

Strategy Pattern

Strategy

Motivation There are common situations when classes differ only in their behavior. For this cases is a good idea to isolate the algorithms in separate classes in order to have the ability to select different algorithms at runtime.

NEW Forum Discussions: NEW

What Design Pattern Should I Choose?

Design Principles and Patterns Strategy

Motivation There are common situations when classes differ only in their behavior. For this cases is a good idea to isolate the algorithms in separate classes in order to have the ability to select different algorithms at runtime.

Intent Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Implementation

Strategy - defines an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy - each concrete strategy implements an algorithm.

Context

- contains a reference to a strategy object.
- may define an interface that lets strategy accessing its data. The Context objects contains a reference to the ConcreteStrategy that should be used. When an operation is required then the algorithm is run from the strategy object. The Context is not aware of the strategy implementation. If necessary, addition objects can be defined to pass data from context object to strategy.

The context object receives requests from the client and delegates them to the strategy object. Usually the ConcreteStrategy is created by the client and passed to the context. From this point the clients interacts only with the context.

Applicability & Examples

Example - Robots Application

Let's consider an application used to simulate and study robots interaction. For the beginning a simple application is created to simulate an arena where robots are interacting. We have the following classes:

IBehaviour (Strategy) - an interface that defines the behavior of a robot

Concrete Strategies: AggressiveBehaviour, DefensiveBehaviour, NormalBehaviour; each of them defines a specific behavior. In order to decide the action this class needs information that is passed from robot sensors like position, close obstacles, etc.

Robot - The robot is the context class. It keeps or gets context information such as position, close obstacles, etc, and passes necessary information to the Strategy class.

In the main section of the application the several robots are created and several different behaviors are created. Each robot has a different behavior assigned: 'Big Robot' is an aggressive one and attacks any other robot found, 'George v.2.1' is really scared and run away in the opposite direction when it encounter another robot and 'R2' is pretty calm and ignore any other robot. At some point the behaviors are changed for each robot.

```
public interface IBehaviour {
    public int moveCommand();
}
```

```
public class AgressiveBehaviour implements IBehaviour{
    public int moveCommand()
    {
        System.out.println("\tAgressive Behaviour: if find another robot attack it");
        return 1;
    }
}
```

```
public class DefensiveBehaviour implements IBehaviour{
    public int moveCommand()
```

```
{
    System.out.println("\tDefensive Behaviour: if find another robot run from it");
    return -1;
}
}
```

```
public class NormalBehaviour implements IBehaviour{
    public int moveCommand()
    {
        System.out.println("\tNormal Behaviour: if find another robot ignore it");
        return 0;
    }
}
```

```
public class Robot {
    IBehaviour behaviour;
    String name;

    public Robot(String name)
    {
        this.name = name;
    }

    public void setBehaviour(IBehaviour behaviour)
    {
        this.behaviour = behaviour;
    }

    public IBehaviour getBehaviour()
    {
        return behaviour;
    }

    public void move()
    {
        System.out.println(this.name + ": Based on current position " +
            "the behaviour object decide the next move:");
        int command = behaviour.moveCommand();
        // ... send the command to mechanisms
        System.out.println("\tThe result returned by behaviour object " +
            "is sent to the movement mechanisms " +
            " for the robot " + this.name + "");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```
public class Main {

    public static void main(String[] args) {

        Robot r1 = new Robot("Big Robot");
        Robot r2 = new Robot("George v.2.1");
        Robot r3 = new Robot("R2");

        r1.setBehaviour(new AgressiveBehaviour());
        r2.setBehaviour(new DefensiveBehaviour());
        r3.setBehaviour(new NormalBehaviour());
    }
}
```

```

r1.move();
r2.move();
r3.move();

System.out.println("\n\nNew behaviours: " +
    "\n\n'tBig Robot' gets really scared" +
    "\n\n't, 'George v.2.1' becomes really mad because" +
    "it's always attacked by other robots" +
    "\n\n't and R2 keeps its calm\n\n");

r1.setBehaviour(new DefensiveBehaviour());
r2.setBehaviour(new AgressiveBehaviour());

r1.move();
r2.move();
r3.move();
}
}

```

Specific problems and implementation Passing data to/from Strategy object Usually each strategy need data from the context have to return some processed data to the context. This can be achieved in 2 ways.

- creating some additional classes to encapsulate the specific data.
- passing the context object itself to the strategy objects. The strategy object can set returning data directly in the context. When data should be passed the drawbacks of each method should be analyzed. For example, if some classes are created to encapsulate additional data, a special care should be paid to what fields are included in the classes. Maybe in the current implementation all required fields are added, but maybe in the future some new strategy concrete classes require data from context which are not include in additional classes. Another fact should be specified at this point: it's very likely that some of the strategy concrete classes will not use field passed to the in the additional classes.

On the other side, if the context object is passed to the strategy then we have a tighter coupling between strategy and context.

Families of related algorithms. The strategies can be defined as a hierarchy of classes offering the ability to extend and customize the existing algorithms from an application. At this point the composite design pattern can be used with a special care.

Optionally Concrete Strategy Objects It's possible to implement a context object that carries an implementation for default or a basic algorithm. While running it, it checks if it contains a strategy object. If not it will run the default code and that's it. If a strategy object is found, it is called instead (or in addition) of the default code. This is an elegant solution to exposing some customization points to be used only when they are required. Otherwise the clients don't have to deal with Strategy objects.

Strategy and Creational Patterns In the classic implementation of the pattern the client should be aware of the strategy concrete classes. In order to decouple the client class from strategy classes is possible to use a factory class inside the context object to create the strategy object to be used. By doing so the client has only to send a parameter (like a string) to the context asking to use a specific algorithm, being totally decoupled of strategy classes.

Strategy and Bridge Both of the patterns have the same UML diagram. But they differ in their intent since the strategy is related with the behavior and bridge is for structure. Further more, the coupling between the context and strategies is tighter that the coupling between the abstraction and implementation in the bring pattern.

Hot points The strategy design pattern splits the behavior (there are many behaviors) of a class from the class itself. This has some advantages, but the main draw back is that a client must understand how the Strategies differ. Since clients get exposed to implementation issues the strategy design pattern should be used only when the variation in behavior is relevant to them.