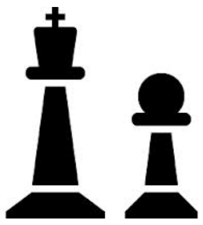
Strategy Pattern

acarlstein.com/

Posted by Alejandro G. Carlstein Ramos Mejia on July 18, 2013 July 22, 2013 About Programming / Design Patterns / Java

NOTIFICATION: These examples are provided for educational purposes. The use of this code and/or information is under your own responsibility and risk. The information and/or code is given 'as is'. I do not take responsibilities of how they are used.

Strategy pattern is the encapsulation of each algorithm in such a way that we can interchange them as best fit our needs.

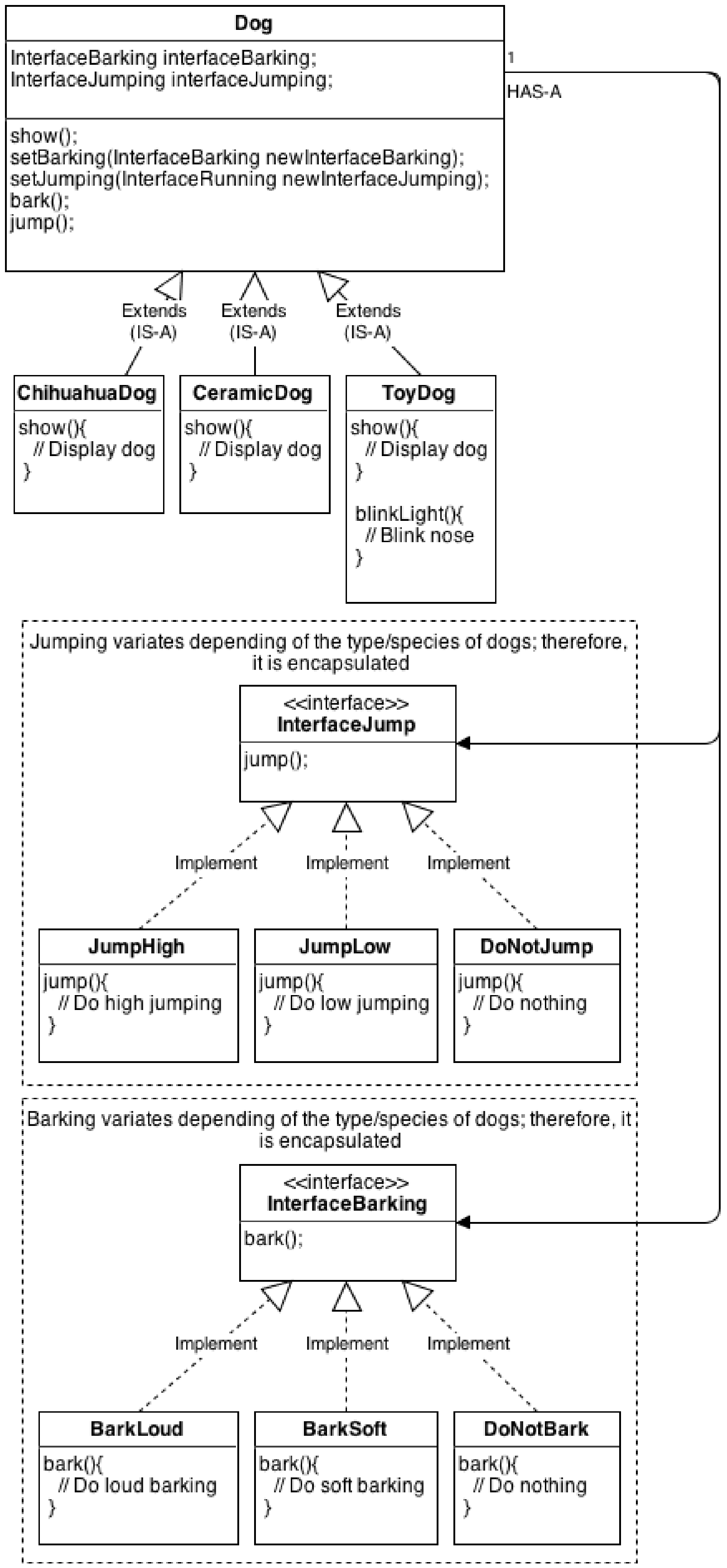


The main objective of this pattern is to apply in the best way possible Object Oriented principles such as abstraction, encapsulation, inheritance, and polymorphism using the following design principles:

- Identify and encapsulate the part of the problem that varies
- Do not program the implementation but the interface instead
- Reduce the number of dependencies between classes
- Avoid the use of concrete classes and depend on abstractions instead
- Each class should focus only on one task
- Super-classes should call sub-classes when needed
- Classes should be available for expansion. No for modification.
- Favor the use of loosely coupled design between objects

Normally, I would walk you step-by-step until you get to the big picture; however, we are going to do a different approach.
First I am going to show you the forest and then we are going to see each tree individually.

Lets assume that you company is a pet store which wishes to show online its products. As an experiment, the company decide to sell all type of dogs (such real dogs, ceramic dogs, and toy dogs) and different species of dogs (such as chihuahuas, great Danes and pugs).



Dog
InterfaceBarking interfaceBarking;
InterfaceJumping interfaceJumping;
show();
setBarking(InterfaceBarking newInterfaceBarking);
setJumping(InterfaceRunning newInterfaceJumping);
bark();
jump();
1
HAS-A
Extends (IS-A)
Extends (IS-A)
Extends (IS-A)
ChihuahuaDog
show(){ // Display dog }
CeramicDog
show(){ // Display dog }
ToyDog
show(){ // Display dog }
blinkLight(){ // Blink nose }
Jumping variates depending of the type/species of dogs; therefore, it is encapsulated
<<interface>> InterfaceJump
jump();
Implement
Implement
Implement
JumpHigh
jump(){ // Do high jumping }
JumpLow
jump(){ // Do low jumping }
DoNotJump
jump(){ // Do nothing }
Barking variates depending of the type/species of dogs; therefore, it is encapsulated
<<interface>> InterfaceBarking
bark();
Implement
Implement
Implement
BarkLoud
bark(){ // Do loud barking }
BarkSoft
bark(){ // Do soft barking }
DoNotBark
bark(){ // Do nothing }

The UML diagram displayed on the left, shows how the strategy pattern works. Please notice that we have the super-class called Dog. We could have called the super-class Animal and have a sub-class called Dog. Then we could create other sub-classes such as the Cat class; however, we are concentrating on Dogs to simplify the explanation.

Lets begin with the Dog super-class.This class have behaviors that all dogs may or not able to do. This super-class HAS-A interface for barking and jumping. The interfaces allow us to point to different algorithms. For example, this class have an interface called “interfaceJumping” which allow us to point to different jumping algorithms: the jump high, jump low and do not jump algorithm. By using the method setJumping(), we can tell this class which jumping algorithm will use when someone calls the method jump().

The three sub-classes below, ChihuahuaDog, CeramicDog and ToyDog inherit methods from the super-class, then you can define a different algorithm for each dog. For example, the chihuahua dog can jump low while the ceramic and toy dog cannot jump. While the chihuahua dog and the toy dog can bark, the ceramic dog cannot bark. Finally, you could add a Great Dane dog which jump higher and have a loud bark if you wish.

Please notice that we are using most of the design principles here. Some few examples are the fact that we are encapsulating those parts that can change such as different jump and barking behaviors. We are using loosely couple design since the classes have, or make use of, with little or no knowledge of the definitions of other separate classes. We have classes that focus only on one task in hand. We were concentrating on programming the interfaces instead of implementing everything in one class and modify the subclasses over and over again. By using composition, we can change the algorithm use for jumping in runtime. We could make the chihuahua jump higher instead of jumping lower depending of the kind of Chihuahua we wish to sell. We only need to pass the JumpHigh object to the ChihuahuaDog object as a parameter for setJumping() and it is done! Every time the jump() method is called, the ChihuahuaDog object will use the JumpHigh object making our Chihuahua jump higher instead of lower. Maybe our chihuahua got a broken leg that didn’t heal, then we can use the DoNotJump object for that particular chihuahua.

> In some programming languages such as JavaScript, the use of interfaces is not use; instead, duck typing, a style of dynamic typing, is being use. Duck typing allows polymorphism without the use of inheritance and concentrate in those aspects of an object rather of the type of the object. In this post, we are not going to go over duck typing.

How about we see the code for this pattern? 😁

Lets first create the interfaces:

InferfaceBarking.java

```
public interface InterfaceBarking {
 public void bark();
}
```

InterfaceJumping.java

```
public interface InterfaceJumping {
 public void jump();
}
```

Second, we create all the algorithms related with barking and jumping using the interfaces we created previously:

JumpHigh.java

```
public class JumpHigh implements InterfaceJumping {

 @Override
 public void jump() {
  System.out.println("I am jumping high!");
 }

}
```

JumpLow.java

```
public class JumpLow implements InterfaceJumping {

 @Override
 public void jump() {
  System.out.println("I am jumping low!");
 }

}
```

DoNotJump.java

```
public class DoNotJump implements InterfaceJumping {

 @Override
 public void jump() {
  System.out.println("I cannot jump!");
 }

}
```

BarkLoud.java

```
public class BarkLoud implements InterfaceBarking {

 @Override
 public void bark() {
  System.out.println("I am barking loud! BARK! BARK! BARK!");
 }

}
```

BarkSoft.java

```
public class BarkSoft implements InterfaceBarking {

 @Override
 public void bark() {
  System.out.println("I am barking soft. jip! jip! jip!");
 }

}
```

DoNotBark.java

```
public class DoNotBark implements InterfaceBarking {

 @Override
 public void bark() {
  System.out.println("(I cannot bark)");
 }

}
```

Third, we are going to create the super-class Dog (One Dog to rule them all, One Dog to find them, One Dog to bring them all and in the program bind them):

Dog.java

```
public abstract class Dog {
 // ASCII image done by JG
 private final String strDogImageASCII = "            |\\\n"
           +"   \\`-. _.._| \\\n"
           +"    |_,'  __`. \\\n"
           +"    (.\\ _/.| _  |\n"
           +"   ,'      __ \\ |\n"
           +" ,'     __/||\\  |\n"
           +"(O8O  ,/|||||/  |\n"
           +"   `-'_----    /\n"
           +"      /`-._.-'/\n"
           +"      `-.__.-'   \n";

 protected InterfaceBarking interfaceBarking;
 protected InterfaceJumping interfaceJumping;

 public Dog(){
  System.out.println("I am a dog\n" + strDogImageASCII);
 }

 public abstract void show();

 public void bark(){
  interfaceBarking.bark();
 }

 public void jump(){
  interfaceJumping.jump();
 }

 public void setBarking(InterfaceBarking newInterfaceBarking){
  this.interfaceBarking = newInterfaceBarking;
 }

 public void setJumping(InterfaceJumping newInterfaceJumping){
  this.interfaceJumping = newInterfaceJumping;
 }

}
```

Fourth, we are going to create the sub-classes ChihuahuaDog, CeramicDog and Toy Dog:

ChihuahuaDog.java

```
public class ChihuahuaDog extends Dog {

 public ChihuahuaDog(){
  System.out.println("I am a chihuahua dog!\n");
  interfaceBarking = new BarkSoft();
  interfaceJumping = new JumpLow();
 }

 @Override
 public void show() {
  System.out.println("Showing chihuahua dog!");
 }

}
```

CeramicDog.java

```
public class CeramicDog extends Dog {

 public CeramicDog(){
  System.out.println("I am a ceramic dog!\n");
  interfaceBarking = new DoNotBark();
  interfaceJumping = new DoNotJump();
 }

 @Override
 public void show() {
  System.out.println("Showing ceramic dog!");
 }

}
```

ToyDog.java

```
public class ToyDog extends Dog {

 public ToyDog(){
  System.out.println("I am a toy dog!\n");
  interfaceBarking = new BarkLoud();
  interfaceJumping = new DoNotJump();
 }

 @Override
 public void show() {
  // TODO Auto-generated method stub
  System.out.println("Showing toy dog!");
 }

}
```

Finally, we going to create the class DogProducts which would use all these work we have done so far:

DogProducts.java

```
import java.util.ArrayList;
import java.util.List;

public class DogProducts {

 /**
  * @param args
  */
 public static void main(String[] args) {

  // We are creating a list of dogs
  List<Dog> dogList = new ArrayList<Dog>();

  // Adding dogs to the list
  dogList.add(new ChihuahuaDog());
  dogList.add(new CeramicDog());
  dogList.add(new ToyDog());

  // For each dog in the list, show them and ask them to bark and jump.
  for (Dog dog : dogList) {
   dog.show();
   dog.bark();
   dog.jump();
   System.out.println();
  }
 }

}
```

The output would look as follow:

```
I am a dog
            |\
   \`-. _.._| \
    |_,'  __`. \
    (.\ _/.| _  |
   ,'      __ \ |
 ,'     __/||\  |
(O8O  ,/|||||/  |
   `-'_----    /
      /`-._.-'/
      `-.__.-'

I am a chihuahua dog!

I am a dog
            |\
   \`-. _.._| \
    |_,'  __`. \
    (.\ _/.| _  |
   ,'      __ \ |
 ,'     __/||\  |
(O8O  ,/|||||/  |
   `-'_----    /
      /`-._.-'/
      `-.__.-'

I am a ceramic dog!

I am a dog
            |\
   \`-. _.._| \
    |_,'  __`. \
    (.\ _/.| _  |
   ,'      __ \ |
 ,'     __/||\  |
(O8O  ,/|||||/  |
   `-'_----    /
      /`-._.-'/
      `-.__.-'

I am a toy dog!

Showing chihuahua dog!
I am barking soft. jip! jip! jip!
I am jumping low!

Showing ceramic dog!
(I cannot bark)
I cannot jump!

Showing toy dog!
I am barking loud! BARK! BARK! BARK!
I cannot jump!
```

Here is the file of this example: StrategyPatternDogProducts.zip

Now lets say that you wished to do a prank; therefore, we can use the flexibility that our code have. The prank is that each dog will have a random jump and bark.

Lets modify the main driver class:

DogProduct.java:

```
import java.util.ArrayList;
import java.util.List;

public class DogProducts {

 // We are creating a list of dogs
 private List<Dog> dogList;

 public DogProducts(){
  dogList  = new ArrayList<Dog>();
  addDogsToList();
 }
```

```
private void addDogsToList(){
 // Adding dogs to the list
 dogList.add(new ChihuahuaDog());
 dogList.add(new CeramicDog());
 dogList.add(new ToyDog());
}

private void show(){

 // For each dog in the list, show them and ask them to bark and jump.
 for (Dog dog : dogList) {
  dog.show();
  dog.bark();
  dog.jump();
  System.out.println();
 }
}

private void doPrank(){
 // Empty List
 dogList.clear();

 // Now the Chihuahua dog will bark loud and jump high LOL
 Dog chihuahuaDog = new ChihuahuaDog();
 chihuahuaDog.setBarking(new BarkLoud());
 chihuahuaDog.setJumping(new JumpHigh());

 // The ceramic dog is possess. It can bark soft and jump low. LOL
 Dog ceramicDog = new CeramicDog();
 ceramicDog.setBarking(new BarkSoft());
 ceramicDog.setJumping(new JumpLow());

 // The toy dog cannot bark but it can jump high. How unexpected! LOL
 Dog toyDog = new ToyDog();
 toyDog.setBarking(new DoNotBark());
 toyDog.setJumping(new JumpHigh());

 dogList.add(chihuahuaDog);
 dogList.add(ceramicDog);
 dogList.add(toyDog);
}

/**
 * @param args
 */
public static void main(String[] args) {

 DogProducts dogProducts = new DogProducts();

 boolean doPrank = true;
 if (doPrank){
  dogProducts.doPrank();
```

```
   dogProducts.show();
  }else{
   dogProducts.show();
  }

 }

}
```

Now the Output is:

```
I am a dog
            |\
   \`-. _.._| \
    |_,'  __`. \
    (.\ _/.| _  |
   ,'      __ \ |
 ,'     __/||\  |
(OBO  ,/|||||/  |
   `-'_----    /
      /`-._.-'/
      `-.__.-'

I am a chihuahua dog!

I am a dog
            |\
   \`-. _.._| \
    |_,'  __`. \
    (.\ _/.| _  |
   ,'      __ \ |
 ,'     __/||\  |
(OBO  ,/|||||/  |
   `-'_----    /
      /`-._.-'/
      `-.__.-'

I am a ceramic dog!

I am a dog
            |\
   \`-. _.._| \
    |_,'  __`. \
    (.\ _/.| _  |
   ,'      __ \ |
 ,'     __/||\  |
(OBO  ,/|||||/  |
   `-'_----    /
      /`-._.-'/
      `-.__.-'

I am a toy dog!
```

```
I am a dog
            |\
   \`-. _.._| \
    |_,'  __`. \
    (.\ _/.| _  |
   ,'      __ \ |
 ,'     __/||\  |
(O8O  ,/|||||/  |
   `-'_----    /
      /`-._.-'/
      `-.__.-'

I am a chihuahua dog!

I am a dog
            |\
   \`-. _.._| \
    |_,'  __`. \
    (.\ _/.| _  |
   ,'      __ \ |
 ,'     __/||\  |
(O8O  ,/|||||/  |
   `-'_----    /
      /`-._.-'/
      `-.__.-'

I am a ceramic dog!

I am a dog
            |\
   \`-. _.._| \
    |_,'  __`. \
    (.\ _/.| _  |
   ,'      __ \ |
 ,'     __/||\  |
(O8O  ,/|||||/  |
   `-'_----    /
      /`-._.-'/
      `-.__.-'

I am a toy dog!

Showing chihuahua dog!
I am barking loud! BARK! BARK! BARK!
I am jumping high!

Showing ceramic dog!
I am barking soft. jip! jip! jip!
I am jumping low!

Showing toy dog!
(I cannot bark)
I am jumping high!
```

Here is the source code: StrategyPatternDogProductsPrank.zip

At the end of this post, we wish to test the changes of the algorithm randomly in run time; therefore, we are going to modify DogProduct.java as follows:

DogProduct.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class DogProducts {

 // We are creating a list of dogs
 private List<Dog> dogList;

 public DogProducts(){
  dogList  = new ArrayList<Dog>();
  addDogsToListWithRandomAlgorithms();
 }

 public Random getNewGenerator(){
  Random seed = new Random();
  return new Random(seed.nextInt());
 }

 private void addDogsToListWithRandomAlgorithms(){

  Dog chihuahuaDog = new ChihuahuaDog();
  chihuahuaDog.setBarking(getRandomBark());
  chihuahuaDog.setJumping(getRandomJump());
  dogList.add(chihuahuaDog);

  Dog ceramicDog = new CeramicDog();
  ceramicDog.setBarking(getRandomBark());
  ceramicDog.setJumping(getRandomJump());
  dogList.add(ceramicDog);

  Dog toyDog = new ToyDog();
  toyDog.setBarking(getRandomBark());
  toyDog.setJumping(getRandomJump());
  dogList.add(toyDog);

 }

 private InterfaceBarking getRandomBark(){

  // Getting a new generator increases the chances of getting a true random number
  int randomBark = getNewGenerator().nextInt(2);

  switch(randomBark){
   case 0:
    return new BarkLoud();
   case 1:
    return new BarkSoft();
  }
  return new DoNotBark();
 }

 private InterfaceJumping getRandomJump(){

  // Getting a new generator increases the chances of getting a true random number
  int randomJump = getNewGenerator().nextInt(2);
  switch(randomJump){
   case 0:
    return new JumpHigh();
   case 1:
    return new JumpLow();
  }
  return new DoNotJump();
 }

 private void show(){

  // For each dog in the list, show them and ask them to bark and jump.
  for (Dog dog : dogList) {
   dog.show();
   dog.bark();
   dog.jump();
   System.out.println();
  }
 }

 /**
  * @param args
  */
 public static void main(String[] args) {

  DogProducts dogProducts = new DogProducts();
  dogProducts.show();

 }

}
```

The output is very random:

```
I am a dog
            |\
   \`-. _.._| \
    |_,'  __`. \
    (.\ _/.| _  |
   ,'      __ \ |
 ,'     __/||\  |
(O8O  ,/|||||/  |
   `-'_----    /
```

```
      /`-._.-'/
      `-.__.-'

I am a chihuahua dog!

I am a dog
            |\
   \`-. _.._| \
    |_,'  __`. \
    (.\ _/.| _  |
   ,'      __ \ |
 ,'     __/||\  |
(O8O  ,/|||||/  |
   `-'_----    /
      /`-._.-'/
      `-.__.-'

I am a ceramic dog!

I am a dog
            |\
   \`-. _.._| \
    |_,'  __`. \
    (.\ _/.| _  |
   ,'      __ \ |
 ,'     __/||\  |
(O8O  ,/|||||/  |
   `-'_----    /
      /`-._.-'/
      `-.__.-'

I am a toy dog!

Showing chihuahua dog!
I am barking soft. jip! jip! jip!
I am jumping high!

Showing ceramic dog!
I am barking loud! BARK! BARK! BARK!
I am jumping high!

Showing toy dog!
I am barking soft. jip! jip! jip!
I am jumping low!
```

Here is the source: StrategyPatternDogProductsRandom.zip
Please leave a comment if you have a question or you wish to provide your feedback.