CIENCIA DE LA COMPUTACION B

Leonardo Daniel Valdivia Ramos

1 ADAPTER

1.1 DEFINICION:

El patrón adapter sirve para integrar funcionalidades que no se pueden implementar por herencia. Esta permite establecer un puente para la clase incompatible y de esta manera se adapta la clase incompatible al resto.

1.2 DESCRIPCION

En una clase abstracta que tiene métodos sus clases heredadas siguen el mismo diseño, en el caso que se quiera adaptar a la clase base una nueva clase heredada se deberá crear una clase donde se adapten los métodos de la clase incompatible a los métodos de la clase base. La usaremos cuando necesitamos hacer compatibles dos interfaces. tambien sirve cuando trabajamos con librerias externas.

1.3 EJEMPLO

Creación de la clase base Motor en esta clase abstracta crearemos tres métodos encender, acelerar, apagar.

```
1 #include <iostream>
2 using namespace std;
3 class Motor{
4   public:
5     Motor(){}
6     virtual void encender()=0;
7     virtual void acelerar()=0;
8     virtual void apagar()=0;
9 };
```

Clase heredada Motorcomun con la implementación de los métodos encender, acelerar y apagar.

```
1  class Motorcomun : public Motor{
2    public:
3    Motorcomun(){
4      cout<<"creando motor comun"<<endl;
5    }
6    void encender(){
7      cout<<"encendiendo motor comun"<<endl;
8    }
9    void acelerar(){
10     cout<<"acelerando motor comun"<<endl;
11   }
12   void apagar(){
13     cout<<"apagando motor comun"<<endl;
14   }
15 };
```

Clase heredada MotorEconomico con la implementación de los métodos encender,acelerar y apagar.

```
1  class Motoreconomico : public Motor{
2    public:
3    Motoreconomico(){
4      cout<<"creando motor economico"<<endl;
5    }
6    void encender(){
7      cout<<"encendiendo motor economico"<<endl;
8    }
9    void acelerar(){
10     cout<<"acelerando motor economico"<<endl;
11   }
12   void apagar(){
13     cout<<"apagando motor economico"<<endl;
14   }
15 };
```

Clase Motorelectrico es la clase incompatible a la clase base y por ende para adaptarla a nuestra clase base tendremos que crear la clase Motorelecetricoadapter que será la compatible.

```
1  class Motorelectrico{
2    private:
3      bool conectado;
4    public:
5      Motorelectrico(){
6        cout<<"creando motor electrico"<<endl;
7        conectado=false;
8      }
9      void conectar() {
10         cout<<"Conectando motor electrico"<<endl;
11         this->conectado = true;
12     }
13
14     void activar() {
15         if (conectado==false) {
16            cout<<"nose puede activar porque no esta conectado el
                   motor electrico"<<endl;
17         } else
18             cout<<"Esta conectado, activando motor electrico"<<
                   endl;
19     }
20
21     void moverMasRapido() {
22         if (conectado==false) {
23            cout<<"el motor electrico porque no esta conectado...
                  "<<endl;
24         } else {
25             cout<<"Moviendo mas rapido...aumentando voltaje"<<
                   endl;
26         }
27     }
28
29     void detener() {
30         if (conectado==false) {
31             cout<<"No se puede detener motor electrico porque no
                    esta conectado"<<endl;
32         } else
33             cout<<"Deteniendo motor electrico"<<endl;
34     }
35
36     void desconectar() {
```

```
37          cout<<"Desconectando motor electrico"<<endl;
38          this->conectado = false;
39      }
40  };
```

Clase motorelectricoadapter heredada de la clase base y en esta implementaremos los métodos encender, acelerar y apagar haciendo uso también de una instancia de Motorelectrico e adaptaremos los métodos de este a los métodos de la clase base.

```
1  class MotorelectricoAdapter : public Motor{
2    private:
3      Motorelectrico motorelectrico;
4    public:
5      void encender(){
6        motorelectrico.conectar();
7        motorelectrico.activar();
8      }
9      void acelerar(){
10       motorelectrico.moverMasRapido();
11     }
12     void apagar(){
13       motorelectrico.detener();
14       motorelectrico.desconectar();
15     }
16 };
```

Creación de una instancia de motorcomun con los métodos de la clase Motor y creación de una instancia de MotorelectricoAdapter que hace uso de los métodos de la clase base Motor.

```
1  int main(){
2    Motorcomun motorcomun;
3    motorcomun.encender();
4    motorcomun.acelerar();
5    motorcomun.apagar();
6
7    MotorelectricoAdapter electrico;
8    electrico.encender();
9    electrico.acelerar();
10   electrico.apagar();
11 }
```

2 DECORATOR

2.1 DEFINICION

Asignar funciones adicionales a un objeto de manera dinamica. Los decoradores ofrecen son flexibles y se usan para las subclases para extender su funcionalidad. Embellecimiento especificado por el cliente de un objeto central.

2.2 DESCRIPCION

Se hace uso de una clase padre y una clase heredada la cual crea un objeto, se crea una clase decorator la cual tendra un puntero a la clase base, luego a este se le agregan metodos que cambiaran de manera dinamica el objeto creado.

2.3 EJEMPLO

Creamos la clase padre TiendaAbstracta

```
1 #include <iostream>
2 using namespace std;
3 class TiendaAbstracta{
4     public:
5
6         virtual void mostrar() = 0;
7 };
```

Creamos la clase heredad tienda que es la que creara instancias.

```
1 class Tienda : public TiendaAbstracta{
2     public:
3         void mostrar()
4         {
5             cout << " Tienda ";
6         }
7 };
```

Creamos la clase heredada TiendaDecorator la cual tendra un puntero a la clase base el cual modificara añadiendole funciones mediante sus clases que heredaran de esta.

```
1 class Tiendadecorator : public TiendaAbstracta
```

```
2 {
3   protected:
4         TiendaAbstracta* _tiendaAbstracta;
5     public:
6         Tiendadecorator(TiendaAbstracta* tiendaAbstracta)
7         {
8             _tiendaAbstracta = tiendaAbstracta;
9         }
10        virtual void mostrar() = 0;
11 };
```

Clase heredada de TiendaDecorator que cambiara al objeto.

```
1 class BordeDecorator : public Tiendadecorator
2 {
3     public:
4         BordeDecorator(TiendaAbstracta* tiendaAbstracta) :
              Tiendadecorator(tiendaAbstracta){}
5         virtual void mostrar()
6         {
7             cout << "|";
8             _tiendaAbstracta->mostrar();
9             cout << "|";
10        }
11 };
```

Clase heredada de TiendaDecorator que cambiara al objeto.

```
1 class BotonDeAyuda : public Tiendadecorator
2 {
3     public:
4         BotonDeAyuda(TiendaAbstracta* tiendaAbstracta) :
              Tiendadecorator (tiendaAbstracta){}
5         virtual void mostrar()
6         {
7             _tiendaAbstracta->mostrar();
8             cout << "[Boton de Ayuda]";
9         }
10 };
```

Creacion del un puntero a una Tienda y mediante Botondeayuda y Bordedecorator "envolvemos al objeto para cambiar su interfaz"

```
1 int main()
2 {
3   Tienda* tienda;
4   tienda = new Tienda;
```

```
5       BotonDeAyuda tiendaconbotondeayuda(tienda);
6       tiendaconbotondeayuda.mostrar();
7       cout << endl;
8       BordeDecorator ventanaConBotonDeAyudaYBorde(&
            tiendaconbotondeayuda);
9       ventanaConBotonDeAyudaYBorde.mostrar();
10      cout << endl;
11      return 0;
12  }
```

3 REFERENCIAS

http://codejavu.blogspot.com/2013/08/ejemplo-patron-decorator.html
https://www.genbeta.com/desarrollo/patrones-de-diseno-adapter
http://codejavu.blogspot.com/2013/08/ejemplo-patron-adapter.html