DESIGN-PRINZIPIEN & DESIGN-PATTERNS

SEW S05

14. DEZEMBER 2014

STEFAN ERCEG
4AHITT

Inhalt

1. Allgemeines

Entwurfsmuster beschreiben einen allgemeinen Ansatz für die Lösung eines häufig auftretenden Design Problems, welches bei gleichartigen Softwareaufgabenstellungen auftaucht. Der Vorteil von solchen Entwurfsmustern ist, dass man die jeweilige Software unabhängig von der zu implementierenden Programmiersprache betrachten kann. [1]

In der Schule wendeten wir ebenfalls z.B. das Strategy Pattern in Java an, das Factory Method Pattern jedoch in PHP.

2. Designprinzipien

2.1 Definition

Designprinzipien sind vordefinierte Richtlinien, die von bestimmten Design Patterns angewendet werden. [2]

2.2 Stark variablen Code vom statischen Code trennen

Bei einem Designentwurf sollte man sich überlegen, welche Teile vom Code gleich bleiben und welche sich oft ändern könnten. Somit kann man leicht änderbare Teile erweitern ohne dass man die statischen ändern muss. ([3], Seite 9)

Dieses Prinzip wird genauer folgendermaßen beschrieben: „Nehmen Sie die Teile, die sich ändern können, und kapseln Sie sie, damit sie diese veränderlichen Teile später ändern oder erweitern können, ohne dass das Auswirkungen auf die Teile hat, die sich nicht ändern.“ ([4], Seite 9)

2.3 Auf eine Schnittstelle statt auf eine Implementierung programmieren

Bevor das Prinzip besprochen wird, muss noch klargestellt werden, dass man unter „Schnittstelle“ nicht nur Interfaces, sondern auch abstrakte Klassen versteht.

Das Prinzip besagt, dass „ein Modul immer von einer klar definierten Schnittstelle des anderen Moduls abhängig sein sollte und nicht von der Art der Implementierung der Schnittstelle“. [5]

Dabei wird diese Schnittstelle getrennt von der Implementierung betrachtet. [5]

2.4 Komposition an Stelle von Vererbung

Unter „Komposition“ versteht man den Aufbau einer bestimmten Klasse aus anderen Klassen (Instanzvariablen sind Referenzen auf andere Klassen).
Unter „Vererbung“ versteht man die Weiterreichung von Eigenschaften und Verhalten an abgeleitete Klassen. [6]

Das Prinzip „Komposition der Vererbung vorziehen“ besagt, das eine Hat-Ein-Beziehung einer Ist-Ein-Beziehung überlegen sein kann. Dies ist flexibler, weil das Verhalten des Objekts zur Laufzeit verändert werden kann, solange das Objekt dir richtige Verhaltensschnittstelle implementiert. ([4], Seite 23)

2.5 Lose Kopplung

Je weniger Objekte einer Anwendung, die interagieren, voneinander wissen, desto weniger sind sie voneinander abhängig. Somit besitzen Änderungen von einem Objekt auf ein anderes keinen Einfluss und sie können unabhängig voneinander wieder verwendet werden. ([3], Seite 53)

2.6 Klassen offen für Erweiterungen / geschlossen für Änderungen

Falls ein Softwaremodul für verschiedene Umgebungen und Anwendungen verwendet werden soll, kann es bei Änderungen dazu kommen, dass der Code in allen Varianten des Moduls geändert werden muss. Dies soll jedoch vermieden werden. Die ursprüngliche Funktionalität eines Moduls soll sich durch Erweiterungsmodule anpassen lassen. Somit müssen keine Änderungen des Moduls getätigt werden, um es erweitern zu können. [7]

2.7 Auf Abstraktionen an Stelle von konkreten Klassen stützen

Eine Abstraktion beschreibt das Wesentliche eines bestimmten Begriffs, ohne die genauen Details zu erwähnen. Unterschiedliche Elemente, welche bestimmte gleiche Eigenschaften besitzen, können somit zusammengefasst werden. Bei einer Abstraktion von allen verschiedenen Betriebssystemen werden z.B. nur die gemeinsamen Fähigkeiten der Systeme besprochen und die genauen speziellen Fähigkeiten weggelassen.

Dieses Prinzip ist eine abgeschwächte Version des Prinzips „Programmierung auf Schnittstellen statt auf Implementierungen“ (siehe Punkt 2.3). Es beschreibt die Stützung auf Abstraktionen, damit man sich nicht auf genaue Implementierungen spezialisieren muss. Somit wird die direkte Abhängigkeit zwischen bestimmten Modulen aufgehoben und die Module sind dann nur noch von der gewählten Abstraktion abhängig. Der Begriff „Umkehr der Abhängigkeiten“ deutet an, dass eine bestehende Abhängigkeit umgedreht wird und eine Abhängigkeit von einer Abstraktion eine wesentlich kleinere Einschränkung besitzt als die Abhängigkeit von Konkretisierungen. [8]

2.8 Prinzip der Verschwiegenheit

Das Prinzip besagt, dass die Interaktion zwischen Objekten nur auf einige „gute Freunde“ beschränkt werden soll. Dies bedeutet, dass man sich bei jedem Objekt die Gedanken machen sollte, mit welchen Klassen es interagiert und wie es mit diesen interagiert.

Durch dieses Prinzip werden keine Entwürfe erstellt, in denen viele Klassen eines Moduls so aneinander gekoppelt sind, dass sich eine bestimmte Änderung eines Teilmoduls auf alle anderen Module auswirkt. Viele Abhängigkeiten sollten vermieden werden. ([3], Seite 265) , ([4], Seite 265)

2.9 Das Hollywood-Prinzip

Bei einem OO-Design kann es dazu kommen, dass „faule Abhängigkeiten“ entstehen. Diese treten auf, wenn man Highlevel-Komponenten besitzt, die von Lowlevel-Komponenten und diese wiederrum von Geschwisterkomponenten abhängig sind. Da bei solchen Modulen die Übersichtlichkeit nicht gewährleistet werden kann, wird das Hollywood-Prinzip angewendet.

Durch dieses Prinzip ermöglicht man Lowlevel-Komponenten, sich in ein Modul einzuhängen, während die Highlevel-Komponenten bestimmen, wann und wie sie erforderlich sind. Die Highlevel-Komponenten behandeln Lowlevel-Komponenten nach dem Prinzip „Rufen Sie uns nicht an, wir rufen Sie an!“. ([3], Seite 296)

2.10 Prinzip einer einzigen Verantwortung

Grundsätzlich trifft die Aussage zu, dass ein komplexeres Modul in mehrere Teilmodule unterteilt werden soll.

Jedes Modul besitzt genau eine einzige Verantwortung, durch welche das Modul verpflichtet ist, bestimmte Anforderungen umzusetzen. Nur falls sich diese Anforderungen ändern, besteht ein Bedarf, das Modul anzupassen. Dadurch lässt sich das Prinzip auch folgendermaßen formulieren: „Ein Modul sollte nur einen einzigen, klar definierten Grund haben, aus dem es geändert werden muss.“ [9]

3. Design Patterns

3.1 Strategy Pattern

Folgende Definition wird für das Strategy Pattern festgelegt:

„Das Strategy-Pattern definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Das Strategy-Pattern ermöglicht es, den Algorithmus unabhängig von den Clients, die ihn einsetzen, variieren zu lassen.“ ([4], Seite 24)

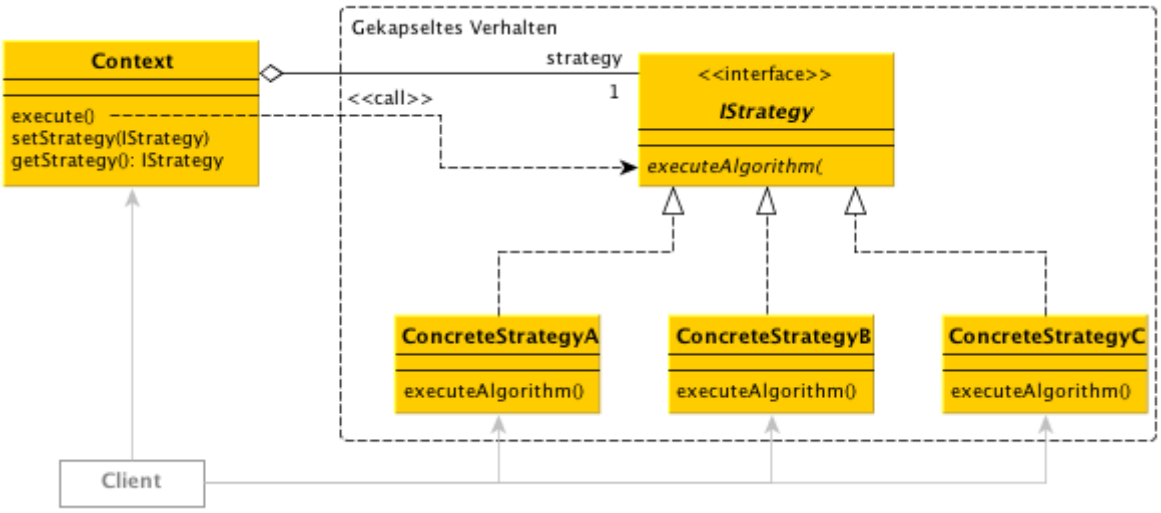

Abbildung 1: Strategy Pattern [Abb1]

Bei einem Strategy Pattern wird daher ein bestimmter Algorithmus eines Objekts in eine eigene Strategieklasse ausgelagert. Der Context besitzt eine Referenz auf das Strategy-Objekt und wenn dieser Algorithmus angewendet werden soll, wird ein Referenzaufruf an das Strategy-Objekt ausgeführt. Dabei kommuniziert der Context mit einer Schnittstelle und nicht mit einer konkreten Implementierung. Somit kann der Algorithmus geändert werden, ohne dass der Code vom Context geändert werden muss. Die konkrete Strategy implementiert das Strategy-Interface. [10]

Vorteile:

- Der Algorithmus eines Contextobjekts kann zur Laufzeit geändert werden, da der Algorithmus contextunabhängig und der Context implementierungsunabhängig ist.
- Es können neue Contextklassen mit einem anderen Algorithmus erstellt werden, ohne Vererbung anwenden zu müssen.
- Dieselben Funktionen, welche sich jedoch durch die Performance oder den Speicherbedarf unterscheiden, können durch verschiedene Implementierungen angeboten werden.

[10]

Nachteile:

- Für das Setzen der Strategy müssen die Implementierungen bekannt sein. Dadurch kommt es zu einer engen Kopplung des Clients an die konkreten Strategien. Deshalb sollte das Strategy Pattern nur angewendet werden, wenn das Variieren der Strategien für den Client wichtig ist.
- Im Strategy-Interface kann es dazu kommen, dass bestimmte Methoden einige Parameter entgegennehmen, die nicht von allen Strategien verarbeitet werden. Der Context erstellt und initialisiert diese trotzdem, weil er nur das Interface kennt und somit kommt es zu einem unnötigen Datentransfer.

[10]

Das Strategy Pattern setzt folgende Prinzipien um:

- Stark variablen Code von statischen Teilen trennen (Punkt 2.2)
- Programmierung auf Schnittstellen statt auf Implementierungen (Punkt 2.3)
- Komposition statt Vererbung (Punkt 2.4)

Ein praktisches Beispiel, wo das Strategy Pattern in der Programmiersprache Java angewendet wird, ist auf folgender Seite zu finden: [20]

3.2 Observer Pattern

Das Observer Pattern wird folgendermaßen beschrieben:

„Das Observer Pattern definiert eine Eins-zu-viele-Abhängigkeit zwischen Objekten in der Art, dass alle abhängigen Objekte benachrichtigt werden, wenn sich der Zustand des einen Objekts ändert.“ ([4], Seite 51)

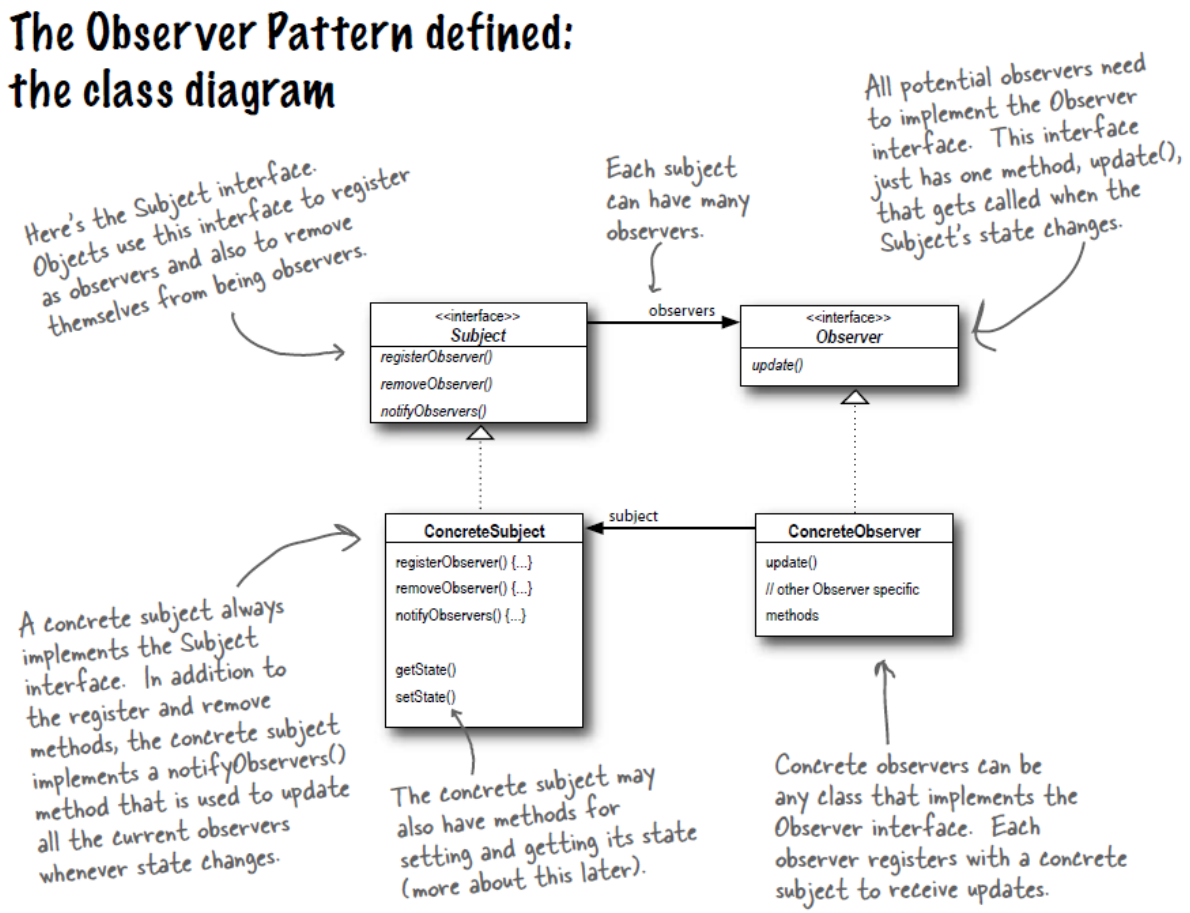

Abbildung 2: Observer Pattern ([Abb2_8], Seite 52)

Bei dem Observer Pattern werden zwei Seiten betrachtet: die Subjekt-Seite und die Observer-Seite. Das Subjekt, welches ein beobachtetes Objekt darstellt, ändert sich bei bestimmten Ereignissen. Die Observer, welche beobachtende Objekte darstellen, werden informiert, wenn sich beim Subjekt etwas ändert.

Dazu muss es beim Subjekt drei Methoden geben:

- registerObserver(Observer o): um die Observer zu registrieren
- removeObserver(Observer o): um bestimmte Observer entfernen zu können
- notifyObservers(): um die Observer bei Änderungen zu informieren

Beim Observer muss eine Methode existieren:

- update(): um bei Änderungen vom Subjekt die Informationen zu aktualisieren

Im Falle von Aktualisierungen wird vom Subjekt das Interface der Observer aufgerufen. Die konkreten Observer implementieren dieses Interface und damit die Update-Methode.

[11]

Es wird zwischen zwei Verfahren unterschieden:

- PUSH-Verfahren: Das Subjekt sendet den Observern die geänderten Informationen.
- PULL-Verfahren: Das Subjekt meldet den Observern, dass es Änderungen gibt, jedoch ist es die Aufgabe von den Observern, sich die neuen Informationen zu holen.

[12]

Vorteile:

- Die Kopplung zwischen den Objekten auf die Zeit zwischen An- und Abmelden des Observers am Subjekt bleibt begrenzt.
- Ein bestimmter Observer kann auch mehrere Subjekte beobachten. Bestimmte Klassen können zugleich Subjekt und Observer sein.
- Das Subjekt und die Observer können unabhängig voneinander variiert werden.
- Das Subjekt kann beliebig viele Observer aufnehmen, da es nur das Interface der Observer kennt. Somit sind das Subjekt und die Observer lose und abstrakt gekoppelt.

[11]

Nachteile:

- Besitzt man ein komplexes System, welches viele Subjekte und Observer besitzt, kann eine Änderung eine ganze Änderungskette nach sich ziehen oder sogar zu rekursiv wiederholenden Aufrufen führen.
- Falls man vergisst, den Observer beim Subjekt abzumelden, kann es in Fällen von Mehrfachanmeldungen zu komischen Nebeneffekten kommen und zur Verhinderung der automatischen Speicherfreisetzung, da das Subjekt weiterhin eine Referenz auf einen nicht mehr gebrauchten Observer besitzt.

[11]

Das Observer Pattern setzt folgende Prinzipien um:

- Stark variablen Code von statischen Teilen trennen (Punkt 2.2)
- Programmierung auf Schnittstellen statt auf Implementierungen (Punkt 2.3)
- Komposition statt Vererbung (Punkt 2.4)
- Lose Kopplung (Punkt 2.5)

Ein praktisches Beispiel, wo das Observer Pattern in der Programmiersprache Java angewendet wird, ist auf folgender Seite zu finden: [21]

3.3 Decorator Pattern

Das Decorator Pattern wird folgendermaßen beschrieben:

„Das Decorator Pattern fügt einem Objekt dynamisch zusätzliche Verantwortlichkeiten hinzu. Dekorierer bieten eine flexible Alternative zur Ableitung von Unterklassen zum Zweck der Erweiterung der Funktionalität.“ ([4], Seite 91)

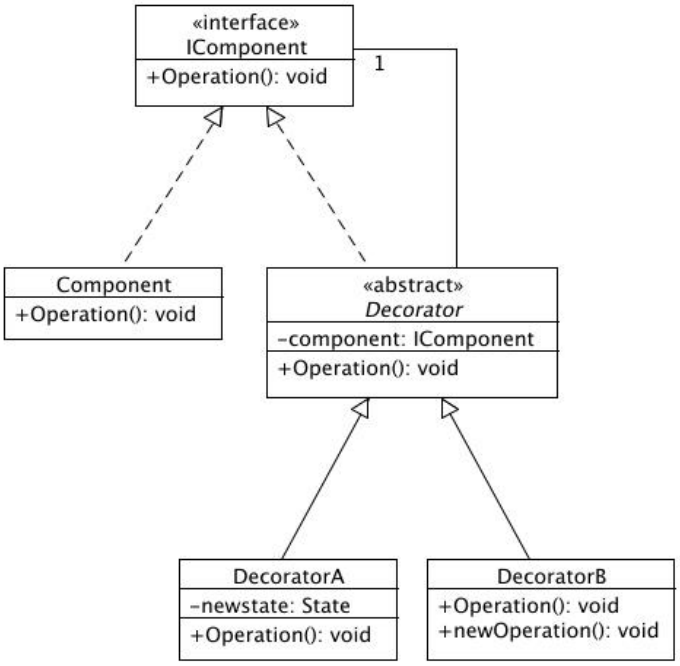

Abbildung 3: Decorator Pattern [Abb3]

Beim Decorator Pattern werden einer bestimmten Klasse dynamisch Fähigkeiten hinzugefügt. Dabei wird die Component-Klasse mit einigen Decorator-Klassen umhüllt. Die Dekorierer besitzen denselben Typen wie die Komponenten, da sie dieselbe Schnittstelle haben. Sie ändern das Verhalten der Komponenten, indem sie den Komponenten vor und/oder nach Methodenaufrufen neue Funktionalitäten hinzufügen. Eine Komponente kann mit beliebig vielen Dekorieren umhüllt werden und somit deren Fähigkeiten immer weiter ausbauen.

Man besitzt also eine Schnittstelle (Interface oder abstrakte Klasse), welche von den konkreten Komponenten und von der abstrakten Decorator-Klasse implementiert wird. Die konkreten Dekorierer erweitern schließlich die abstrakte Decorator-Klasse. [13]

Vorteile:

- Eine Klasse kann sowohl zur Kompilier-, als auch zur Laufzeit ohne statische Vererbung um Verhalten erweitert werden. Komponenten und Dekorierer sind voneinander unabhängig und können frei geändert und wiederverwendet werden.
- Es kommt zu keiner Unübersichtlichkeit aufgrund von vielen Vererbungshierarchien.
- Wenn ein Client mit einer Komponente arbeitet, kann ihm eine dekorierte Komponente untergeschoben werden.
- Es müssen nur die Funktionalitäten initialisiert werden, die auch wirklich benötigt werden.
- Unübersichtliche Basisklassen werden vermieden, da jeder Dekorierer nur eine Funktion besitzt.

[13]

Nachteile:

- Falls Fehler bei den oft lang verketteten Decorator-Objekten auftreten, sind diese schwer zu finden.
- Da jedes zusätzliche Feature eine neue Decorator-Klasse benötigt, kann es schnell zu vielen und ähnlichen Klassen und zur Unübersichtlichkeit kommen.
- Die Komplexität eines Moduls steigt mit der Einführung des Decorator Patterns.
- Es herrscht keine Objektidentität zwischen einer Komponente und einer dekorierten Komponente.

[13]

Das Decorator Pattern setzt folgende Prinzipien um:

- Stark variablen Code von statischen Teilen trennen (Punkt 2.2)
- Programmierung auf Schnittstellen statt auf Implementierungen (Punkt 2.3)
- Komposition statt Vererbung (Punkt 2.4)
- Offen/Geschlossen-Prinzip (Punkt 2.6)

Ein praktisches Beispiel, wo das Decorator Pattern in der Programmiersprache Java angewendet wird, ist auf folgender Seite zu finden: [22]

3.4 Factory Method Pattern

Das Factory Method Pattern wird folgendermaßen beschrieben:

„Das Factory Method Pattern definiert eine Schnittstelle zur Erstellung eines Objekts, lässt aber die Unterklassen entscheiden, welche Klassen instanziiert werden. Factory Method ermöglicht einer Klasse, die Instanziierung in Unterklassen zu verlagern.“ ([4], Seite 134)

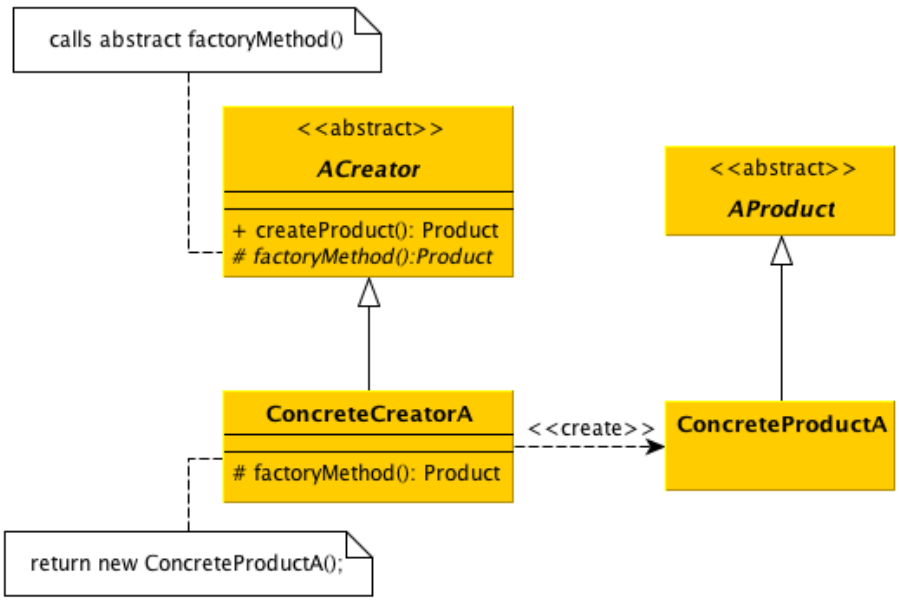

Abbildung 4: Factory Method Pattern [Abb4]

Bei dem Factory Method Pattern werden zwei Seiten betrachtet: die Creator-Seite und die Product-Seite. Diese Aufteilung wird gemacht, um ein erstelltes Objekt elegant austauschen zu können. Die Creator-Seite dient dabei zur Objektherstellung und die Product-Seite zur Objektverarbeitung.

Sowohl der Creator, als auch das Product besitzt jeweils eine abstrakte Klasse, die von den konkreten Creator- bzw. Produktklassen erweitert wird. Die konkrete Objektinstanziierung wird daher an die Unterklasse delegiert.

Der Creator kennt nur die Schnittstelle vom Product und instanziiert daher kein konkretes Product-Objekt, sondern lässt die Unterklassen entscheiden, welches konkrete Product-Objekt erzeugt werden soll.

[14]

Vorteile:

- Die Creator-Seite ist von einer konkreten Implementierung getrennt.
- Beliebig viele neue konkrete Creator- und Product-Klassen können erstellt werden, dank der Schnittstelle ohne Eingreifung in den Code des Clients.

[14]

Nachteile:

- Ein konkreter Creator ist mit einem konkreten Produkt eng gekoppelt. Erstellt man ein neues Produkt, muss zugleich ein neuer Creator geschrieben werden.

[14]

Das Factory Method Pattern setzt folgende Prinzipien um:

- Stark variablen Code von statischen Teilen trennen (Punkt 2.2)
- Programmierung auf Schnittstellen statt auf Implementierungen (Punkt 2.3)
- Offen/Geschlossen-Prinzip (Punkt 2.6)
- Umkehrung der Abhängigkeiten (Punkt 2.7)

Ein praktisches Beispiel, wo das Factory Method Pattern in der Programmiersprache Java angewendet wird, ist auf folgender Seite zu finden: [23]

3.5 Abstract Factory Pattern

Das Abstract Factory Pattern wird folgendermaßen beschrieben:

„Das Abstract Factory Pattern bietet eine Schnittstelle zum Erstellen von Familien verwandter oder zusammenhängender Objekte an, ohne konkrete Klassen anzugeben.“ ([4], Seite 156)

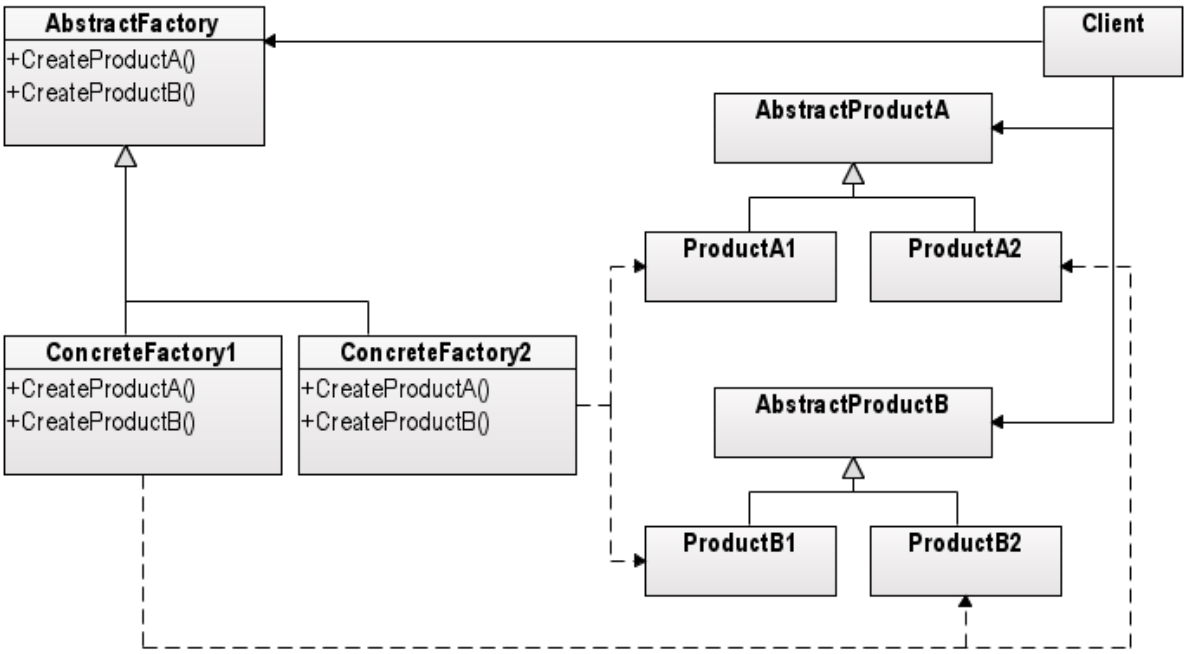

Abbildung 5: Abstract Factory Pattern [Abb5]

Beim Abstract Factory Pattern wird eine zusammenhängende Familie aus Produkten definiert. Die Familien können jederzeit ausgetauscht werden. Die Instanziierung erfolgt in einer konkreten Factory. Diese erweitert die abstrakte Factory-Klasse, welches für jede Produktart eine Methode definiert, mit der der Client eine Instanz des konkreten Produkts erhält. Sowohl bei der Factory, als auch bei den Produkten stützt man sich alleine auf Abstraktion. Der Client kennt nur diese Schnittstellen, die konkreten Implementierungen bleiben ihm verborgen. [15]

Vorteile:

- Es ist kein Code für spezielle Fälle notwendig, da der Clientcode durch das Abschirmen der konkreten Klassen allgemein gültig ist.
- Es wird sichergestellt, dass nur ein konkretes Familienmitglied eines Typs zur gleichen Zeit eingesetzt wird.
- Da sich der Client nur auf Abstraktionen stützt, können ganze Objektfamilien ohne Probleme ausgetauscht werden.
- Neue Produkte können einfach hinzugefügt werden.
- Konkrete Produkte können Mitglieder verschiedener Produktfamilien sein.

[15]

Nachteile:

- Die abstrakte Factory-Klasse muss bei einem Hinzufügen eines neuen Produkts geändert werden. Der Änderungsaufwand kann dabei sehr hoch sein. Je mehr man später am Modul erweitert, desto mehr Code muss geändert werden.

[15]

Das Abstract Factory Pattern setzt folgende Prinzipien um:

- Stark variablen Code von statischen Teilen trennen (Punkt 2.2)
- Programmierung auf Schnittstellen statt auf Implementierungen (Punkt 2.3)
- Offen/Geschlossen-Prinzip (Punkt 2.6)
- Umkehrung der Abhängigkeiten (Punkt 2.7)

Ein praktisches Beispiel, wo das Abstract Factory Pattern in der Programmiersprache PHP angewendet wird, ist auf folgender Seite zu finden: [24]

3.6 Singleton Pattern

Das Singleton Pattern wird folgendermaßen beschrieben:

„Das Singleton Pattern sichert, dass es nur eine Instanz einer Klasse gibt, und bietet einen globalen Zugriffspunkt für diese Instanz.“ ([4], Seite 177)

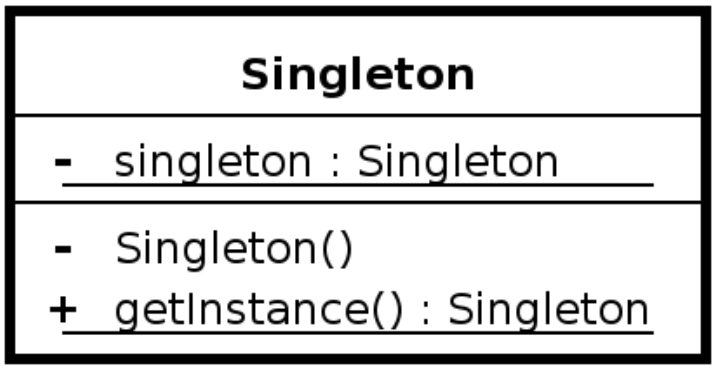

Abbildung 6: Singleton Pattern [Abb6]

Beim Singleton Pattern wird von einer Klasse nur eine einzige Instanz erzeugt und diese global zugänglich gemacht. Damit dies umgesetzt werden kann, muss eine Instanziierung durch den Client verhindert werden. Dafür wird der Konstruktor privat deklariert und somit kann sich das Singleton nur selbst instanziieren. Es wird ebenfalls eine getter-Methode angeboten, mit der diese einzigarte Singletoninstanz zurückgegeben wird. [16]

Vorteile:

- Eine Singletonklasse kann sehr schnell und unkompliziert implementiert werden.
- Der Zugriff auf das Singleton kann durch die Kapselung genau kontrolliert werden.
- Man kann ein Singleton ebenfalls ableiten, wenn man ihm neue Funktionalitäten hinzufügen möchte. Dabei wird dynamisch zur Laufzeit entschieden, welche Unterklasse genutzt wird.

[16]

Nachteile:

- Die Verwendung von Singletons entspricht eher der prozeduralen Programmierung und hat nichts mit Objektorientierung zu tun.
- Das Singleton stellt seine Daten für die „gesamte Welt“ zur Verfügung. Dies könnte zu einer Gefährlichkeit werden.
- Ob eine bestimmte Klasse ein Singleton verwendet, wird erst aus der Implementierung klar. Diese wird hart an das Singleton gekoppelt.

[16]

Beim Singleton Pattern werden keine (mir bekannten) Design-Prinzipien umgesetzt.

Ein praktisches Beispiel, wo das Singleton Pattern in der Programmiersprache Java angewendet wird, ist auf folgender Seite zu finden: [25]

3.7 Command Pattern

Das Command Pattern wird folgendermaßen beschrieben:

„Das Command Pattern kapselt einen Auftrag als ein Objekt und ermöglicht es so, andere Objekte mit verschiedenen Aufträgen zu parametrisieren, Aufträge in Warteschlangen einzureihen oder zu protokollieren oder das Rückgängigmachen von Operationen zu unterstützen.“ ([4], Seite 206)

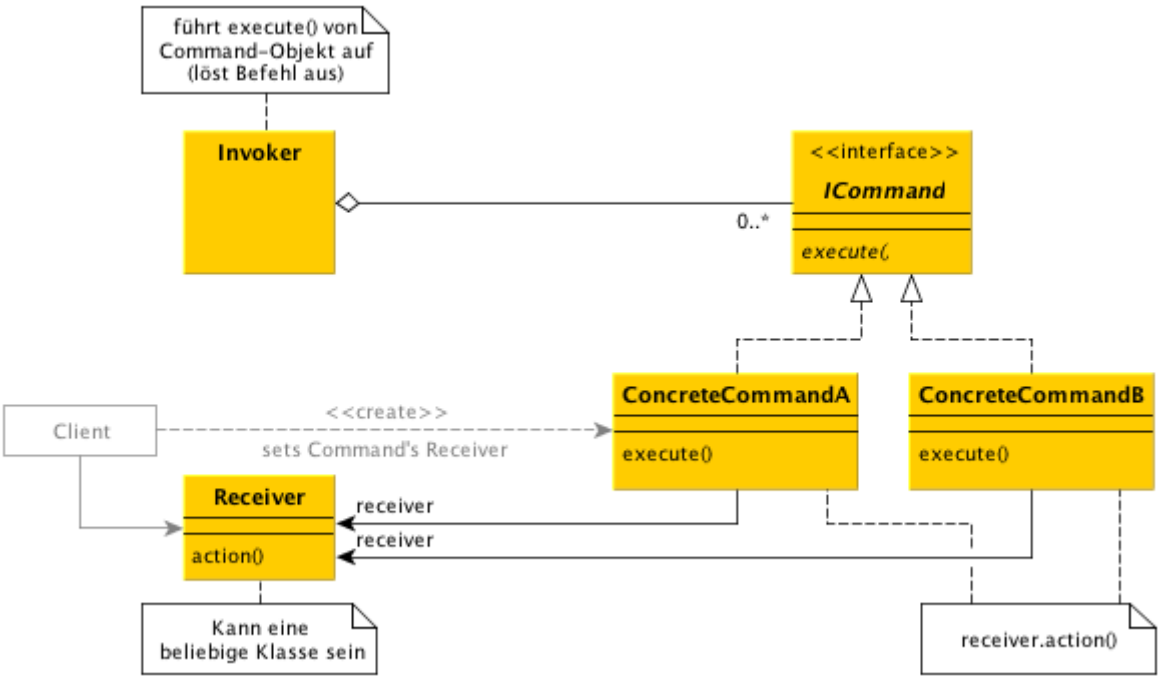

Abbildung 7: Command Pattern [Abb7]

Beim Command Pattern werden Befehle und Aufrufe modularisiert. Befehle können rückgängig gemacht, protokolliert oder in eine Warteschlange gelegt werden. Der Invoker und der Receiver sind davon entkoppelt. Der einzige, der weiß, welche Aktionen auf welchem Empfänger durchzuführen sind, ist das Command-Objekt. Der Invoker kennt den Receiver nicht, sondern nur das Command-Interface.

Der Invoker besitzt einen oder mehrere Commands und kann einen bestimmten Command über die execute()-Methode ausführen. Die konkrete Command-Klasse kennt seinen Receiver und ruft die action()-Methode auf. Damit wird der Aufruf des Invokers an den Receiver delegiert.

[17]

Vorteile:

- Die Commands können von verschiedenen Invokern wiederverwendet werden.
- Die Kohäsion jedes Teilnehmers (Invoker, Command und Receiver) wird durch die Reduzierung der Abhängigkeiten erhöht.
- Coderedundanzen und Inkonsistenzen werden vermieden.
- Commandobjekte können wie normale Objekte verwendet werden.

[17]

Nachteile:

- Da für jeden neuen Command eine neue Klasse erzeugt werden muss, kommt es leicht zur Unübersichtlichkeit.

[17]

Das Command Pattern setzt folgende Prinzipien um:

- Programmierung auf Schnittstellen statt auf Implementierungen (Punkt 2.3)
- Lose Kopplung (Punkt 2.5)
- Offen/Geschlossen-Prinzip (Punkt 2.6)

Ein praktisches Beispiel, wo das Command Pattern in der Programmiersprache PHP angewendet wird, ist auf folgender Seite zu finden: [26]

3.8 Adapter Pattern

Das Adapter Pattern wird folgendermaßen beschrieben:

„Das Adapter Pattern konvertiert die Schnittstelle einer Klasse in die Schnittstelle, die der Client erwartet. Adapter ermöglichen die Zusammenarbeit von Klassen, die ohne nicht zusammenarbeiten können, weil sie inkompatible Schnittstellen haben.“ ([4], Seite 243)

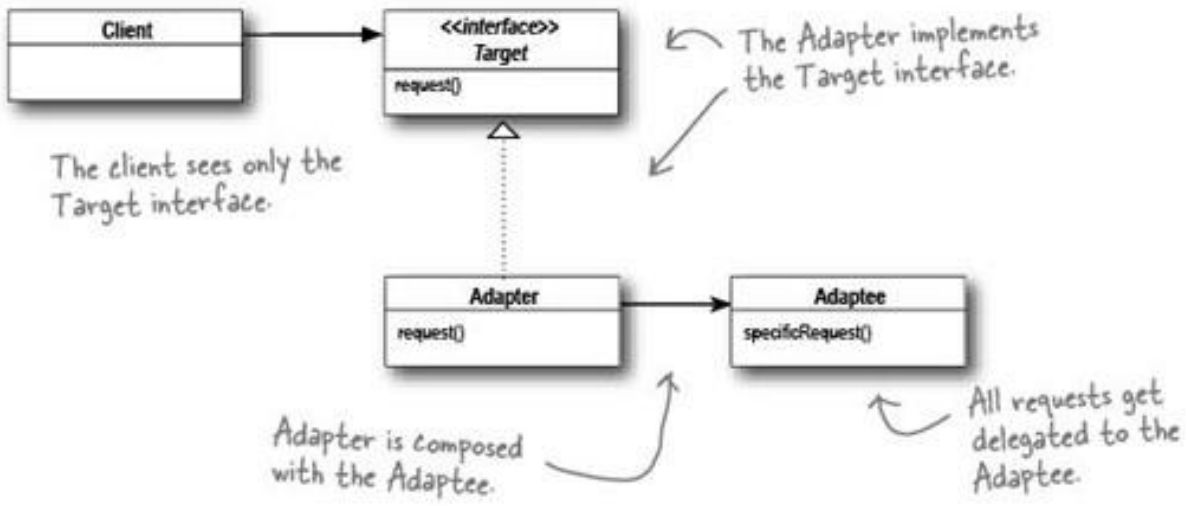

Abbildung 8: Adapter Pattern ([Abb2_8], Seite 243)

Beim Adapter Pattern arbeitet der Client mit Objekten, die von ihm eigentlich nicht verwendet werden können. Dabei wird ein Adapter-Objekt verwendet und mit diesem gearbeitet. Ein Adapter-Objekt implementiert das Adapter-Interface und delegiert die Aufrufe des Clients weiter. Bei der Initialisierung des Adapters wird das andere Objekt im Konstruktor angegeben.

([4], Seite 236 – 257)

Vorteile:

- Eine Adapter-Klasse passt sich nur einer Zielklasse an und überschreibt nur das Verhalten dieser Klasse.

([4], Seite 236 – 257)

Nachteile:

- Eine Adapter-Klasse kann nicht zur automatischen Anpassung von Unterklassen verwendet werden.
- Wenn viele neue Adapter-Klassen erstellt werden, kann es zur Unübersichtlichkeit kommen.

([4], Seite 236 – 257)

Das Adapter Pattern setzt folgende Prinzipien um:

- Programmierung auf Schnittstellen statt auf Implementierungen (Punkt 2.3)
- Lose Kopplung (Punkt 2.5)
- Prinzip der Verschwiegenheit (Punkt 2.8)

Ein praktisches Beispiel, wo das Adapter Pattern in der Programmiersprache Java angewendet wird, ist auf folgender Seite zu finden: [27]

3.9 Facade Pattern

Das Facade Pattern wird folgendermaßen beschrieben:

„Das Facade Pattern bietet eine vereinheitliche Schnittstelle für einen Satz von Schnittstellen eines Basissystems. Die Fassade definiert eine hochstufigere Schnittstelle, die die Verwendung des Basisystems vereinfacht.“ ([4], Seite 264)

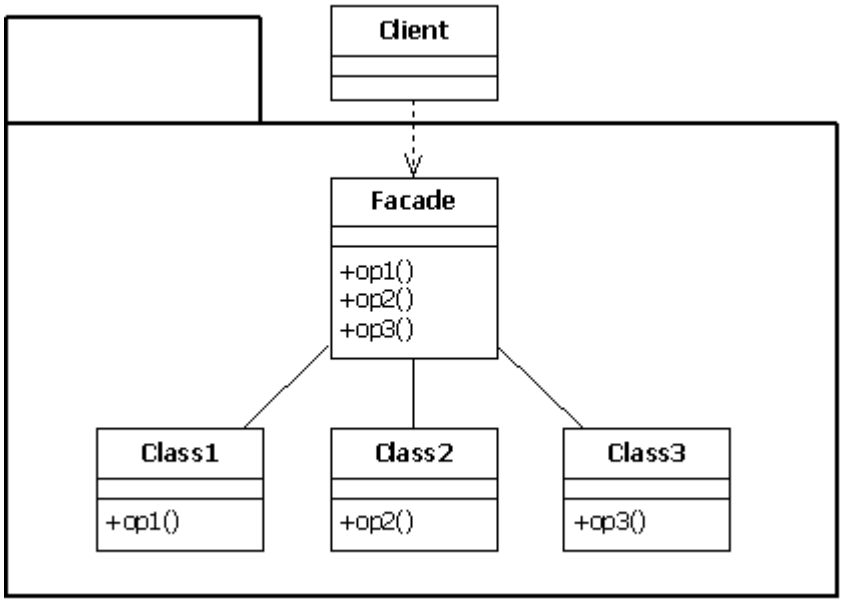

Abbildung 9: Facade Pattern [Abb9]

Beim Facade Pattern wird eine Schnittstelle definiert, um die Benutzung eines Moduls zu erleichtern. Dabei wird zwischen den Clients und dem Submodul geschaltet. Das Submodul wird gekapselt und die Facade besitzt die notwendige Logik zum Arbeiten mit dem Submodul. Für die Clients wird eine Schnittstelle nach außen angeboten. Die Aufrufe vom Client werden dann an das Submodul delegiert. Somit besitzt der Client kein Wissen über die Klassen, deren Beziehungen und Abhängigkeiten und kann das Modul trotzdem über die Facade nutzen. [18]

Vorteile:

- Durch die angebotene Schnittstelle kann der Client ein komplexes System einfach nutzen, da er die Klassen nicht kennen muss.
- Zwischen dem Client und den jeweiligen Subsystemen herrscht eine lose Kopplung. Modifikationen beim Subsystem bedeuten nur Änderungen innerhalb des gesamten Systems.

[18]

Nachteile:

- Die Schnittstelle muss bei bestimmten Änderungen in den Subsystemen modifiziert werden.

[18]

Das Facade Pattern setzt folgende Prinzipien um:

- Stark variablen Code von statischen Teilen trennen (Punkt 2.2)
- Lose Kopplung (Punkt 2.5)
- Offen/Geschlossen-Prinzip (Punkt 2.6)
- Prinzip der Verschwiegenheit (Punkt 2.8)

Ein praktisches Beispiel, wo das Facade Pattern in der Programmiersprache Java angewendet wird, ist auf folgender Seite zu finden: [28]

3.10 Composite Pattern

Das Composite Pattern wird folgendermaßen beschrieben:

„Das Composite Pattern ermöglicht es, Objekte zu einer Baumstruktur zusammenzusetzen, um Teil- bzw. Ganz-Hierarchien auszudrücken. Das Composite Pattern erlaubt den Clients, individuelle Objekte und Zusammensetzungen von Objekten auf gleiche Weise zu behandeln.“ ([4], Seite 356)

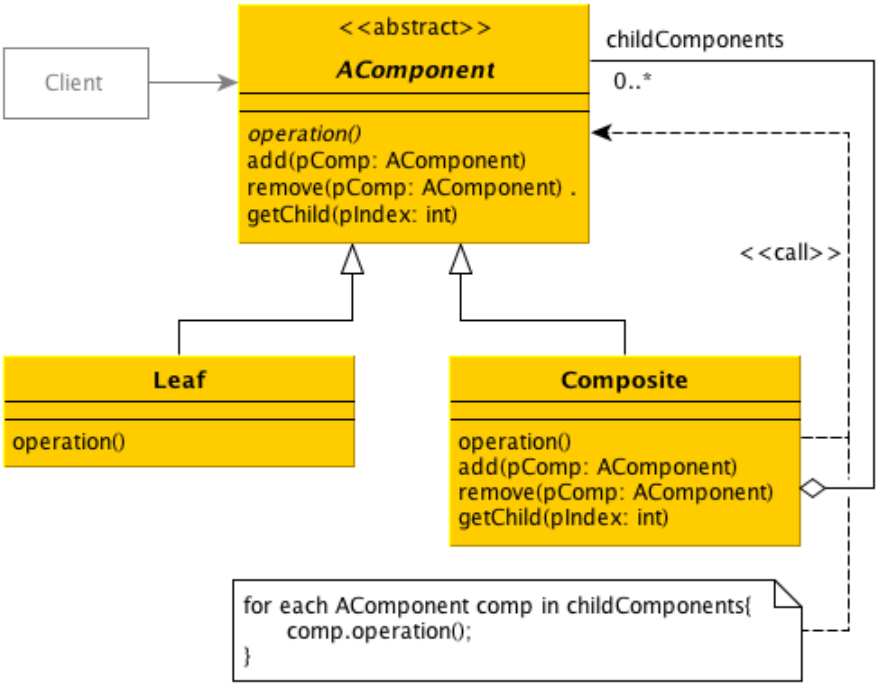

Abbildung 10: Composite Pattern [Abb10]

Beim Composite Pattern wird eine gemeinsame Component-Schnittstelle für die atomaren Elemente (Leaf, dt.: Blatt) und für die Elementbehälter (Composite, dt: Kompositum, Knoten) definiert. In der Component-Schnittstelle werden alle Methoden zur Verfügung gestellt, die gleichermaßen von den Leafs und den Composites angewandt werden. Aufrufe von den Composites werden an den Component delegiert, welcher ein atomares Leaf sein kann. Somit muss der Client nicht mehr zwischen Composite und Leaf unterscheiden. [19]

Vorteile:

- Es können Strukturen mit primitiven und zusammengesetzten Objekten implementiert werden. Die Verschachtelungstiefe und –breite ist daher unbegrenzt.
- Der Client muss nicht separate Funktionen für jedes einzelne Element erstellen. Durch die Component-Schnittstelle ist der Client von einer konkreten Implementierung entkoppelt und kann sich allein auf die Abstraktion stützen.
- Es sind keine Codeänderungen beim Hinzufügen neuer Elemente (Composites oder Leafs) notwendig.

[19]

Nachteile:

- Die allgemeine Component-Schnittstelle lässt sich nicht einfach definieren, da man manchmal nicht genau weiß, welche Methoden in der Schnittstelle und welche in einer Composite-Klasse definiert werden sollen.
- Möchte man Kindkomponenten, die ein Composite besitzen kann, einschränken, kann dies nicht einfach umgesetzt werden.

[19]

Das Composite Pattern setzt folgende Prinzipien um:

- Offen/Geschlossen-Prinzip (Punkt 2.6)
- Umkehrung der Abhängigkeiten (Punkt 2.7)

Ein praktisches Beispiel, wo das Composite Pattern in der Programmiersprache Java angewendet wird, ist auf folgender Seite zu finden: [29]

4. Quellenangaben

[1] Klaus Lipinski (2014). Entwurfsmuster [Online]. Available at: http://www.itwissen.info/definition/lexikon/Entwurfsmuster-design-pattern.html [zuletzt abgerufen am 10.12.2014]

[2] Luke Wroblewski (2009). Developing Design Principles [Online]. Available at: http://www.lukew.com/ff/entry.asp?854 [zuletzt abgerufen am 10.12.2014]

[3] Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates (2004). Head First Design Patterns [Online]. Download available at: https://www.ebooks-it.net/ebook/head-first-design-patterns [zuletzt abgerufen am 11.12.2014]

[4] Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates – dt. Übersetzung von Lars Schulten und Elke Buchholz (2006). Entwurfsmuster von Kopf bis Fuß [Online]. Download available at: http://books.google.de/books/about/Entwurfsmuster_von_Kopf_bis_Fu%C3%9F.html?hl=de&id=k4_7uAJrwkEC [zuletzt abgerufen am 11.12.2014]

[5] Bernhard Lahres, Gregor Rayman (2009). Objektorientierte Programmierung – Kap. 3.5 „Trennung der Schnittstelle von der Implementierung“ [Online]. Available at: http://openbook.galileo-press.de/oop/oop_kapitel_03_005.htm#mjf3cd04445959793a762a62c9b9ff6c75 [zuletzt abgerufen am 11.12.2014]

[6] Jarka Arnold, Aegidius Plüss (2004). Komposition und Vererbung [Online]. Available at: http://www.java-online.ch/gpanel/index.php?inhalt_mitte=lernprogramm/komposition.inc.php [zuletzt abgerufen am 11.12.2014]

[7] Bernhard Lahres, Gregor Rayman (2009). Objektorientierte Programmierung – Kap. 3.4 „Offen für Erweiterung, geschlossen für Änderung“ [Online]. Available at: http://openbook.galileo-press.de/oop/oop_kapitel_03_004.htm#mj887f0556cf1d83e4d56c5875b300fe87 [zuletzt abgerufen am 11.12.2014]

[8] Bernhard Lahres, Gregor Rayman (2009). Objektorientierte Programmierung – Kap. 3.6 „Umkehr der Abhängigkeiten“ [Online]. Available at: http://openbook.galileo-press.de/oop/oop_kapitel_03_006.htm#mjf155956d6de05131afd30434fd85e74f [zuletzt abgerufen am 11.12.2014]

[9] Bernhard Lahres, Gregor Rayman (2009). Objektorientierte Programmierung – Kap. 3.1 „Prinzip einer einzigen Verantwortung“ [Online]. Available at: http://openbook.galileo-press.de/oop/oop_kapitel_03_001.htm#mj3779726901d09a441b65ad53bd5975d9 [zuletzt abgerufen am 11.12.2014]

[10] Philipp Hauer (2009, 2010). Das Strategy Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/strategy.php [zuletzt abgerufen am 11.12.2014]

[11] Philipp Hauer (2009, 2010). Das Observer Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/observer.php [zuletzt abgerufen am 11.12.2014]

[12] OO-Design (2014). Observer Pattern – Push and pull communication methods [Online]. Available at: http://www.oodesign.com/observer-pattern.html [zuletzt abgerufen am 11.12.2014]

[13] Philipp Hauer (2009, 2010). Das Decorator Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/decorator.php [zuletzt abgerufen am 14.12.2014]

[14] Philipp Hauer (2009, 2010). Das Factory Method Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/factory-method.php [zuletzt abgerufen am 14.12.2014]

[15] Philipp Hauer (2009, 2010). Das Abstract Factory Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/abstract-factory.php [zuletzt abgerufen am 14.12.2014]

[16] Philipp Hauer (2009, 2010). Das Singleton Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/singleton.php [zuletzt abgerufen am 14.12.2014]

[17] Philipp Hauer (2009, 2010). Das Command Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/command.php [zuletzt abgerufen am 14.12.2014]

[18] Philipp Hauer (2009, 2010). Das Facade Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/facade.php [zuletzt abgerufen am 14.12.2014]

[19] Philipp Hauer (2009, 2010). Das Composite Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/composite.php [zuletzt abgerufen am 14.12.2014]

[20] Pankaj Kumar (2013). Strategy Design Pattern in Java – Example Tutorial [Online]. Available at: http://www.journaldev.com/1754/strategy-design-pattern-in-java-example-tutorial [zuletzt abgerufen am 14.12.2014]

[21] Rainer Friesen, Markus Stollenwerk, Daniel Valentin (2006). Observer Pattern [Online]. Available at: https://public.hochschule-trier.de/~rudolph/gdv/cg/node45.html [zuletzt abgerufen am 14.12.2014]

[22] Reinhard Schiedermeier (2012). Beispiel Decorator-Pattern [Online].
Available at: http://sol.cs.hm.edu/4129/html/461-beispieldecoratorpattern.xhtml
[zuletzt abgerufen am 14.12.2014]

[23] Rias A. Sherzad (2006). Factory Method Pattern in Java - Beispiel [Online].
Available at: http://www.theserverside.de/factory-method-pattern-in-java/
[zuletzt abgerufen am 14.12.2014]

[24] Alan Shalloway, James Trott (2002).
Book „Design Patterns Explained Simply“ – Abstract Factory in PHP [Online].
Available at: http://sourcemaking.com/design_patterns/abstract_factory/php/2
[zuletzt abgerufen am 14.12.2014]

[25] David Geary (2003). How-To: Simply Singleton [Online]. Available at:
http://www.javaworld.com/article/2073352/core-java/simply-singleton.html
[zuletzt abgerufen am 14.12.2014]

[26] Alan Shalloway, James Trott (2002).
Book „Design Patterns Explained Simply“ – Command in PHP [Online].
Available at: http://sourcemaking.com/design_patterns/command/php
[zuletzt abgerufen am 14.12.2014]

[27] Alan Shalloway, James Trott (2002).
Book „Design Patterns Explained Simply“ – Adapter in Java [Online].
Available at: http://sourcemaking.com/design_patterns/adapter/java/2
[zuletzt abgerufen am 14.12.2014]

[28] Alan Shalloway, James Trott (2002).
Book „Design Patterns Explained Simply“ – Facade in Java [Online].
Available at: http://sourcemaking.com/design_patterns/facade/java/1
[zuletzt abgerufen am 14.12.2014]

[29] OO-Design (2014). Composite Pattern – Shapes Example – Java Sourcecode [Online].
Available at: http://www.oodesign.com/composite-pattern-shapes-example-java-sourcecode.html [zuletzt abgerufen am 14.12.2014]

5. Abbildungsverzeichnis

[Abb1] Philipp Hauer (2009, 2010). Das Strategy Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/strategy.php [zuletzt abgerufen am 11.12.2014]

[Abb2_8] Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates (2004). Head First Design Patterns [Online]. Download available at: https://www.ebooks-it.net/ebook/head-first-design-patterns [zuletzt abgerufen am 11.12.2014]

[Abb3] Alexander Schatten, Markus Demolsky, Erik Gostischa-Franta (2013). Best-Practise Software Engineering [Online]. Available at: http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/images/decorator_simple.jpg [zuletzt abgerufen am 14.12.2014]

[Abb4] Philipp Hauer (2009, 2010). Das Factory Method Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/factory-method.php [zuletzt abgerufen am 14.12.2014]

[Abb5] K. Hong (2014). Abstract Factory Design Pattern [Online]. Available at: http://www.bogotobogo.com/DesignPatterns/images/abstfactorymethod/Abstract_Factory_design_pattern.png [zuletzt abgerufen am 14.12.2014]

[Abb6] K. Hong (2014). Singleton Design Pattern [Online]. Available at: http://www.bogotobogo.com/DesignPatterns/images/singleton/singleton.png [zuletzt abgerufen am 14.12.2014]

[Abb7] Philipp Hauer (2009, 2010). Das Command Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/command.php [zuletzt abgerufen am 14.12.2014]

[Abb9] Estilos (2014). Facade Model [Online]. Available at: http://www.ideal.inf.br/estilos/projeto/StarUML/modules/staruml-pattern/pattern%20repository/GoF/Facade/Facade_Model.gif [zuletzt abgerufen am 14.12.2014]

[Abb10] Philipp Hauer (2009, 2010). Das Composite Design Pattern [Online]. Available at: http://www.philipphauer.de/study/se/design-pattern/composite.php [zuletzt abgerufen am 14.12.2014]