# Strategy Pattern

This design pattern is really good when we have a lot of if/else statements in our code and the problem is that code with if/else statements it’s not really easy for testing. Each condition leads to increasing the complexity of the code. If we need to test code with many if/else statements, there will be test for each case so it's not easy job. The benefit of strategy pattern is that we use one public method who is built via interface which leads to easier code for testing.

What about the scalability?

Let's image that we are writing framework or functionality which someone else will be using. To provide a good API for this user, he should be able to inject his own logic. For example, if we have a class which do some stuff, and then it must sort the data. So, everyone should be able to choose what sorting strategy to choose (plug in).

What can we do:

1. Define a class with the required functionality, and with a following property:

**private ISortStrategy strategy { get; set; }**

1. Everyone who wants to use your class needs to write a class which implements specific interface and pass it to the instance of your class. Example:  
   *//create your own strategy class* which implement ISortStrategy interface  
   **ISortStrategy myOwnStrategy = new WhateverStrategy();**

*//create an instance of the SorterClass and inject the strategy via the constructor*

**IUserSorter sorter = new UserSorter (myOwnStrategy);**

And instead of having this:

**If (type == 1) {  
 //tons of code  
} else if (type == 2) {**

**//even more code  
} else if (type == 3) {  
 //code again  
}**  
which by the way is hard to read + it may contains repeating code + it is hard for testing, you can just replace with:  
  
**IUserSorter sorter = new UserSorter (new SortActiveUsersWithActiveProgram());**

And within the class you can invoke the strategy like this -> **this.strategy.Sort(parameters);** which is a way better, cleaner and separated.

1. **Open/Closed Principle** applies really nice with this pattern. The code will be open for extension and closed for modification. The cool part is that we are not changing anything in **UserSorter** we just plug concrete implementation of the class
2. Once wrote this way the class doesn’t need modification anymore. If we want to add some change we can go directly into the strategy class.

Good example in .Net is **ICompare** interface.

This pattern affects (sort of) dependency injection which lets us to turn around flow of control of the modules which means that main modules don’t need to know about concrete implementation of some module from lower level.

Example:

We are making a game with tanks. You have a lot of tanks which attacks with different attack power and different speed. One of your options is to use inheritance and to derive from the base class but the problem is when we have tanks with the same speed but different attack power.

**Public class Tank**

**{  
 public IMovement moveStrategy { get; set; }  
 public IAttack attackStrategy { get; set; }  
  
 public Tank(IMovement newMoveStrategy)  
 {  
 this.moveStrategy = newMoveStrategy;  
 }  
}  
  
public class MoveSlow : IMovement   
{  
 public void Move() { //something }  
}  
  
public class MoveFast : IMovement  
{  
 public void Move() { //something else }  
}**

*// + similar implementation for the attacking as well.  
// In the same time in Main() we have ->* **Tank T62 = new Tank(new MoveSlow());  
Tank T55 = new Tank(new MoveFast());  
  
T62.Move(); T55.Move();**

Inheritance

What is wrong with the inheritance. Imagine that you have class Animal and half of the animals should be able fly, and the other half shouldn’t.

Bad solution:

**public string Fly () { return “I can fly” };**

**public string Fly () { return “I can’t fly” };**

Good solution:

We will have an **IFlyable canFly** property in the base class. So:   
 1. We won’t repeat ourselves in each deriver class. If we want to make a change, we will make it only in the base class.

2. If we add a new class, we will be able to inject a new strategy for it only without affecting any other class.  
  
And the final version:   
 **Animal dog = new Dog(new CantFly());  
 Animal tweety = new Bird(new CanFly());**

**Another interesting example**: <https://www.codeproject.com/Articles/1018930/Strategy-Design-Pattern-Explained-With-A-Real-Worl>

## Example 1 – Game

using System;

namespace Game

{

class Program

{

private const string MOVE\_COMMAND = "move";

private const string ATTACK\_COMMAND = "attack";

private const string INFO\_COMMAND = "info";

private const string QUIT\_COMMAND = "quit";

private const int METERS\_REQUIRED\_LEVEL\_2 = 5;

private const int METERS\_REQUIRED\_LEVEL\_3 = 12;

private const int FAR\_FAR\_AWAY = 25;

static void Main(string[] args)

{

string command = string.Empty;

int meters = 0;

Hero svetiGeorgi = new Hero(new SmallSword(), new Walking());

Console.WriteLine("Hello Goshe");

Console.WriteLine(string.Format("Second level: after {0} meters", METERS\_REQUIRED\_LEVEL\_2));

Console.WriteLine(string.Format("Third level: after {0} meters", METERS\_REQUIRED\_LEVEL\_3));

Console.WriteLine(string.Format("God level: after {0} meters", FAR\_FAR\_AWAY));

while (!command.Equals(QUIT\_COMMAND))

{

command = Console.ReadLine();

switch (command)

{

case MOVE\_COMMAND:

meters += svetiGeorgi.Move();

ChooseGear(meters, svetiGeorgi);

Console.WriteLine(string.Format("Current distance: {0} meters", meters));

break;

case ATTACK\_COMMAND:

svetiGeorgi.Attack();

break;

case INFO\_COMMAND:

Console.WriteLine(svetiGeorgi.Preview());

break;

}

}

Console.WriteLine("The game is over.");

Console.ReadLine();

}

private static void ChooseGear(int meters, Hero player)

{

if (METERS\_REQUIRED\_LEVEL\_2 <= meters && meters < METERS\_REQUIRED\_LEVEL\_3 && player.Level == 1)

{

player.LevelUp(new BigSword(), new Teleporting());

}

else if (meters >= METERS\_REQUIRED\_LEVEL\_3 && player.Level == 2)

{

player.LevelUp(new Bow(), new Flying());

}

else if (meters >= FAR\_FAR\_AWAY && player.Level == 3)

{

player.UseWeapon(new Gun());

}

}

}

public class Hero

{

public Hero(Weapon startUpWeapon, Movement startUpMovement)

{

this.Level = 1;

this.Weapon = startUpWeapon;

this.Movement = startUpMovement;

this.Rank = "Fisherman";

}

public string Rank { get; private set; }

public int Level { get; private set; }

public Weapon Weapon { get; private set; }

public Movement Movement { get; private set; }

public void UseWeapon(Weapon newWeapon)

{

if (newWeapon != null)

{

this.Weapon = newWeapon;

}

}

public void SetMovement(Movement newMovement)

{

if (newMovement != null)

{

this.Movement = newMovement;

}

}

public void LevelUp()

{

this.LevelUp(null, null);

}

public void LevelUp(Movement newMovement)

{

this.LevelUp(null, newMovement);

}

public void LevelUp(Weapon newWeapon)

{

this.LevelUp(newWeapon, null);

}

public void LevelUp(Weapon newWeapon, Movement newMovement)

{

this.Level++;

this.UseWeapon(newWeapon);

this.SetMovement(newMovement);

if (Level == 2)

{

this.Rank = "Hunter";

}

else if (Level == 3)

{

this.Rank = "Dragon Slayer";

}

}

public string Preview()

{

return string.Format(

"Rank: {0}, Weapon: {1}, Moving: {2}",

this.Rank,

this.Weapon.Type,

this.Movement.Type

);

}

public int Attack()

{

Console.WriteLine(string.Format(

"Rank: {0} ; Attack with {1} damage via {2}", this.Rank, this.Weapon.Damage, this.Weapon.Type));

return this.Weapon.Damage;

}

public int Move()

{

Console.WriteLine(string.Format(

"Rank: {0} ; Moved {1} meters via {2}", this.Rank, this.Movement.Distance, this.Movement.Type));

return this.Movement.Distance;

}

}

public interface Weapon

{

string Type { get; }

int Damage { get; }

int Range { get; }

bool IsOneHandWeapon { get; }

}

public class SmallSword : Weapon

{

public string Type => "Small Sword";

public int Damage => 1;

public int Range => 2;

public bool IsOneHandWeapon => true;

}

public class BigSword : Weapon

{

public string Type => "Big Sword";

public int Damage => 2;

public int Range => 3;

public bool IsOneHandWeapon => true;

}

public class Bow : Weapon

{

public string Type => "Bow";

public int Damage => 3;

public int Range => 4;

public bool IsOneHandWeapon => false;

}

public class Gun : Weapon

{

public string Type => "Gun";

public int Damage => 20;

public int Range => 20;

public bool IsOneHandWeapon => false;

}

public interface Movement

{

string Type { get; }

int Distance { get; }

}

public class Walking : Movement

{

public string Type => "Walking";

public int Distance => 2;

}

public class Teleporting : Movement

{

public string Type => "Teleporting";

public int Distance => 50;

}

public class Running : Movement

{

public string Type => "Running";

public int Distance => 4;

}

public class Flying : Movement

{

public string Type => "Flying";

public int Distance => 10;

}

}

## Example 2 – Click Strategy (Typescript)

export interface ClickStrategy {

onClick(e: any): void;

}

// Could be Iframe, Body, SomethingElse

export interface WorkingArea {

showPopover(e: any): void;

thereIsSelectedText(): boolean;

// and every single method related to the working area

}

// Could be click on image, dropdown, text, link, navigation, etc... Could save in DB, could send requests.

export interface ClickHandler {

handle(e: any): void;

}

export interface DomTranslator {

isLink(el: any): boolean;

isImage(el: any): boolean;

}

export class ClickInEditMode implements ClickStrategy {

constructor(

private area: WorkingArea,

private domTranslator: DomTranslator,

private onImageClick: ClickHandler,

private onLinkClick: ClickHandler) { }

// private properties for edit mode

onClick(e: any): void {

if (this.domTranslator.isLink(e)) {

this.onLinkClick.handle(e);

this.area.showPopover("link");

} else if (this.domTranslator.isImage(e)) {

this.onImageClick.handle(e);

this.area.showPopover("image");

} else if (this.area.thereIsSelectedText()) {

this.area.showPopover("text-editor");

}

}

}

export class ClickInRealMode implements ClickStrategy {

constructor(

private area: WorkingArea,

private domTranslator: DomTranslator,

private onHiperLinkClick: ClickHandler,

private onTabClick: ClickHandler) { }

// private properties for edit mode

onClick(e: any): void {

if (this.domTranslator.isLink(e)) {

this.onHiperLinkClick.handle(e);

this.area.showPopover("hiper-link");

} else if (this.isTab(e)) {

this.onTabClick.handle(e);

this.area.showPopover("tab");

}

}

private isTab(e: any): boolean {

// logic

return true;

}

}

export class EditorComponent {

private isEditMode;

private clickStrategy: ClickStrategy;

constructor() {

this.isEditMode = false;

this.chooseClickStrategy(this.isEditMode);

}

switchMode() {

this.isEditMode = !this.isEditMode;

this.chooseClickStrategy(this.isEditMode);

}

onClick(e) {

this.clickStrategy.onClick(e);

}

chooseClickStrategy(isEditMode) {

this.clickStrategy = ClickStrategiesPool.for(isEditMode);

}

}

// <body (click)="onClick($event)">

// <div (click="switchMode()")>

// <span> EDIT MODE IS </span>

// <span \*ngIf="isEditMode">ON</span>

// <span \*ngIf="!isEditMode">OFF</span>

// </div>

// html.......

// </body>

export class ClickStrategiesPool {

public static for(isEditMode) {

return isEditMode

? this.editModeStrategy()

: this.realModeStrategy();

}

private static editModeStrategy(): ClickStrategy {

return new ClickInEditMode(

new IFrameWorkingArea(),

new DomTranslator(),

new ImageClickHandler(),

new LinkClickHandler()

);

}

private static realModeStrategy(): ClickStrategy {

return new ClickInRealMode(

new IFrameWorkingArea(),

new DomTranslator(),

new HiperLinkClickHandler(),

new BootstrapTabClickHandler()

);

}

}