The Strategy-Pattern

Ein Referat von Alexander Hafen,
im Rahmen der Software Design Patterns Vorlesung
April 2004

Inhaltsverzeichnis

1 Einleitung

Das Strategy Pattern ist ein **objektbasiertes Verhaltensmuster** und ist auch unter dem Namen **Policy** bekannt. Es beschreibt eine Möglichkeit, das Verhalten von Algorithmen zur Laufzeit zu bestimmen. Die Algorithmen sind in Klassen gekapselt und austauschbar. Die Strategy wird erst zur Laufzeit festgelegt, d.h. es wird zur Laufzeit eine der unterschiedlichen Strategy-Ableitungen instantiiert. Die Strategies sind von einer abstrakten Strategy-Klasse abgeleitet und unterscheiden sich nur in ihrem Verhalten.

2 Ohne Strategy

Wenn wir einen Klienten ohne Strategy-Pattern betrachten beinhaltet dieser in der Regel Kontext und Algorithmus zugleich. Wobei der Algorithmus den Kontext bearbeitet. Betrachten wir folgendes Beispiel:

```
int bruttoPreis = 100;
int nettoPreis = bruttoPreis*0.9;
```

Die Definition des Bruttopreises entspricht dem Kontext und die Formel für die Berechnung des Nettopreises entspricht dem Algorithmus. In der Praxis treten solche Konstellationen meist in Verbindung mit einer Bedingung auf. Betrachten sie bitte das zweite Beispiel, wenn nun der Klient für verschiedene Kunden jeweils einen anderen Berechnungsalgorithmus zur Verfügung stellen:

```
int bruttoPreis = 100;
if(stammKunde){
int nettoPreis = bruttoPreis*0.8; } // 20% Rabatt
if(neuKunde){
int nettoPreis = bruttoPreis*0.95;} // 5% Rabatt
```

3 Motivation

Gründe für den Einsatz des Strategy-Pattern ergeben sich nun wenn wir das zweite Beispiel Betrachten:

1. Unübersichtlichkeit und komplizierte Wartung des Quellcodes

   In diesem Beispiel sind die kundenspezifischen Algorithmen noch recht übersichtlich. Was ist aber, wenn wir weit über zwanzig Kundenprofile besitzen und deren Algorithmen weit komplizierter sind als eine einfache Prozentrechnung. Der Quellcode würde die Klasse schnell sprengen und der Programmierer verliert die Übersicht über alle implementierten Funktionen. Außerdem wäre es durchaus möglich, das der Algorithmus nicht nur an einer zentralen Stelle im Programmablauf aufgerufen wird. Wenn der kundenspezifische Algorithmus an verschiedenen Stellen des Programmes abgearbeitet wird, verliert man die Übersicht.

2. Ungenutzte Algorithmen sind unnötig.

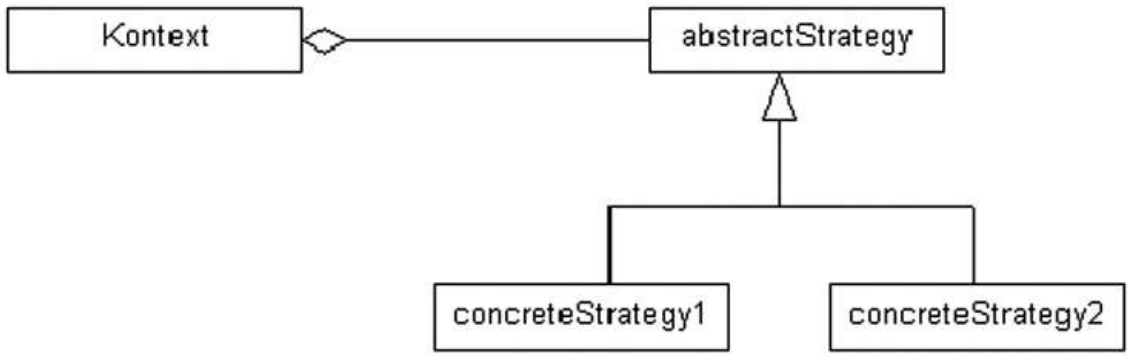
   Jeder Algorithmus der speziell einem Profil zugeordnet ist wird nur dann verwendet, wenn dieses Profil aktiv ist. Wenn wir nun auch zwanzig verschiedene Kunden Algorithmen zur Verfügung haben, wird doch immer nur einer Verwendet. Alle anderen 19 bleiben ungenutzt und verschwenden Speicher.

3. Schwer zu Erweitern

   Unter Berücksichtigung von Punkt 1 dürfte es auch klar sein, dass sich ein solcher Quellcode recht schwer erweitern lässt. Wenn sie nun Beispielsweise einen 21zigsten Algorithmus hinzufügen wollen, wird es nicht so einfach sein die Stellen im Code zu finden, an denen die Erweiterung hinzugefügt werden kann.

4 Struktur

Betrachten wir nun die Struktur des Strategy-Patterns:



– Kontext verwendet eine konkrete Strategy-Instanz

– Schnittstelle von Strategy muss mächtig genug sein, so dass alle erdenklichen Algorithmen abgedeckt sind

5 Ein Beispiel

Betrachten wir nun unser Preisberechnungstool mit den zwei Berechnungsalgorithmen für einen Stammkunden und einen Neukunden. Der Stammkunde erhält einen Rabatt von 20% und der Neukunden 5%.

Die Klasse Preisrechner bildet die abstrakte Strategie. Sie beschreibt den Algorithmus getPreis() aber implementiert keine Funktionalität:

```
public abstract class PreisRechner{
    public int getPreis(int bruttoPreis) {}
}
```

Nach der Definition der abstrakten Strategie werden wir eine konkrete Strategie für deinen Neukunden anlegen. Der Algorithmus soll einen Rabatt von 5% berechnen:

```
public class PreisRechner_Neukunde extends PreisRechner{
    public int getPreis(int bruttoPreis) {return 5;}
}
```

Als zweite konkrete Strategie definieren wir eine Klasse die den Algorithmus für den Stammkunden enthält. Sie berechnet einen Rabatt von 20%:

```
public class PreisRechner_Stammkunde extends PreisRechner
{
 public int getPreis(int bruttoPreis) {return 20;}
}
```

Zu guter letzt müssen wir nur noch den Klienten schreiben. Dieser enthält den Kontext und eine konkrete Strategie. Um die Rabattberechnung zu starten muss er den Kontext an die betreffende Methode der Strategie übergeben:

```
public class PreisTool{
    int bruttoPreis = 100;
    PreisRechner xxx = new PreisRechner_Neukunde()
    System.out.println( xxx.getPreis(bruttoPreis) );
}
```

6 Anwendung

Es wird einem sicherlich manchmal schwer fallen sich für das richtige Entwurfsmuster zu entscheiden. Darum gibt es ein paar Anhaltspunkte an denen sie sich orientieren können, ob das Strategie-Muster für ihr Projekt anwendbar ist, oder nicht:

Separation

Wenn ihre Klasse unterschiedliches Verhalten benötigt, das je nach Bedingung genutzt werden soll, ist dies ein Indiz für das Strategie Muster. Wie in unserem Beispiel aus Punkt 2 könnten viele IF-Abfragen auf Separation hinweisen.

Kombination

Es existieren innerhalb ihres Projektes viele Klassen die sich lediglich durch Ihr Verhalten unterscheiden. Der Kontext (Informationsgehalt) ist aber bei jeder Klasse gleich. Es liegt nahe, den gemeinsamen Kontext in einer einzigen Klasse anzulegen und die verschiedenen Algorithmen anhand des Strategie-Musters zu entwerfen.

Erweiterbarkeit

Sollten sie schon vor Beginn eines Projektes wissen, dass das Verhalten eines Objektes zukünftig durch ein anderes ersetzt oder erweitert werden soll, entscheiden sie sich besser gleich für das Strategie-Muster.

7 Nachteile

Strategien müssen bekannt sein

Ob dies nun ein Nachteil ist oder Nicht, auf jeden Fall müssen dem Klienten alle Informationen über vorhandene Strategien zugänglich gemacht werden. Ansonsten kann er die Strategien nicht instanzieren.

Umfangreiche Schnittstellen

In den Methodendefinitionen der einzelnen Strategien können die Argumente unter Umständen recht Komplex werden. So ist es durchaus möglich das ein Algorithmus mehr als 20 Parameter des Kontextes benötigt um seine Arbeit zu erledigen. Dies kann unter Umständen zu Problemen führen.

Unnötiger Kommunikationsaufwand

Wenn nun eine Strategie viele Parameter benötigt, müssen diese alle in der abstrakten Klasse definiert werden. Andere Strategien, die auf der selben Interface-Methode aufsetzen bekommen diese Parameter ebenfalls übergeben. Egal ob die Informationen benötigt werden oder nicht. Dies führt zu unnötigem Kommunikationsaufwand und verlangsamt das System.

Mögliche Lösung:

Um die beiden zuletzt beschriebenen Nachteile zu kompensieren, wäre es möglich anstatt der Parameter lediglich die Referenz auf das Kontext-Objekt zu übergeben. So ist es dem Algorithmus möglich, selbständig seine benötigten Informationen vom Kontext-Objet zu erfragen. Selbstverständlich muss dies vom Kontext-Objekt unterstützt werden. Auf diese Weise ist sichergestellt, dass nur die wirklich benötigten Daten übertragen werden und die Methodenköpfe nicht überladen

.

8 Referenzen

Patrones de diseño

http://vico.org/pages/PatronsDisseny.html
Pattern description by Erich Gamma & Co

Anwendung von Design Patterns

http://portal.dfpug.de/dFPUG/Dokumente/Konferenzen/VFP-Konferenz%202001/D-PATT.doc
by Nathalie Mengel, Wizards & Builders GmbH