### **Ejercicio 1:**

Parte 1:

En este caso podemos ver que es un claro uso del antipatrón “Blob” el cual se trata de tener como una “super clase” la cual es la responsable de todo, por lo cual termina siendo de una longitud considerable además del hecho de que complica varios aspectos de la programación como puede ser el mantenimiento y la extensión del código.

En extra a la gran cosa, podemos ver la clase se llama “TODOcontroller” haciendo alusión a esto mismo, que tiene todas demasiadas responsabilidades

Parte 2

Para solucionar este dilema deberemos aplicar diferentes patrones para principalmente reducir la carga de responsabilidades que sufre esta clase, para lo cual aplicaremos el principio SRP el cual avala que las clases deberían tener una única razón para cambiar.

Otro patrón que podemos aplicar el LSV para lograr cumplir la funcionalidad que se quería obtener con la clase que cumplía con el antipatrón.

public abstract class Controller{

execute();

}

public class AddController extends Controller{

public AddController(){

//Constructor

}

public void add(){

//Add method

}

}

public class DeleteController extends Controller{

public AddController(){

//Constructor

}

public void delete(){

//Delete method

}

}

public class UpdateController extends Controller{

public AddController(){

//Constructor

}

public void add(){

//Update method

}

}

public class GeneralController {

public GeneralController(){

//Constructor

}

public void execute(controller: Controller){

//Execute controller

}

}

### **Ejercicio 2**

Parte 1

En este caso vemos que en todos los métodos siempre existe una verificación, por lo que podemos concluir que en todos estos método tiene este código repetido, por lo que se hace presente el antipatrón “Cut−and−Paste Programming”.

Parte 2

Por lo que una solución muy optimo seria crear una clase que se encargue de la verificación y dentro de cada uno de los métodos de la clase se llame al método de la clase verificadora.

public class verifier{

verify (string name, string email, string password): boolean{

//Verefy something

}

}

public class UserManager{

private verifier: Verifier;

public UserManager(){

verifier = new Verifier();

}

public void CreateUser (string name, string email, string password){

verifier.verify(name, email, password);

//Create user

}

public void DeleteUser (string *name*, string email, string password){

verifier.verify(*name*, email, password);

//Delete user

}

public void UpdateUser (string *name*, string email, string password){

verifier.verify(*name*, email, password);

// Update user

}

public void ChangePassword (string *name*, string email, string password){

verifier.verify(*name*, email, password);

//Change password

}

}

### **Ejercicio 3**

Parte 1:

En este caso se presenta el caso del antipatrón “Spaghetti Code” ya que se presentan varias características de este estilo de código, una característica es la gran longitud y en la ejecución del mismo este salta a diferentes partes de sí mismo para cumplir su objetivo logrando confundir a aquellos que intenten entender su contenido.

Parte 2

Para arreglar el código habría que fijarnos en cada caso, pero se debería en primer lugar chequear que se cumpla con los patrones SOLID en una base y una vez satisfecha esto, se deberá organizar el código para que no realice saltos confusos.

public void ProcessData(){

int y = mehtod(GetData);

SaveData(y);

}

public int method(int x){

x = x + 10;

if (x >50){

//Do something

} else {

//Do something else

}

return x;

}

### **Ejercicio 4:**

Parte 1:

En este código vemos que se utilizó la técnica de código “Cut−and−Paste Programming” la cual se basa en tener el mismo código en varios lugares en lugar de tener una solución general que se pueda aplicar en muchos casos.

Parte 2:

Para solucionarlo bastará con crear un método que verifique que el input sea válido, por ejemplo que no tenga espacios o características similares.

public void ValidateInput(string input){

//Validar input

}

### **Ejercicio 5**

Parte 1

Este es el antipatrón de “Lava Flow” el cual consiste que existe código de baja calidad que es viejo y obsoleto. Esto se puede dar por varías razones, una de ellas es que por temas de tiempo o razones similares por lo que no sirve de mucho; otra razón que ese código se utiliza pero nadie entiende cómo funciona

Parte 2

Se elimina el código o en cuyo caso se necesite, se crea uno que cumpla con los patrones, sea entendible y funcione.

### **Ejercicio 6**

Parte 1

En este ejercicio se hizo presente el antipatrón conocido como “golden hammer” ya que se intenta adaptar una herramienta, en este caso una clase, para cumplir cierta tarea aunque no sea la mejor forma.

Parte 2

Existen variadas soluciones que pueden aplicar y solucionar el dilema en cuestión, una posible es llamar a una superclase o alguna librería que realice la tarea que se tenga que hacer de la forma más eficiente posible.

### import librery from 'librery';

### 

### public void DoWork(){

### //Do something

### librery.DoSomething();

### //Do something else

### }

### **Ejercicio 7**

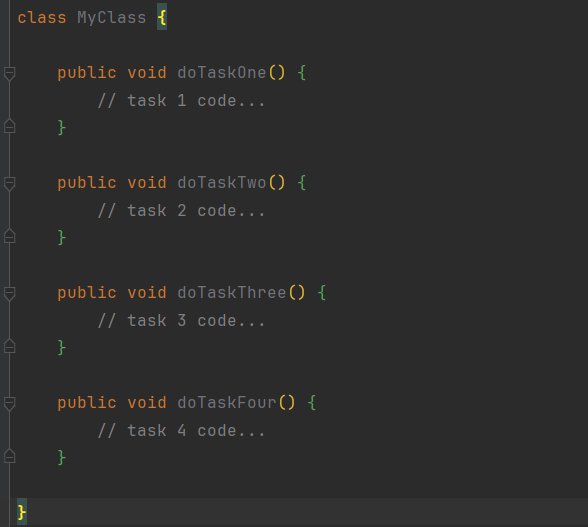
Golden Hammer, la misma librería que sirve para emplear un método de forma óptima, se utiliza en todos los métodos.

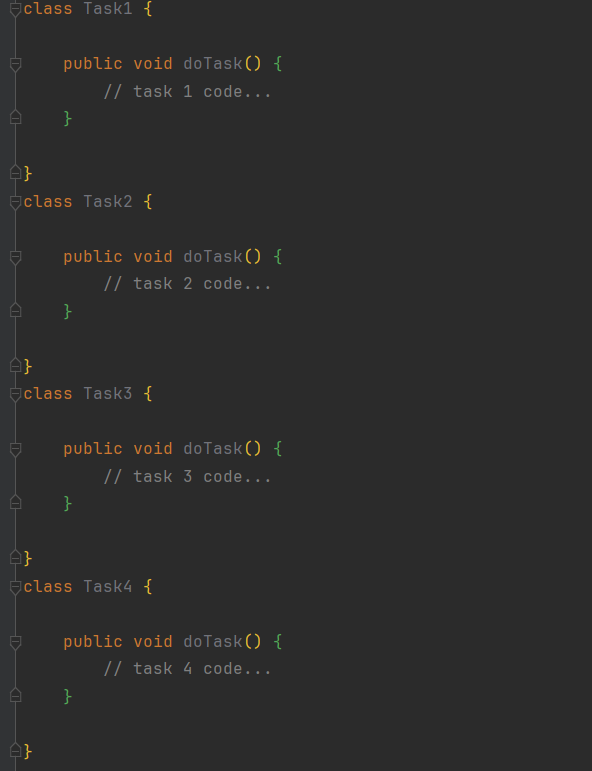


### 

### **Ejercicio 8**

The Blob, tiene un método que realiza muchas tareas, muy largas, lo mejor sería separar en varias clases, o al menos, varios métodos.





### **Ejercicio 9**

Parte 1

Este caso es muy similar al caso anterior de “Spaghetti Code”, pero la diferencia aquí es que la lógica implementada en los condicionales es confusa y muy poco clara, debido a que con un valor mayor a 1000 va entrar en la primer cápsula aunque cumpla con la condición de la segunda.

Parte 2

Para brindar una solución a este problema tenemos que corregir lo aclarado en la parte anterior por lo que la solución quedaría así:

public doble calculateNetSalary(double grosssalary){

double texRate;

double netSalary;

if(grosssalary > 5000){

taxrate = 0,2;

}else if(grosssalary > 1000){

taxrate = 0,3;

}else{

taxrate = 0,1;

}

netSalary = grossSalary - (grossSalary \* taxRate);

if (netSalary <0){

netSalary = 0;

}

return netSalary

}

### **Ejercicio 10**

Parte 1

En este último ejercicio se hace presente el anti patrón con el nombre de “Tester driven development”, este antipatrón se basa en que el desarrollo del código esa centralizado en la escritura de código en base a pruebas que se hacen, por ejemplo vemos que luego de varias iteraciones el código cambia añadiendo verificaciones y excepciones para controlar la cantidad máxima de productos que se pueden agregar en una sola vez, lo cual parece haber sido una respuesta a errores encontrados durante las pruebas, en lugar de haber sido planificado desde el inicio.

Parte 2

Considerar la lógica de negocio desde un principio como el hecho de la restricción de cantidad máxima.