileHandler(new HuffamCodeCompressor());
```

Man sieht, dass man leicht die Algorithmen ändern, austauschen oder verbessern könnte. Ein Nachteil des Strategy Musters ist, dass der Klient die Algorithmen und möglicherweise auch ihre Unterschiede kennen muss, um sie nutzen zu können.

3.2 Decorator

Decorator ist ein objektbasiertes Strukturmuster.

Name

Decorator

Zweck

Erweitere die Funktionalität einer Klasse, möglichst flexibel, und ohne neue Unterklassen zu bilden.

Motivation

Betrachten wir als Beispiel visuelle Komponenten, wie man sie von vielen Programmen her kennt. Es gibt Dialogboxen, Textfelder, Register, Menüs usw., die alle visuelle Komponenten mit bestimmten Grundfunktion sind. Nun könnte man wollen, einige Komponenten mit Rahmen, Menüs oder Scrollbars zu versehen. Manchmal will man Komponenten mit Scrollbar, mit Rahmen, ohne Rahmen aber mit Scrollbar, mit Menü und Rahmen aber ohne Scrollbar, usw. Dieses Beispiel zeigt, das viele Unterklassen nötig wären, um alle diese unterschiedlichen Komponenten zu realisieren. Die spätere Erweiterung um neue Komponenten wäre aufwendig. Hier hilft der Decorator. Er selbst erbt von der visuellen Komponenten Klasse und spezielle Decorator erweitern sie entsprechend. Am Ende gibt es dann einen ScrollbarDecorator, MenuDecorator, FrameDecorator usw. Diese kann man dann beliebig kombinieren.

Anwendbarkeit

Folgenden Situationen bieten sich an, um dieses Muster anzuwenden.

- Wenn sie die Funktionalität von Objekten erweitern wollen, ohne andere Objekte zu verwenden.
- Wenn sie Funktionalität hinzufügen wollen, die auch leicht wieder zu entfernen ist.
- Wenn eine Erweiterung mittels Vererbung zu umständlich ist, weil zum Beispiel zu viele Unterklassen entstehen würden.
- Wenn sie eine Klasse erweitern wollen, deren Quellcode sie nicht kennen.

Struktur

Das Decorator Muster in UML Notation (siehe Abbildung 3.2).

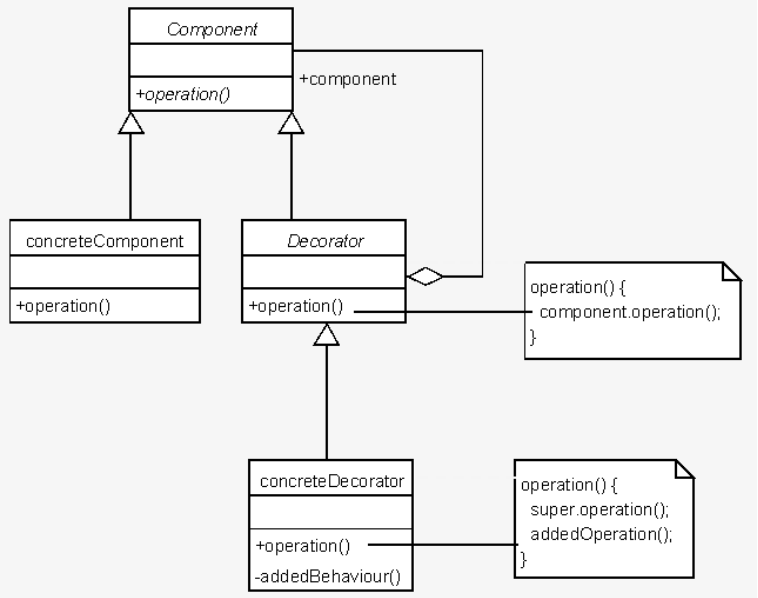

Abbildung 3.2: Decorator

Beispielcode

Für das Beipiel aus der Motivation, sähe der Quellcode folgendermaßen aus. `VisualComponent` hat die Standardfunktion `showComponent`, die das Fenster auf den Bildschirm zeichnet.

```
// Unsere Component-Klasse
public abstract class VisualComponent {
        //...
        // vorhandene operation
        public abstract void showComponent();
        //...
}
```

Der `Decorator` erbt von `VisualComponent` und fügt zusätzliche Funktionen hinzu. Vorhandene Funktionen leitet er einfach weiter.

```
// Unser Decorator
public abstract class Decorator extends VisualComponent {

        private VisualComponent comp;

        public Decorator(VisualComponent comp) {
         this.comp = comp;
        }
```

```
        // Leite operation weiter
        public void showComponent() {
         comp.showComponent();
        }
        //...
}
```

Unser `ScrollbarDecorator` überschreibt `showComponent()`. Sie zeigt jetzt auch einen Scrollbar.

```
// Die Implementation eines konkreten Decorator
public class ScrollbarDecorator extends Decorator {

        public ScrollbarDecorator(VisualComponent comp) {
         super(comp);
        }

        // Ueberschreibe Standardfunktion und fuege zusaetzliche Funktion zu
        public void showComponent() {
         super.showComponent();
         showScrollbar();
        }

        // Zusatzfunktion
        private void showScrollbar() {
        //...
        }
}
```

Wenn man jetzt eine `VisualComponent`-Klasse `Window` hat, die andere `VisualComponent`-Objekte beinhaltet, beispielsweise ein `Panel`-Objekt, kann man mit der Funktion `setContent()` einfach `VisualComponent`-Objekte hinzufügen.

```
        void setContent(VisualComponent comp) {
        //...
        }
```

Um ein `Panel` mit Rahmen und Scrollbar zu einem Fenster hinzuzufügen, muss man nur noch folgendes tun:

```
        Window window = new Window();
        Panel panel = new Panel();

        window.setContent(
                new ScrollbarDecorator(
                        new FrameDecorator(panel)
                )
```

```
                    );
```

3.3 Observer

Observer ist ein objektbasiertes Verhaltensmuster.

Name

Observer

Zweck

Der Zustand eines Objektes wird von anderen Objekten beobachtet, wenn sich der Zustand des Objektes ändert sollen die beobachtenden Objekte benachrichtigt werden.

Motivation

Angenommen man hat eine Tabelle die Daten enthält. Man könnte diese Daten mittels eines Balkendiagramms und gleichzeitig mittels eines Tortendiagramms veranschaulichen. Ändern sich die Werte in der Tabelle, dann möchte man das sich die Anzeigen der verschiedenen Diagramme mit ändern, möglichst ohne die genaue Implementierung der Diagramme zu wissen oder deren Anzahl. Dieses Problem kann das Observer-Muster lösen.

Anwendbarkeit

Es wird empfohlen, das Muster in folgenden Situationen anzuwenden.

- Wenn man unterschiedliche Sichtweisen auf das selbe Objekt hat. Man kann so die Sichtweisen leichter variieren und später wieder verwenden.

- Wenn man möchte, dass die Änderung des Zustandes eines Objektes bewirkt, dass andere abhängige Objekte davon benachrichtigt werden, und die Anzahl, der zu benachrichtigenden Objekte unbekannt ist.

- Wenn ein Objekt andere benachrichtigen soll, ohne Details über die zu benachrichtigenden Objekte zu kennen.

Struktur

Die Abbildung 3.3 zeigt den Aufbau des Observer Musters.

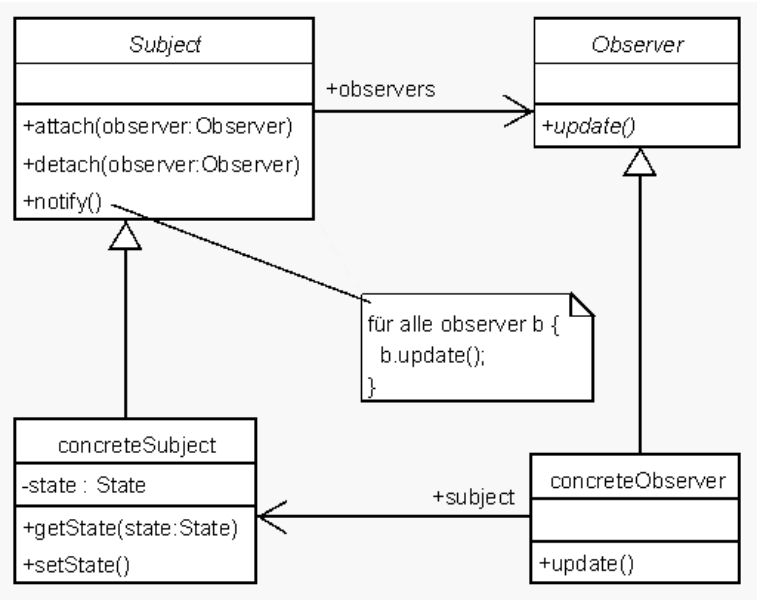

Abbildung 3.3: Observer

Beispielcode

Hier nun das Observer-Muster angewendet auf das Problem aus der Motivation. Als erstes definieren wir unsere Observer-Schnittstelle.

```
// Observer Schnittstelle
public interface Observer {
```

```
        // Methode zur Aktualisierung
        void update();
}
```

Unser Subjekt-Schnittstelle sieht wie folgt aus.

```
// Unsere Subjekt Schnittstelle
public interface Subject {
        // Observer anmelden
        void attach(Observer obs);
        // Observer abmelden
        void detach(Observer obs);
        // alle Observer benachrichtigen
        void inform();
}
```

Ein konkrete Implementation unseres Subjekts, was in unserem Beispiel Tabelle ist, sieht wie folgt aus. Die Methode `inform()` meldet allen angemeldeten Observer, das sie sich aktualisieren sollen.

```
// unsere konkretes Subjekt
public class Table implements Subject {
        // Liste aller angemeldeten Observer
        private ArrayList observers = new ArrayList();

        // die Daten
        private Data data;

        // Daten veraendern
        void setData(Data data) {
         this.data = data;
         inform();
        }

        // melde Observer an
        public void attach(Observer obs) {
         observers.add(obs);
        }

        // melde Observer ab
        public void detach(Observer obs) {
         observers.remove(obs);
        }

        // informiere alle angemeldeten Observer
        public void inform() {
         Iterator iter = observers.iterator();
         while (iter.hasNext()) {
          ((Observer) iter.next()).update(data);
```

```
        }
      }
}
```

Ein konkreter Observer ist in unserem Fall die `BarChart`-Klasse, wenn sie über eine Änderung der Tabelle informiert wird, aktualisiert sie die Daten und zeichnet sich neu.

```
// ein konkreter Observer
public class BarChart implements Observer {

      private Data data;

      public void update(Data data) {
       // aktualisiere die Daten
       this.data = data;
       // und zeichne Diagramm neu
       repaint();
      }

      // zeichne das Diagramm
      void repaint() {
      //...
      }
}
```

Es ist gut zu erkennen, dass es keinerlei Mühe macht, weitere Diagrammtypen zu implementieren, die die Tabelle beobachten. Dem Tabellen-Subjekt kann es vollkommen egal sein wieviele Observer beobachten, oder was sie genau machen.

3.4 Entscheidungshilfen

Es ist nicht immer leicht herauszufinden welches Muster mein Problem am besten lösen könnte. Im folgenden werden allgemeine Probleme beschrieben, die häufig Ursache einer Entwurfsrevision sind, und die Entwurfsmuster, die einem dabei helfen können.

1. Sie möchten sich nicht auf eine bestimmte Implementierung eines Objekte festlegen und ihre Objekte erzeugen ohne explizite Nennung des Klassenames. Entwurfsmuster die Objekte indirekt erzeugen sind Abstract Factory, Factory Method und Prototyp.

2. Ihre Software soll möglichst plattformunabhängig und leicht zu portieren sein. Also möchten sie unabhängig von externen Programmierschnittstellen und Bestriebssystemen sein. Hier helfen Abstract Factory und Bridge.

3. Sie wollen vermeiden das Klienten, die ihre Objekte benutzen abhängig von deren Implementierung oder Repräsentation sind. Dies würde bedeuten, dass sich eine Änderung der Objekte möglicherweise auf die Klienten überträgt. Beim Verstecken von Details der Implementierung sind die Entwurfsmuster Abstract Factory, Bridge, Memento und Proxy nützlich.

4. Sie verwenden unterschiedliche Algorithmen zur Lösung einer Aufgabe und möchte diese vielleicht später optimieren, austauschen oder verändern. Die Algorithmen müssen deshalb aus den Klassen gelöst werden. Muster, die Algorithmen von ihren Klassen trennen, sind Builder, Iterator, Strategy und Template Method.

5. Sie wollen eine enge Kopplung ihrer Klassen vermeiden damit sie sie später wieder verwenden können. Zu große Abhängigkeiten erschweren das Lernen, die Änderung und das Portieren von Klassen. Entwurfsmuster die eine lose Kopplung ermöglichen sind Abstract Factory, Bridge, Chain of Responsibility, Observer, Command, Facade oder Mediator.

6. Sie wollen auf bequemen Weg die Funktionalität von Klassen erweitern, ohne die Unterklassenanzahl explodieren zu lassen, oder lästige Folgeänderungen vorzunehmen. Versuchen sie es doch mal mit den Entwurfsmuster Bridge, Observer, Decorator, Strategy, Chain of Responsibility oder Composite.

7. Sie möchten Klassen ändern, die sich schwer oder gar nicht ändern lassen. Wenn einem zum Beispiel der Quellcode einer Klasse fehlt, bei Klassenbibliotheken ist das häufig der Fall, oder man durch die Änderung einer Klasse unzählige Unterklassen mit ändern müsste. Hier könnten die Muster Adapter, Decorator oder Visitor helfen.

Man sieht das es immer mehrere Möglichkeiten gibt und man selbst entscheiden muss, welches Muster man verwenden will. Viele Muster arbeiten eng mit anderen zusammen. Die Abbildung 3.4 [5] zeigt die Beziehungen zwischen den einzelnen Mustern.

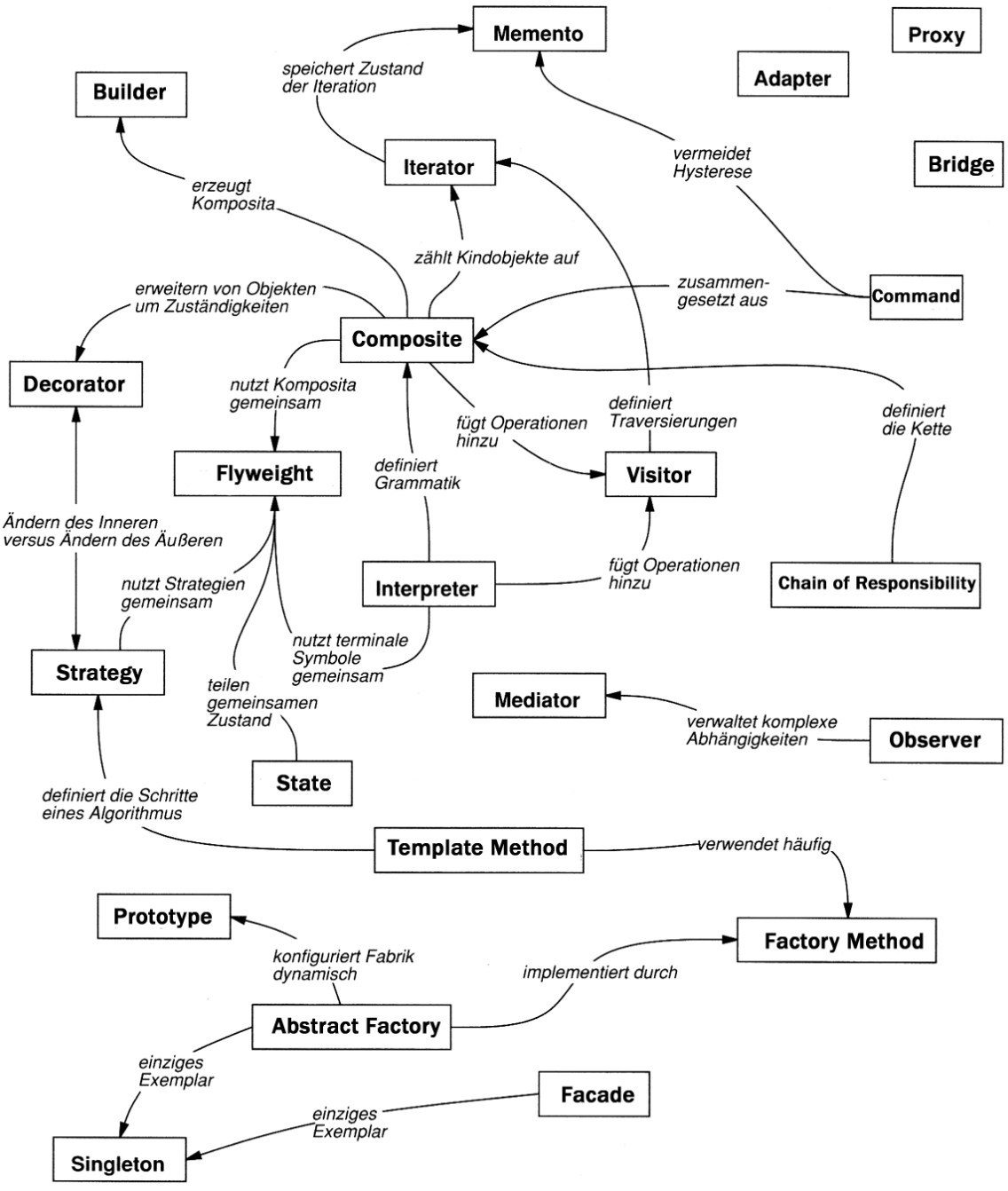

Abbildung 3.4: Beziehungen zwischen den einzelnen Entwurfsmustern

4 Zusammenfassung

4.1 Vorteile

Entwurfsmuster haben viele Vorteile und sind heute aus der Softwareentwicklung kaum noch wegzudenken. Sie helfen dabei Expertenwissen zu sammeln und schneller zu verbreiten. Man muss nicht mehr jedes Problem selbst lösen und dabei lange nach der besten Lösung suchen. Stattdessen greift man auf die Erfahrungen anderer zurück. Entwurfsmuster bieten einem nicht einfach eine Lösung für ein Problem, sondern auch Vor- und Nachteile dieser Lösung, verwandte Muster, Beispiele und Entscheidungshilfen. Auch weniger erfahrene Programmierer können so bessere Software entwerfen.

Entwurfsmuster fördern die Kommunikation zwischen Entwicklern. Sie helfen dabei über Entwürfe zu reden. Indem man dem Problem einen Namen gibt, kann man darüber reden. Es ist leichter anderen zu sagen, dass man einen Observer benutzt hat, anstatt die gesamte Klassenstruktur zu beschreiben. Somit erleichtern Entwurfsmuster das Erklären eines Entwurfs. Außerdem verbessern sie das Verständnis von Dokumentation und vereinfachen die Wartung von Software, das spart Zeit und Geld. Sie sind wieder verwendbar und sprachunabhängig. Der Entwickler muss sie nur noch implementieren.

4.2 Nachteile

Beim Einsatz von Entwurfsmustern ist die Gefahr groß, dass man sich zu sehr auf sie verlässt. Sie ersparen dem Entwickler nicht das Denken, er sollte sich immer Gedanken darüber machen, ob der Einsatz eines Entwurfsmusters sinnvoll ist, oder ob es den Entwurf unnötig aufbläht. Exzessiver Einsatz von Mustern nur um der Muster Willen sollte vermieden werden, dass heißt man sollte nie eine Anwendung so schreiben, dass man möglichst viele Muster einsetzen kann, oder die Anwendung extra für Entwurfsmuster anpassen. Gute Entwickler schreiben ihren ersten Entwurf ohne Bezug auf Entwurfsmuster, und suchen dann nach eventuell anwendbaren Mustern, um ihren Entwurf zu verbessern. Es gelingt selten einen Entwurf beim ersten Mal

gleich richtig zu machen.

Entwurfsmuster erhöhen die Anzahl von Klassen und können die Kosten unnötig in die Höhe treiben. Sie können einen Entwurf viel zu allgemein und flexibel gestalten, bei dem die Flexibilität vielleicht nie gebraucht wird. Häufig leidet darunter die Effizienz des Programms, deshalb sollten Muster nur dort angewendet werden, wo ihre Vorteile klar überwiegen. Der Konsequenzenabschnitt eines Musters kann helfen die Vor- und Nachteile eines Musters abzuschätzen.

Der riesige anwachsende Katalog von Entwurfsmustern sorgt dafür, dass man viel Zeit damit verbringt das richtige Muster zu finden. Es bedarf schon viel Einarbeitungszeit, mit Entwurfsmustern richtig umzugehen. Wenn man aber Entwurfsmuster gezielt und überlegt einsetzt, helfen sie einem dabei ein gutes Design zu schaffen, und sich mit seinem Entwurf wohl zu fühlen.

4.3 Abschluss

Die Grundidee der Entwurfsmuster stammt von Christopher Alexander und obwohl sie sich in der Architektur nicht durchsetzen konnte, fand sie in der Softwaretechnik viele Anhänger. Entwurfsmuster sind keine neue Erfindung, viele Entwickler kannten sie schon, haben sie eingesetzt und kein Wort darüber verloren. Die Arbeiten von Erich Gamma, Richard Helm, John Vlissides und Ralph Johnson trugen dazu bei, Entwurfsmuster zu verbreiten. Sie gaben ihnen Namen und verbreiteten so das Expertenwissen. Durch die einheitliche Beschreibung der Muster und deren Organisation halfen sie Entwicklern ihre Entwürfe zu verbessern. Es gibt heute zahlreiche Muster und ihre Verbreitung ist nicht mehr aufzuhalten. Entwurfsmuster gehören einfach zur guten Softwareentwicklung dazu. Es ist Erich Gamma zu verdanken, dass sich Anfänger wie Experten verhalten können. Entwurfsmuster haben so Software eine neue Qualität gegeben.

A Literaturverzeichnis

[1] http://www.c2.com/doc/oopsla87.html.

[2] Alexander, C., Ishkawa, S., and Silverstein, M., Eds. *A Pattern Language*. Oxford University Press, New York, 1977.

[3] Coplien, J., Ed. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, MA, 1992.

[4] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Eds. *Design Patterns: Abstraction and Reuse of Object-Oriented Design ECOOP '93, Lecture Notes in Computer Science 707*. Springer, Berlin Heidelberg New York, 1993.

[5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Eds. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

B Abbildungsverzeichnis