1. **QUE ES MVC(MODELO VISTA CONTROLADOR)**

**R:** El termino MVC (Model-View-Controller o Modelo-Vista-Controlador) es un patrón de diseño que separa los datos, la lógica y las interfaces de usuario. De tal manera que, está separado en tres componentes: modelo, controlador y vista cumpliendo de manera uniforme con lo que es el concepto de patrón de diseño, por lo tanto MVC separa las responsabilidades dentro de nuestra aplicación y es muy utilizado en la web por su enfoque y las ventajas que ofrece con respecto a algunas otras formas o patrones de desarrollo de aplicaciones web.

* Modelo

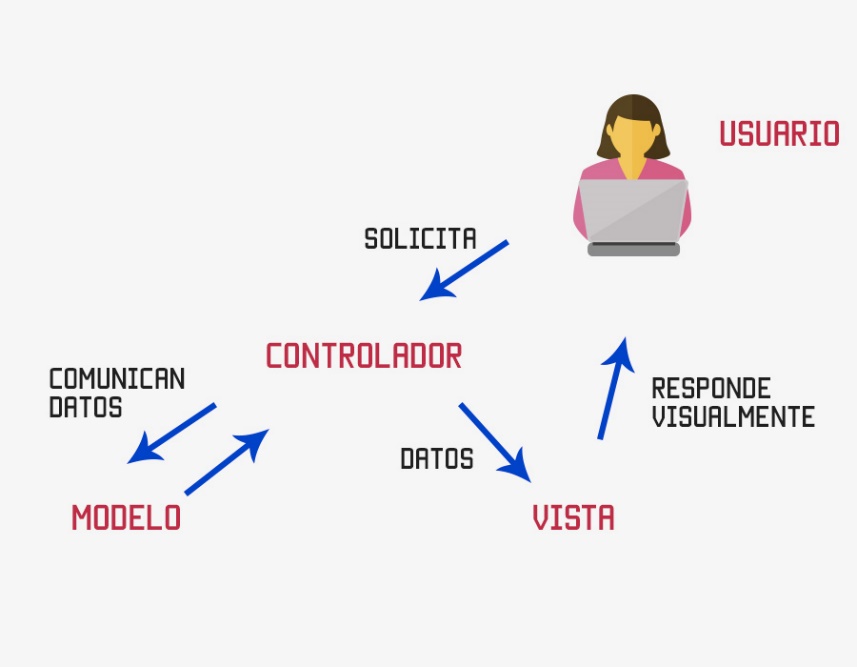
Es la capa encargada de los datos, es decir, tiene mecanismos para acceder a la información y también para actualizar su estado. Comúnmente se encarga de comunicarse con la base de datos mediante funciones que accederán a las tablas y realizarán las funciones habituales de datos.

* Vista

Se trata del código que nos permitirá presentar los datos que el modelo nos proporciona, como ejemplo podríamos decir que en una aplicación web es el código HTML que nos permite mostrar la salida de los datos procesados.

* Controlador

Es la capa que sirve de enlace entre la vista y el modelo. Envía comandos al modelo para actualizar su estado, y a la vista correspondiente para cambiar su presentación, pero no es el encargado de manipular los datos ni de generar una salida.



1. **10 EJEMPLOS DE PATRONES DE DISEÑO**

**R:**

1. **Factory Method:**

Define una interfaz para crear un objeto, pero permite a las subclases decidir que clases instancia. Factory Method permite a una clase derivar la instanciación a las subclases.

class Button(object):

html = ""

def get\_html(self):

return self.html

class Image(Button):

html = "<img></img>"

class Input(Button):

html = "<input></input>"

class Flash(Button):

html = "<obj></obj>"

class ButtonFactory():

def create\_button(self, typ):

targetclass = typ.capitalize()

return globals()[targetclass]()

1. **Singleton:**

Nos garantiza la existencia de una única instancia para una clase.

class Singleton {

public:

static Singleton& Handle();

private:

static Singleton\* psingle;

Singleton();

Singleton( const Singleton& );

Singleton& operator=( const Singleton& );

};

1. **Iterator:**

El patrón Iterator se usa en el contexto de las listas y conjuntos. Tenemos una serie de objetos que internamente trabajan con conjuntos de elementos y necesitamos manipularlos abstrayéndonos de cómo están implementados internamente. De esta manera si por alguna razón de eficiencia o funcionalidad necesitáramos cambiar la implementación interna del conjunto de elementos, el resto de nuestro sistema seguiría funcionando sin problemas.

Gambas class file

Private lista As Variant[]

Public Sub \_new(l As Variant[])

  lista = l

End

Public Function getlista() As Variant[]

  Return lista

End

Public Function currenItem() As Variant

  'dato de la lista actual

End

Public Function primero() As Variant

End

Public Function siguiente() As Variant

End

Public Function isdone() As Boolean

  'hay alguno mas?

End

1. **Builder:**

Este patrón puede ser utilizado cuando necesitemos crear objetos complejos compuestos de varias partes independientes.

package Builder01;

public class Main

{

public static void main(String[] args)

{

// Crear el objeto Director

Director objFabrica = new Director();

// Crear los objetos ConcreteBuilder

BuilderCoche base = new ConstructorCocheBase();

BuilderCoche medio = new ConstructorCocheMedio();

BuilderCoche full = new ConstructorCocheFull();

// Construir un coche con equipamiento base

objFabrica.construir( base );

Coche cocheBase = base.getCoche();

1. **Prototype:**

Este patrón nos será útil si necesitamos crear y manipular copias de otros objetos.

package Prototype01;

public class Main

{

public static void main(String[] args)

{

GestorEnemigo objGP = new GestorEnemigo();

// Obtenemos el Guerrero original

Enemigo g1 = objGP.getEnemigo("Warrior1");

System.out.println("==============================");

// Mostramos su datos

System.out.println("El Guerrero original se llama [" + g1.getNombre() + "]");

System.out.println("Su arma es [" + g1.getArma() + "]");

System.out.println("==============================");

// Obtener un segundo Guerrero (clon del anterior)

Enemigo g2 = objGP.getClon("Warrior1");

1. **Proxy:**

El patrón proxy trata de proporcionar un objeto intermediario que represente o sustituya al objeto original con motivo de controlar el acceso y otras características del mismo**.**

public class ProxyServer implements Server{

RealServer srv;

private int port;

private String host;

public ProxyServer(int port, String host){

this.port=port;

this.host=host;

srv=null;

System.out.println("Proxy iniciado...");

}

1. **Decorator:**

Sencillo e interesante patrón que permite añadir funcionalidades a un objeto en aquellos casos en los que no sea necesario o recomendable hacerlo mediante herencia

package estructurales.decorator.decorator01;

public class DecoradorDesplazamientoHoriz extends DecoradorDesplazamiento

{

public DecoradorDesplazamientoHoriz( IVentana v )

{

super( v );

}

1. **Mediator:**

Se usa para definir una clase que hará de mediadora encapsulando la comunicación entre los objetos, evitándose con ello la necesidad de que lo hagan directamente entre sí.

def retorna\_objecte(f):

....: def obj():

....: return object()

....: return obj

....:

In [17]: def di\_hola():

....: return "Hola"

....:

In [18]: di\_hola = retorna\_objecte(di\_hola)

In [19]: di\_hola()

Out[19]: <object object at 0xf7f745e8>

1. **Observer:**

Es un patrón de comportamiento el cual define una relación entre clases de uno-a-muchos en el cual si un objeto cambia de estado los objetos dependientes serán notificados y serán actualizados de forma automática.

abstract class AbstractSubject {

abstract function attach(AbstractObserver $observer\_in);

abstract function detach(AbstractObserver $observer\_in);

abstract function notify();

}

1. **Strategy:**

Este patrón estrategia permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.

class StrategyExample:

def \_\_init\_\_(self, func=None):

self.name = "Strategy Example 0"

if func:

self.execute = create\_bound\_method(func, self)

def execute(self):

print(self.name)

if \_\_name\_\_ == "\_\_main\_\_":

strat0 = StrategyExample()

strat1 = StrategyExample(executeReplacement1)

strat0.execute()

1. **Delegation:**

Se utiliza cuando se desea extender y reutilizar la funcionalidad de una clase sin utilizar la herencia.

class C extends A {

B objB;

C ( ) { // En la constructora se puede crear obj. de B

objB=new B();

}

void b1( ) { objB.b1( );}

….