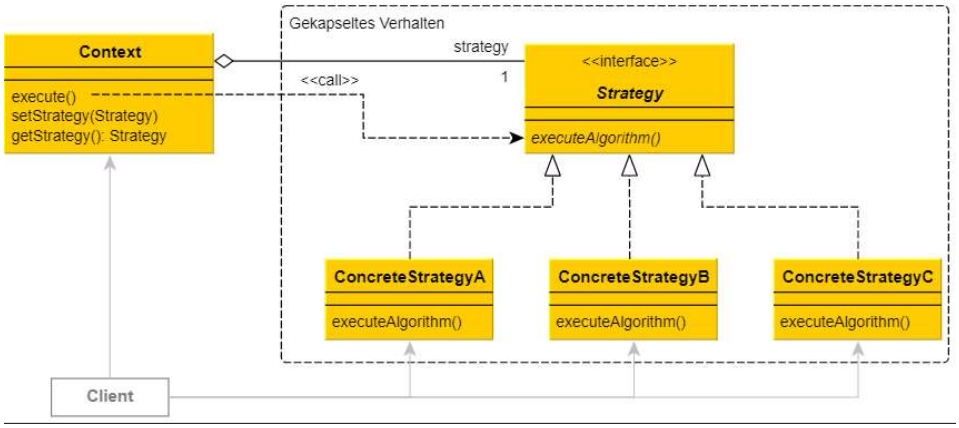
Merkblatt Strategy Design Pattern

Kurzbeschreibung / Zweck

Der Zweck des Strategy Pattern ist das Flexibilität und Dynamik gewonnen werden. Die Wartung soll erleichtert werden und Erweiterungen sollen schnell und unkompliziert umgesetzt werden. Man definiert eine Familie von Algorithmen, kapselt jeden einzelnen und macht die Algorithmen austauschbar. Das Strategy Pattern ermöglicht es das man den Algorithmus unabhängig vom nutzenden Klienten variieren kann.

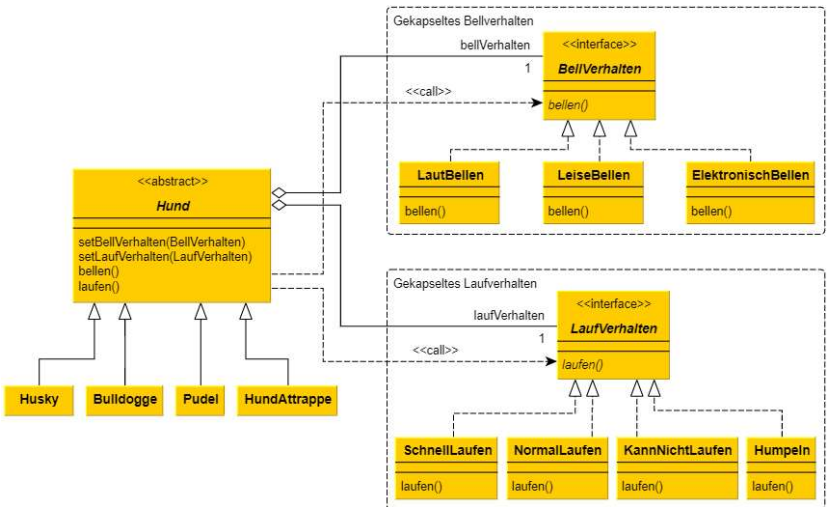
UML



Beschreibung

Die Ziele werden erreicht indem die Funktionalität eines Objekts in einen eigenen gekapselten Algorithmus ausgelagert wird. In eine sogenannte Strategieklasse.
Der Context hält eine Referenz auf sein Strategieobjekt und wenn er das ausgelagerte Verhalten ausführen soll, so delegiert er den Aufruf an sein referenziertes Strategieobjekt. Der Context arbeitet dabei nicht mit einer konkreten Implementierung, sondern mit einer Schnittstelle. Er ist damit implementierungsunabhängig und kann somit mit neuen Verhalten ausgestattet werden, ohne dass sein Code dafür geändert werden muss. Für ein neues Verhalten kann man ganz einfach ein neues Interface und neue Konkrete Strategien dazu erstellen. Einzige Bedingung ist, dass die neue Strategie das Strategieinterface korrekt implementiert.

Beispiel



In diesem Beispiel ist:
Hund=Context. BellVerhalten=Strategy-Interface . LautBellen, Leise Bellen = Konkrete Strategy, bellen()-Methode, laufen()-Methode = Algorithmen

Bellverhalten (Interface und Implementationen):

```
 interface BellVerhalten {
     public void bellen();
 }

 class LeiseBellen implements BellVerhalten {
     public void bellen() {
         System.out.println("ganz leise bellen...");
     }
 }

 class LautBellen implements BellVerhalten{
     public void bellen() {
         System.out.println("GANZ LAUT BELLEN!!");
     }
 }

 class ElektronischBellen implements BellVerhalten {
     public void bellen() {
         System.out.println("Elekkkkktronisch Bellen!");
     }
}
```

Laufverhalten (Interface und Implementationen):

```
interface LaufVerhalten {
    public void laufen();
}

class NormalLaufen implements LaufVerhalten{
    public void laufen() {
        System.out.println("Normal laufen.");
    }
}

class SchnellLaufen implements LaufVerhalten {
    public void laufen() {
        System.out.println("Schnell laufen.");
    }
}

class KannNichtLaufen implements LaufVerhalten{
    public void laufen() {
        System.out.println("Kann doch gar nicht laufen.");
    }
}

class Humpeln implements LaufVerhalten{
    public void laufen() {
        System.out.println("Humpeln.");
    }
}
```

Abstrakte Hundklasse und die konkrete Hundeklasse Husky

```
 public abstract class Hund {

     //Instanzvariablen vom Typ des Interfaces. Defaultverhalten
     BellVerhalten bellVerhalten = new LautBellen();
     LaufVerhalten laufVerhalten = new SchnellLaufen();

     public void setBellVerhalten(BellVerhalten bellVerhalten) {
         this.bellVerhalten = bellVerhalten;
     }

     public void setLaufVerhalten(LaufVerhalten laufVerhalten) {
         this.laufVerhalten = laufVerhalten;
     }

     public void bellen(){
         //Delegation des Verhaltens an Verhaltensobjekt
         bellVerhalten.bellen();
     }

     public void laufen(){
         //Delegation des Verhaltens an Verhaltensobjekt
         laufVerhalten.laufen();
     }
 }

 public class Husky extends Hund {
       public Husky(){
              setBellVerhalten(new LeiseBellen());
              setLaufVerhalten(new SchnellLaufen());
       }
}
```

Beispielclient:

```
 public class Client {
     public static void main(String[] args) {
         Husky husky = new Husky();
         husky.bellen(); //ganz leises bellen...
         husky.laufen(); //Schnelles laufen
         husky.setLaufVerhalten(new Humpeln());
         husky.laufen(); //Humpeln
         //...
     }
}
```