Hochschule für angewandte Wissenschaften
Fachhochschule Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

Schwerpunktseminar

Design Patterns

Dmitrij Ritter

Eingereicht am: 17.12.2012

Inhaltsverzeichnis

1 Strategy

1.1 Kurzbeschreibung

Das *Strategy* [1] Entwurfsmuster ist ein einfaches objektbasiertes Verhaltensmuster.

> *"Strategie definiert eine Familie von Algorithmen, kapsele jeden einzelnen und mach sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren."* [GOF95]

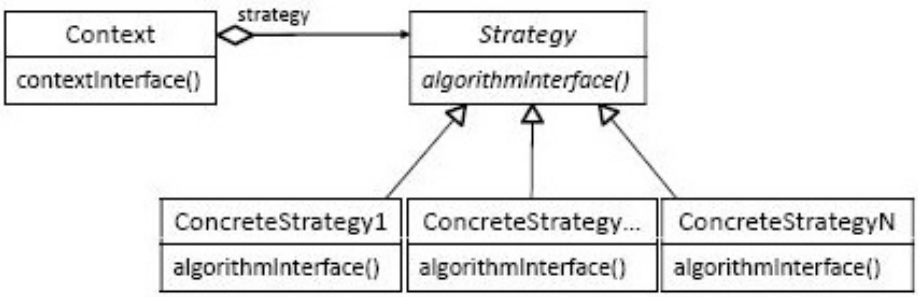


Abb. 1.1: Klassendiagramm nach [GOF95]

Mit diesem Muster wird also eine Klassenhierarchie von verhaltensähnlichen Algorithmen erstellt und jeden in einer Unterklasse vom Kontext abgekapselt.

Der *Context* bekommt von Schnittstelle Strategie nur eine Variable, die mit einer Referenz auf das gewünschte Strategieobjekt zeigt. So wird der konkrete Algorithmus über die Schnittstelle eingebunden und kann bei Bedarf selbst zur Laufzeit noch dynamisch gegen eine andere Implementierung ausgetauscht werden.

1.2 Szenario

Es sollen für einen Zoo verschiedene Affenarten modelliert werden. Jeder Affe soll schreien und klettern können.

Es liegt nah eine Vererbung zu verwenden: Eine abstrakte Superklasse Affe von der alle konkreten Subklassen (Gorilla, Pavian, Schimpanse) ableiten und damit das Schrei- und Kletterverhalten erben (oder gezwungen sind, schreien () und klettern () zu implementieren, wenn diese Methoden abstrakt deklariert sind.)

[1] Bekannt auch als *policy* oder *Strategie*

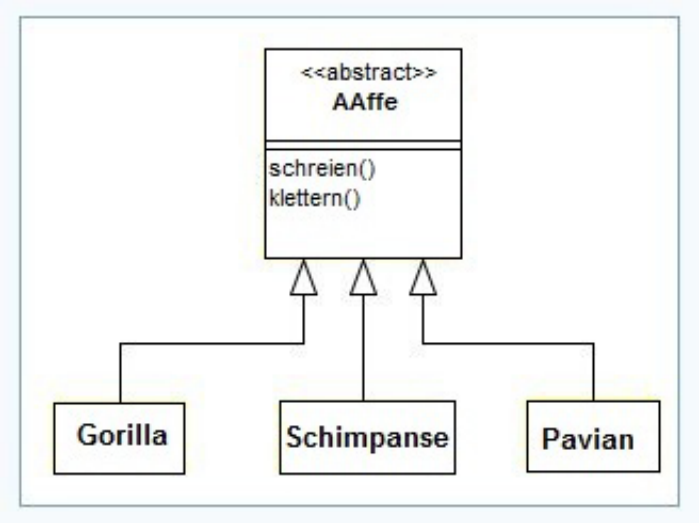


Abb. 1.2: Klassendiagramm für Beispiel

Soweit ist alles in Ordnung, allerdings ändern sich plötzlich die Anforderungen. Ein Gorilla soll nun anders Klettern als eine Schimpanse oder ein Pavian. Und ein Gorilla schreit definitiv anders als die anderen beiden Affenarten. Dazu müssen die Subklassen schreien () und klettern () überschreiben. Weiterhin soll auch eine weitere Klasse modelliert werden: Affenattrappen.

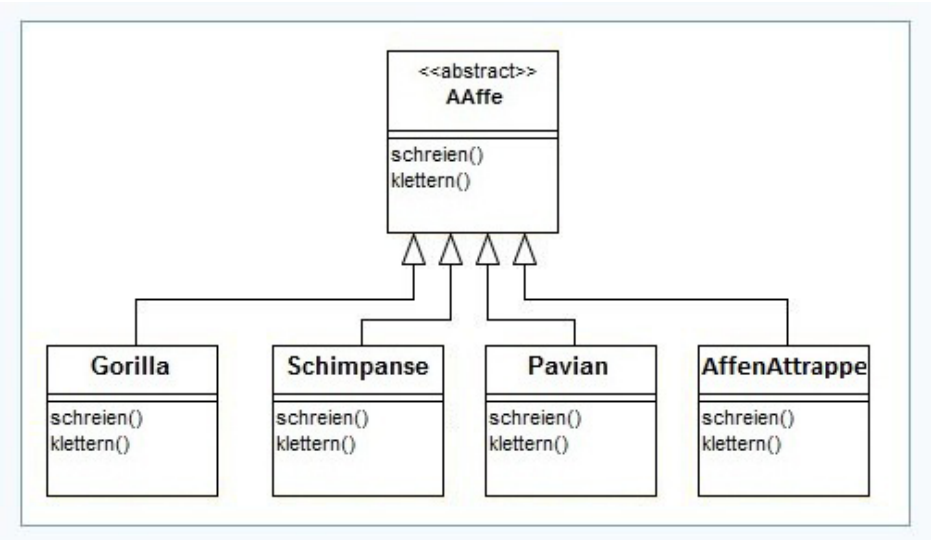


Abb. 1.3: Erweiterte Klassendiagramm für Beispiel

Es sieht gut aus, aber wenn man in Abbildung 1.3 genauer hinschaut, zeigen sich eine Reihe von Nachteilen bzw. Problemen:

- schreien () und klettern () werden immer vererbt (wenn in Class Affe bereits implementiert) bzw. müssen in der Subklasse implementiert werden (wenn abstrakt dekla-

riert), auch bei Subklassen, die beispielweise gar nicht klettern oder schreien können.

- *Code Redundanz*. Angenommen Gorilla und Pavian haben gleiches Kletterverhalten, so muss nichtsdestotrotz der klettern()-Code doppelt geschrieben werden. Dazu kommt, das Änderungen an diesem Verhalten, Codeänderungen in mehreren Klassen bedeutet. Ein Erweiterungs- und Wartungsalptraum mit hoher Fehleranfälligkeit.
- Außerdem ist es nicht möglich, das Verhalten der Affen zur Laufzeit zu ändern.
- Es können keine allgemeinen Aussagen über das Verhalten von Affen getroffen werden, da jeder konkrete Affe sie für sich selbst codiert.
- *Wiederverwendbarkeit*. Existierendes Verhalten kann nicht wiederverwendet werden: Erstellt man beispielsweise einen neuen Affenart, so muss man wieder schreien () und klettern () neu schreiben, unabhängig davon ob es sich dabei um neues oder bereits implementiertes Verhalten handelt.

Also der Entwurf aus der Sicht der Softwareentwicklung ist nicht so gelungen, da die Entwürfe so gemacht werden müssen, dass Änderungen minimale Auswirkungen auf den bestehenden Code haben, so genannte *Änderungsstabilität*.

1.3 Lösung

Zunächst müssen wir feststellen, dass bei unsere Modelierung nur das Verhalten schreien () und klettern () sich ändert. Der Rest der Affe-Klassen bleibt konstant.

Also liegt es nahe, den Verhaltenscode aus den Affe-Klassen herauszuziehen und in separaten Klassen zu kapseln.

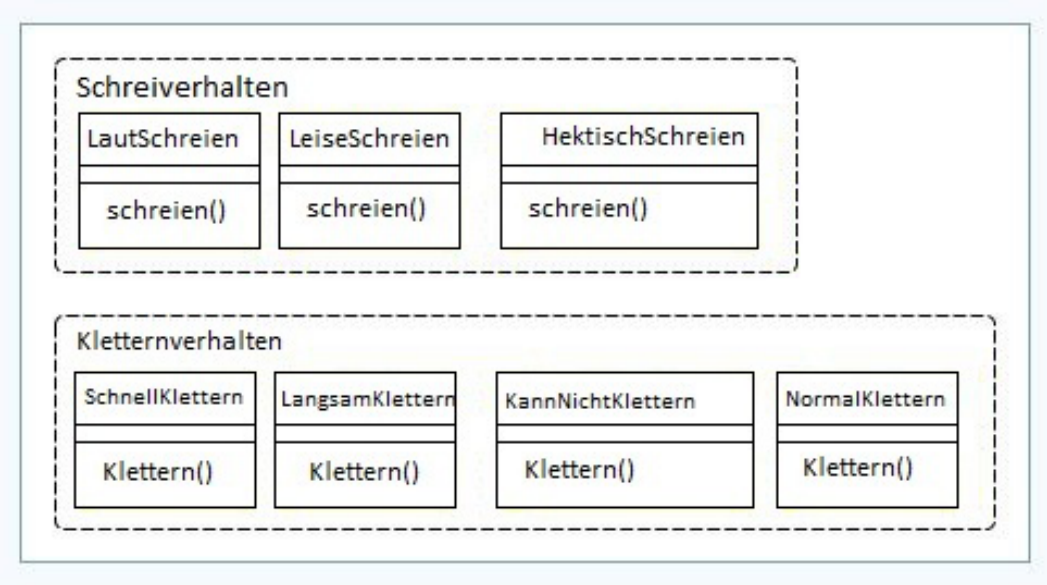


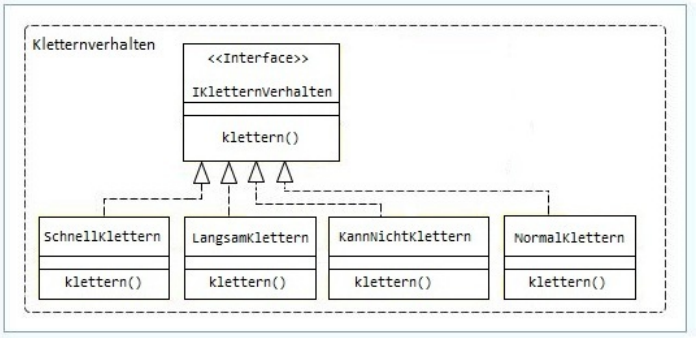
Abb. 1.4: Separate Kapselung von Verhaltenscode/Strategien

Dann ist es nur noch nötig, dass jeder Affe sein Verhaltensobjekt mit einer Instanzvariable kennt. Wenn der Gorilla klettern muss, so lässt er in Abbildung 1.5 sein Klettern-Verhaltensobjekt für ihn klettern.

```
1
2  abstract class Affe {
3      //Variablen vom Typ des Interfaces
4      IKletternVerhalten kletternVerhalten = new SchnellKlettern();
5
6      public void klettern(){
7          //Überträgt Verhalten an Verhaltensobjekt
8          kletternVerhalten.klettern();
9      }
10 }
```

Abb. 1.5: Delegation von Verhalten an das Verhaltensobjekt

In der Abbildung 1.6 definieren wir ein Interface für unser Verhalten und lassen die konkreten Verhaltensklassen dieses Interface implementieren.




```
1
2  interface IKletternVerhalten {
3      public void klettern();
4  }
5
6  class NormalKlettern implements IKletternVerhalten{
7      public void klettern() {
8          System.out.println("Normal klettern.");
9      }
10 }
11
12 class LangsamKlettern implements IKletternVerhalten{
13     public void klettern() {
14         System.out.println("Langsam klettern.");
15     }
16 }
17
18 class SchnellKlettern implements IKletternVerhalten {
19     public void klettern() {
20         System.out.println("Schnell klettern.");
21     }
22 }
23
24 class KannNichtKlettern implements IKletternVerhalten{
25     public void klettern() {
26         System.out.println("Kann doch gar nicht klettern.");
27     }
28 }
29
30
```

Abb. 1.6: Implementierung einer konkreten Strategy

Durch Polymorphie erlangen wir Flexibilität, denn der Gorilla weiß jetzt nicht mehr, was für Verhalten er besitzt. Er weiß nur, dass er damit schreien() bzw. klettern() kann. Und wir können nun durch entsprechende Setter oder Konstruktoren das Verhalten der Affe zur Design- und Laufzeit dynamisch setzen, ohne ihren Code anzufassen. Näher sieh Abbildung 1.7 und 1.8

Analyse:

- Keine Vererbung mit abgeleiteter abstrakter Affe-Klasse. Die Klasse entsteht durch die Erstellung neuer Verhaltensklassen oder durch die neue Kombination bestehender Verhaltensobjekte.
- Affe ist von seinem Verhalten entkoppelt und kennt nur noch die Schnittstelle seines Verhaltens. Dadurch wird das Verhalten eines Affes beliebig ändern, ohne seinen

```
1
2  public class Gorilla extends Affe {
3      //Default: LeiseSchreien
4      private ISchreiVerhalten schreiVerhalten = new LeiseSchreien();
5
6      public void schreien(){
7          schreiVerhalten.schreien();
8      }
9
10     public void setSchreiVerhalten(ISchreiVerhalten pSchreiVerhalten){
11         schreiVerhalten = pSchreiVerhalten;
12     }
13 }
14
15
```

Abb. 1.7: Fertiger Gorilla-Code mit gekapselten Schreiverhalten

```
1
2  public class Client {
3      public static void main(String[] args) {
4          Gorilla gorilla = new Gorilla();
5          gorilla.schreien(); //ganz leise Schrei...
6          gorilla.klettern(); //Schnelles klettern
7          gorilla.setKletternVerhalten(new LangsamKlettern()); //Verhalten dynamisch setzen
8          gorilla.klettern(); //Langsames klettern
9          //...
10     }
11 }
12
```

Abb. 1.8: Entkoppelte und variierende Verhalten

Code zu manipulieren.

- Keine Code Redundanz. Es gibt für jedes Verhalten eine Klasse. Jeder Affe, der dieses Verhalten benutzen will, kann eine entsprechende Instanz dieser Klasse nutzen. Dadurch wird doppelte Implementierung identischen Verhaltens in verschiedenen Klassen vermieden.
- Die Entwicklungszeit wird beschleunigt, weil eine Wiederverwendbarkeit des Verhaltens möglich ist.
- Das Verhalten eines Affes kann nur zur Designzeit und zur Laufzeit geändert werden.
- Es lassen sich sofort allgemeine Aussagen über die verschiedenen möglichen Verhalten von Affen treffen.
- Durch die Kapselung des Verhaltens können nun auch andere Klassen, außer Affen, das Verhalten wiederverwenden, beispielsweise wilde Katzen.

- Strategy Pattern ermöglicht erhöhte Flexibilität, Dynamik, erleichterte Wartung und Erweiterungen im Code reinzubringen

1.4 Implementierung

Das Beispiel in Abschnitt 1.3 ist sehr einfach gehalten, kann aber so implementiert werden.

Die Implementierung kann in angehefteten Anhang ”ExampleForStrategy”angeschaut werden.

Das vollständige UML-Diagramm wird in der Abbildung 1.9 gezeigt.

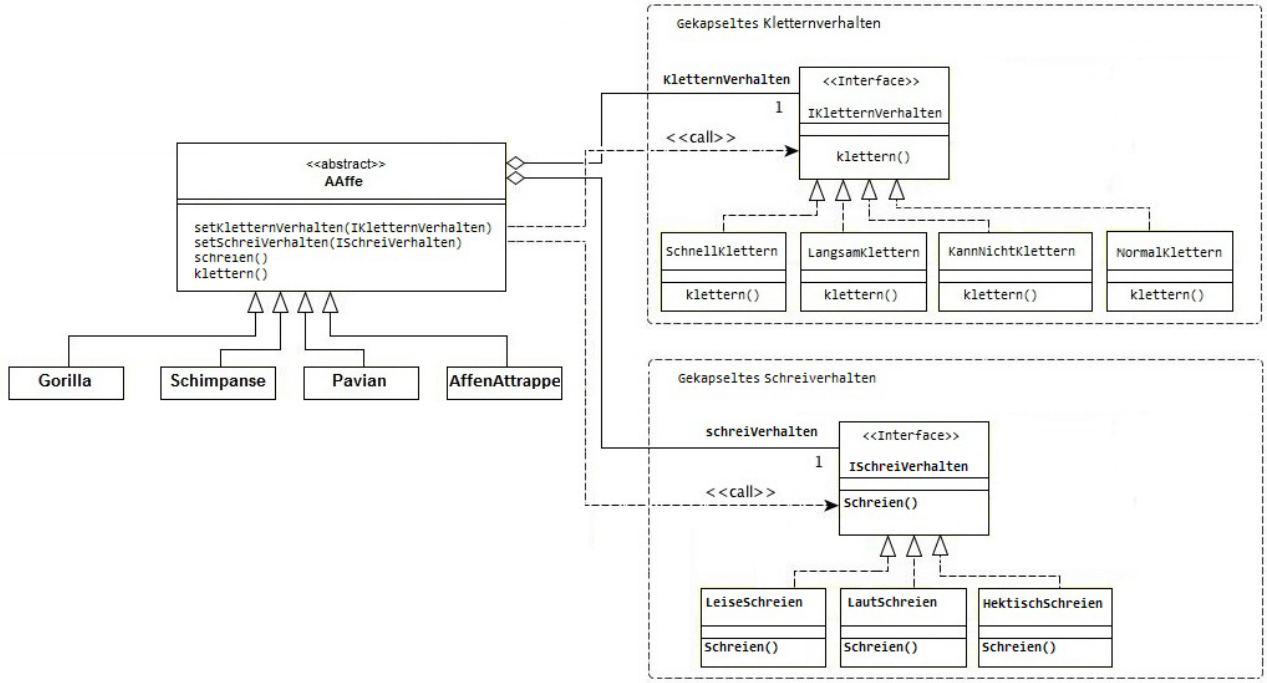


Abb. 1.9: Affenmodelierung nach Strategy Entwurfsmuster

1.5 Anwendungsfälle

- Bei Klassen, die sich nur im Verhalten unterscheiden, können verschiedenen Verhalten als Strategy-Klassen gekapselt werden.
- Verhalten sollte auch dann vom Context entkoppelt werden, wenn unterschiedliche Ausformungen ein und der selben Funktion benötigt werden.

  - Sortierung einer Collection (Array, List), wobei die konkreten Strategys verschiedene Sortierverfahren repräsentieren. [PK07]
  - Packer, der verschiedene Kompressionsalgorithmen unterstützt[WiKi12]
  - Kapselung verschiedener Speicherbelegungsstrategien [GOF95]

- dem Klienten Details eines Algorithmus verborgen bleiben sollen.
  - Die von den verschiedenen Sortierverfahren benötigten Datenstrukturen (zusätzliche Arrays und Zeiger) werden in der konkreten Strategy-Klasse verborgen.
- Wenn in einer Kontextklasse verschiedene Verhalten implementiert sind und die gewünschte mit zahlreichen Bedingungen (if ()... else if()... else ...) ausgewählt wird, kann das Strategy Pattern genutzt werden, um die Verhaltensweisen aus dem Kontext auszulagern.

1.6 Vorteile

- Es wird eine Familie von Algorithmen definiert.
- Es wird die Auswahl aus verschiedenen Implementierungen ermöglicht und dadurch erhöhen sich die Flexibilität und die Wiederverwendbarkeit.
- Es können Mehrfachverzweigungen vermieden werden und dies erhöht die Übersicht des Codes.
- Strategien bieten eine Alternative zur Unterklassenbildung der Kontexte.
- Wiederverwendbarkeit der einzelnen Algorithmen aus der Klassenhierarchie.
- Die Kapselung der Verhaltensweisen in einzelne Strategieklassen ermöglicht eine leichte Auswechslsung und Erweiterbarkeit der speziellen Algorithmen.
- Erleichterung beim Lesen und Modifizieren des Quelltextes.

1.7 Nachteile

- Klienten müssen die unterschiedlichen Strategien kennen.
- Zusätzlicher Kommunikationsaufwand zwischen Strategie und Kontext.
- Die Anzahl von Objekten wird erhöht.
- Im Falle, dass die Algorithmen sehr unterschiedlich in ihren Implementierungen sind und nicht alle Operationen der angebotenen Schnittstelle nutzen.

- Im Falle, dass der Kontext zu stark gekapselt ist und Parameter erzeugt, die vom aktuellen Algorithmus nicht benötigt werden.
- Durch die Verwendung von gekapselten Algorithmen in Unterklassen entstehen bei der Anwendung eine höhere Anzahl an Objekten die verwaltet werden müssen.

1.8 Implementierungsaspekte

Dieses Entwurfsmuster verfolgt, wie viele andere Muster auch, das Open-Closed Prinzip und schütz somit vor Modifikationen und unterstützt Modulerweiterungen.

> *"Modules should be both open (for extension) and closed (for modification)"*
> *Bertrand Meyer* [MEY88]

"Module sollten sowohl offen (für Erweiterungen), als auch geschlossen (für Modifikationen) sein"

Dieses Prinzip sollte beim Entwurf beachtet werden!

1.9 Strategy vs. State

Das State und das Strategy Muster sind fast gleich und haben strukturgleiche Klassendiagramme, aber verfolgen unterschiedliche Ziele. Das Strategy Entwurfsmuster dient dazu, dass ein Objekt die Ausführung eines Algorithmus an ein konkretes Strategieobjekt delegiert, was den auszuführenden Algorithmus bestimmt. Das State Entwurfsmuster hingegen, ermöglicht es einem Objekt auf einfache Art und Weise zwischen konkreten Zustandsobjekten zu variieren.

Abbildungsverzeichnis

Literaturverzeichnis

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Design Patterns - Elements of Reusable Object-Oriented Software Addison-Wesley, 1995

[MEY88] Bertrand Meyer Object-Oriented Software Construction Prentice Hall, 1988

[PK07] Karl Eilebrecht, Gernot Starke:”Patterns kompakt”. Spektrum Verlag. 2. Auflage 2007.

[WiKi12] Wikipedia: ”Strategie (Entwurfsmuster)”.[Stand: 14.12.12]

[VKBF] E.Freeman, Entwurfsmuster von Kopf bis Fuß, O‘Reilly, 2006