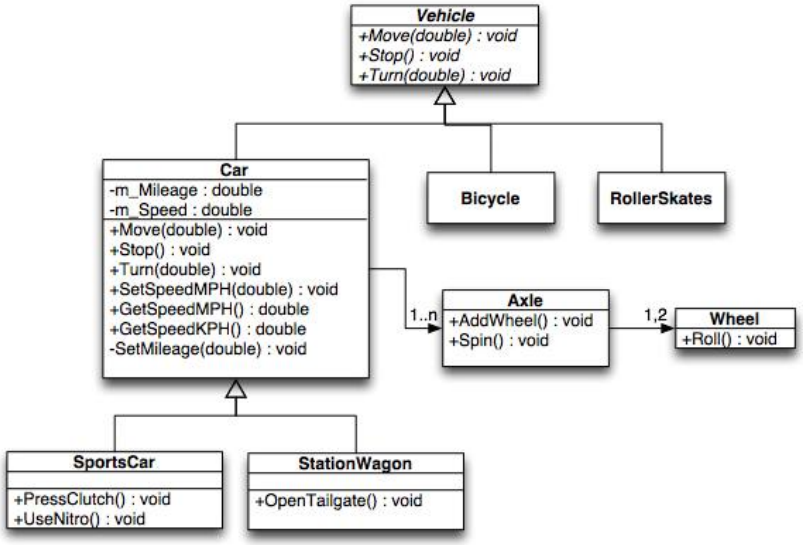
Principios del diseño de clases



Introducción

Los principios de diseño nos ayudan a exprimir todo el potencial de la programación orientada a objetos. Permitiéndonos programar software flexible, extensible y escalable. Estos principios suelen conocerse con el acrónimo de SOLID, que se refiere a cada una de las iniciales de estos principios:

- Single Responsabilty
- Open-Closed
- Liskov Substitution
- Inteface Segregation
- Dependency Inversión

Single Responsability

Nunca debe haber más de una razón para un cambiar una clase.

La responsabilidad de una clase es definida como una razón para cambiarla, porque una responsabilidad implica implementación. Y las implementaciones pueden cambiar en el futuro. Si una clase tiene más de una razón para cambiar entonces tiene más de una responsabilidad.

Vamos a ver la siguiente clase que representa el jugador de un equipo

```
public class Player{

        private String name;
        private int weight;
        private int height;
        private int position;

        public Player(String name, int weight, int height, int
position){
      this.name = name;
      this.weight = weight;
      this.height = height;
      this.position = position;
        }

        public int calculateBMI(){
                //Implementation
        }

        public void SaveToFile(){
                //Implementation
        }

    public void LoadFromFile(){
                //Implementation
        }
}
```

Como podemos observar la clase tiene la responsabilidad de trabajar con la información de un jugador. Pero ademas, también de guardar y cargar dicha información.

Como podemos observar las funciones de guardado y carga pretenden hacerlo des de un fichero. Si mas adelante tuviéramos que hacerlo des de una base de datos nos veriamos obligados a modificar el código, teniendo que retestear todo por la posible introducción de errores. Estos errores afectarían a la clase Player y a las que dependan de esta.

Resulta mucho mas util separar las responsabilidades en dos clases diferentes.

```
public class Player{

        private String name;
        private int weight;
        private int height;
        private int position;

        public Player(String name, int weight, int height, int
position){
      this.name = name;
      this.weight = weight;
      this.height = height;
      this.position = position;
        }

        public int calculateBMI(shapeType){
                //Implementation
        }
}

public class Db{

        private String playerInfo;

         public Db(String playerInfo, int weight, int height,
int position){
      this.playerInfo = playerInfo;
        }

        public void Save(){
                //Implementation
```

```
        }

    public void Load(){
                //Implementation
        }
}
```

Open-Closed Principle

Las entidades de software (clases, módulos, funciones…) deben estar preparadas para ser extensibles pero cerradas a modificaciones.

En el siguiente código podemos ver una clase que dibuja figuras y calcula su área:

```
public enum ShapeType {
    CIRCLE, SQUARE, TRIANGLE
}

public class ManageShape{

        private ShapeType shapeType;
    private int factorResize;

        public ManageShape(ShapeType shapeType){
      this.shapeType = shapeType;
        }

        void calculateArea(shapeType){
                switch(shapeType){
                        case shapeType.CIRCLE:
                                //implementation
                                break;
                        case shapeType.TRIANGLE:
                                //implementation
                                break;
                        case shapeType.SQUARE:
                                //implementation
                                break;
```

```
                        default:
                                //implementation
                                break;
                }
        }

        void drawShape(shapeType){
                switch(shapeType){
                        case shapeType.CIRCLE:
                                //implementation
                                break;
                        case shapeType.TRIANGLE:
                                //implementation
                                break;
                        case shapeType.SQUARE:
                                //implementation
                                break;
                        default:
                                //implementation
                                break;
                }
   }
}
```

Podemos encontrarnos con la necesidad de añadir mas figuras en el futuro. Estos cambios pueden introducir errores y causar varios fallos. Nuestro enum deber cambiar también y todos los módulos que depengan de la clase ManageShape con todo lo que ello supone.

El principio «open-closed» nos sugiere programar de la siguiente manera:

- Los módulos están abiertos para su extensión. La funcionalidad puede cambiar añadiendo nuevo código.
- El código existente esta cerrado a modificación y rara vez se tiene que modificar el código existente para cambiar el comportamiento del módulo.

La programación orientada a objetos nos provee de herramientas para implementar el principio «open-close». Una opción is

codificar nuestro programa utilizando herencia y polimorfismo. Llevado esto a nuestro código crearemos una estructura en la que cada figura heredara de ManageShape y cada subclase se encargara de mantener su funcionalidad y la enumeración no sera necesaria.

```
public abstract class ManageShape{

    private int factorResize;

        public abstract void calculateArea(){

        }

        public abstract void drawShape(){

   }
}

public class Triangle extends ManageShape{

        void calculateArea(shapeType){

        }

        void drawShape(shapeType){

   }
}

public class Circle extends ManageShape{

        void calculateArea(shapeType){

        }

        void drawShape(shapeType){

   }
```

```
}

public class Square extends ManageShape{

        void calculateArea(shapeType){

        }

        void drawShape(shapeType){

   }
}
```

De esta manera los módulos que necesiten utilizar una figura dependerán de la clase abstracta ManageShape y utilizaran las subclsases o punteros. Si se añaden mas clases de tipo figura el código existente no se tendrá que modificar, no habrá impacto en el resto del código. Otra ventaja es que el código común esta contenido en la clase base y así se evitan duplicidades.

Liskov Substitution

Las funciones que utilizan punteros o referencias a una clases base tienen que poder utilizar objetos de las clases derivadas sin tener conocimiento de ello. Es decir, una clase debe funcionar correctamente cuando se utiliza su sublcase en vez de la clase padre.

Cojamos el siguiente código de ejemplo:

```
class Bird {
        public void fly(){}
        public void eat(){}
}

class Crow extends Bird {}

class Ostrich extends Bird{
```

```
}
```

Como podemos obervars la classe Birs tiene los metodos fly() y eat(). Nos encontramos que al crear la clase Ostrich nos sobra el método fly(). Y aquí nos encontramos con el problema al utilizar la clase Ostrich (subclase de Bird) nos encontramos con un método que debería volver una excepció del tipo UnsupportedOperationException().

Una solución sería la siguiente:

```
class Bird{
	void eat();
}

class Crow extends Bird{
	public void fly(){
	}
	@Override
	public void eat() {
		// TODO Auto-generated method stub

	}
}

class Ostrich extends Bird{

	@Override
	public void eat() {
		// TODO Auto-generated method stub

	}
}
```

Una la clase Bird contiene solo el método eat(), el cual es comun en todas la aves. El método fly() el cual se implementará en la subclase y a su vez Ostrich.

Inteface Segregation

Los clientes no se deben ver forzados a depender de métodos que no utilizan.

Este principio trata de lidiar con interfícies no cohesionadas y excesivamente generalistas.Esto produce que los clientes se vean forzados a depender de métodos que jamas van a utilizar.

Por ejemplo Vamos a poner el caso de una interficie IFile utilizada para operaciones de ficheros. Dos clases heredan de ella TextFile y SocketFile

```
interface IFile {
        void Open();
        void Close();
        void Read();
        void Write();
        void Seek();
        void Position();
}

class TextFile implements IFile {

        public void Open(){
                //Implementation
        }

        public void Close(){
                //Implementation
        }

        public void Read(){
                //Implementation
        }

        public void Write(){
                //Implementation
        }

        public void Seek(){
                //Implementation
        }
```

```
        public void Position(){
                //Implementation
        }

}

class SocketFile implements IFile {

        public void Open(){
                //Implementation
        }

        public void Close(){
                //Implementation
        }

        public void Read(){
                //Implementation
        }

        public void Write(){
                //Implementation
        }

        public void Seek(){
                throw new NotImplementedException();
        }

        public void Position(){
                throw new NotImplementedException();
        }
}
```

Los métodos seek() y position() no son útiles para SocketFile. The echo está clase no puede implementarlos y por ello se lanza una excepción de tipo NotImplementedException(). Una solución que para nada resulta pulida. La clase puede acceder a los métodos pero para asegurarnos que no va haber ningún problema tenemos que lanzar una excepción.

El principio de segregación de interficies sugiere que una interficie no cohesionadas se deben dividir en interficies pequeñas, para que cada una de estas de servicio al cliente correspondiente.

Aplicando esto a nuestro caso, la interficie IFile se verá dividida en dos interficies

```
interface IFile {
        void Open();
        void Close();
        void Read();
        void Write();
}

interface IDiskFile implements IFile{
        void Seek();
        void Position();
}
```

La interficie IDiskFile será implementada por la clase TextFile, la cual implementara los metodos de las dos interficies (vease que IDiskFile implementa IFile). I SocketFile implementara solo la interficie IFile y no tendrá que preocuparse de métodos que no tulilizará (seek() y position()).

Dependency Inversión

Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Y ambos deben depender de la abstracción. La abstracción no debe depender de los detalles. Los detalles deberían depender de las abstracciones

En los diseños de análisis estructurado, el software que se crea tiende a tener una dependencia en la dirección equivocada. Las reglas de negocio dependen de los detalles de implementación de bajo nivel. Estos causa problemas cuando las

reglas de negocio cambian y la implementación de bajo nivel se tiene que modificar. Sino que más bien debería ser al revés; la implementación de bajo nivel debe depender de las reglas de negocio. Por lo tanto, la dependencia tiene que ser invertida; de ahí el nombre de principio de dependencia invertida.

Consideremos el ejemplo de un botón que enciende y apaga una lampara

```
class Lamp {

        public void turnOn(){
                //Implementation
        }

        public void turnOff(){
                //Implementation
        }

}

class Button {
        private Lamp lamp = new Lamp();
        private Boolean pressed = false;

        public void pressButton(){
                if(pressed){
                        lamp.turnOff();
                }else{
                        lamp.turnOn();
                        pressed = !pressed;
                }
        }
}
```

la clase Lamp tiene la implementación de las funcionalidades de la lampara. Està controlada por la clase Button que llama a los métodos turnOn() y turnOff(). Esto es un ejemplo de un diseño estructurado. La logica de control esta en Button y la implementación de bajo nivel en Lamp. De esta manera la clase Button contiene la implementación de alto nivel (logica) y

Lamp contiene la implementación de bajo nivel.

Es bastante obvio que la dirección de la dependencia es de Button a Lamp. Si la implementación de Lamp se tiene que modificar, también afectará a la clase Button. Esto significa que la lógica de negocio se ve afectada por la implementación de bajo nivel. Este ejemplo muestra un código muy pequeño pero en una aplicación grande puede generar una modificación en cascada de las clases de alto nivel.

Idealmente, las lógica de negocio define cómo se debe hacer la implementación de bajo nivel. Así, los módulos de bajo nivel idealmente dependen de la lógica de negocio y operan de acuerdo a ellos.Dado esto debemos invertir la dependencia de la clase Button en la clase Lamp.

La solución es la interficie Switchable que contiene el protocolo que todos los objetos deben seguir para controlar la clase Button. La clase Button no interactuará con Switchable en vez de con la clase Lamp.

```
interface Switchable {
        void turnOn();
        void turnOff();
}

class Lamp implements Switchable{

        public void turnOn(){
                //Implementation
        }

        public void turnOff(){
                //Implementation
        }
}

class Television implements Switchable{
```

```
        public void turnOn(){
                //Implementation
        }

        public void turnOff(){
                //Implementation
        }
}

class Button {
        private Switchable switchable = new Lamp();
        private Boolean pressed = false;

        public Button(Switchable s){
                this.switchable = s;
        }

        public void pressButton(){
                if(pressed){
                        this.switchable.turnOff();
                }else{
                        this.switchable.turnOn();
                        pressed = !pressed;
                }
        }
}
```

Ahora, es posible añadir mas objetos que se puedan encender y apagar a traves de Button sin que este tenga que ser modificado. La dependencia se ha invertido, los módulos de bajo nivel dependen de los módulos de alto nivel. Esto crea una estructura flexible facil de mantener y a la cual se pueden añadir nuevas funcionalidades

Observaciones

En esta segunda aproximación hemos definido los principios del diseño de clases. Este conjunto de reglas nos ayudará a mantener un código limpio, ordenado, fácil de mantener y

robusto. Todo ello enfocado a facilitar la nueva entrada de funcionalidades a nuestra aplicación y la corrección de errores.

Ruben.