**Modelo vista controlador ejemplo en c++**

El modelo responde directamente para el manejo de datos. Por ejemplo, el componente del componente accede a la [base de datos MySQL](https://helloacm.com/how-to-generate-100k-test-data-to-mysql-database/) . El modelo no debe depender de otros componentes como View o Controller. En otras palabras, al modelo no le importa cómo se pueden mostrar sus datos o cuándo deben actualizarse.

Los cambios de datos en el Modelo generalmente se publicarán a través de algunos manejadores de eventos. Por ejemplo, el modelo de vista debe registrarse en el modelo para que entienda los cambios en los datos. Podemos definir una función de devolución de llamada cuando los datos cambian:

*// common.h*

*//* [*https://helloacm.com/model-view-controller-explained-in-c/*](https://helloacm.com/model-view-controller-explained-in-c/)

#pragma once

#include <string>

using namespace std;

typedef void (\*DataChangeHandler)(string newData);

*DataChangeHandler* es ahora un tipo de puntero de función que devuelve vacío y toma un parámetro de una cadena (tipo de datos). El modelo es responsable de la recuperación de datos y, opcionalmente, puede registrar el evento de cambio de datos.

*// model.h*

*//* [*https://helloacm.com/model-view-controller-explained-in-c/*](https://helloacm.com/model-view-controller-explained-in-c/)

#pragma once

#include <string>

using namespace std;

#include "common.h"

*// Model is responsible for data get and set*

class Model {

    public:

        Model(const string &data) {

            this->SetData(data);

        }

        Model() { } *// default constructor*

        string Data(){

            return this->data;

        }

        void SetData(const string &data) {

            this->data = data;

            if (this->event != nullptr) { *// data change callback event*

                this->event(data);

            }

        }

*//  register the event when data changes.*

        void RegisterDataChangeHandler(DataChangeHandler handler) {

            this->event = handler;

        }

    private:

        string data = "";

        DataChangeHandler event = nullptr;

};

VER

El componente Vista sabe cómo presentar los datos a los usuarios. Necesita acceder al Modelo y normalmente necesita definir su función 'Render ()'.

*// view.h*

*//* [*https://helloacm.com/model-view-controller-explained-in-c/*](https://helloacm.com/model-view-controller-explained-in-c/)

#pragma once

#include <iostream>

#include "model.h"

*// View is responsible to present data to users*

class View {

    public:

        View(const Model &model) {

            this->model = model;

        }

        View() {}

        void SetModel(const Model &model) {

            this->model = model;

        }

        void Render() {

            std::cout << "Model Data = " << model.Data() << endl;

        }

    private:

        Model model;

};

## CONTROLADOR

El controlador puede pedirle al modelo que actualice sus datos. Además, el Controlador puede pedirle a la Vista que cambie su presentación, por ejemplo, Mostrando un diálogo en lugar de Salir a la consola. Básicamente, es un componente que recibe información del usuario y envía comandos a la Vista o al Modelo.

*// controller.h*

*//* [*https://helloacm.com/model-view-controller-explained-in-c/*](https://helloacm.com/model-view-controller-explained-in-c/)

#pragma once

#include "model.h"

#include "view.h"

*// Controller combines Model and View*

class Controller {

    public:

        Controller(const Model &model, const View &view) {

          this->SetModel(model);

          this->SetView(view);

        }

        void SetModel(const Model &model) {

            this->model = model;

        }

        void SetView(const View &view) {

            this->view = view;

        }

*// when application starts*

        void OnLoad() {

            this->view.Render();

        }

    private:

        Model model;

        View view;

};

## MVC DEMO

Con las tres clases de componentes anteriores, podemos tener el siguiente código para demostrar MVC.

*// mvc.cpp*

*//* [*https://helloacm.com/model-view-controller-explained-in-c/*](https://helloacm.com/model-view-controller-explained-in-c/)

#include <iostream>

#include "view.h"

#include "model.h"

#include "controller.h"

#include "common.h"

using namespace std;

void DataChange(string data) {

  cout << "Data Changes: " << data <<endl;

}

int main() {

    Model model("Model");

    View view(model);

*// register the data-change event*

    model.RegisterDataChangeHandler(&DataChange);

*// binds model and view.*

    Controller controller(model, view);

*// when application starts or button is clicked or form is shown...*

    controller.OnLoad();

    model.SetData("Changes"); *// this should trigger View to render*

    return 0;

}

**Singleton en java**

1. package com.arquitecturajava;
2. public class Configurador {
3. private String url;
4. private String baseDatos;
5. private static **final** Configurador miconfigurador;
6. public static Configurador **getConfigurador**(String url,String baseDatos) {
7. **if** (miconfigurador==null) {
8. miconfigurador=**new** **Configurador**(url,baseDatos);
9. }
10. **return** miconfigurador;
11. }
12. private **Configurador**(String url,String baseDatos){
13. **this**.url=url;
14. **this**.baseDatos=baseDatos;
15. }
16. public String **getUrl**() {
17. **return** url;
18. }
19. public void **setUrl**(String url) {
20. **this**.url = url;
21. }
22. public String **getBaseDatos**() {
23. **return** baseDatos;
24. }
25. public void **setBaseDatos**(String baseDatos) {
26. **this**.baseDatos = baseDatos;
27. }
28. }

Para conseguir que una clase sea de tipo Singleton necesitamos en primer lugar que **su constructor sea privado.** De esa forma ningún programa será capaz de construir objetos de este tipo**. En** segundo lugar, necesitaremos disponer de una **variable estática privada** que almacene una referencia al objeto que vamos a crear a través del constructor. Por último, **un método estático público que se encarga de instanciar el objeto la primera vez y almacenarlo en la variable estática.**

1. package com.arquitecturajava;
2. public class Principal {
3. public static void **main**(String[] args) {
4. Configurador c= Configurador.getConfigurador("miurl", "mibaseDatos");
5. System.out.println(c.getUrl());
6. System.out.println(c.getBaseDatos());
7. }
8. }

**Factory phyton**

In serializer\_demo.py

import json

import xml.etree.ElementTree as et

class Song:

def \_\_init\_\_(self, song\_id, title, artist):

self.song\_id = song\_id

self.title = title

self.artist = artist

class SongSerializer:

def serialize(self, song, format):

if format == 'JSON':

song\_info = {

'id': song.song\_id,

'title': song.title,

'artist': song.artist

}

return json.dumps(song\_info)

elif format == 'XML':

song\_info = et.Element('song', attrib={'id': song.song\_id})

title = et.SubElement(song\_info, 'title')

title.text = song.title

artist = et.SubElement(song\_info, 'artist')

artist.text = song.artist

return et.tostring(song\_info, encoding='unicode')

else:

raise ValueError(format)

>>> import serializer\_demo as sd

>>> song = sd.Song('1', 'Water of Love', 'Dire Straits')

>>> serializer = sd.SongSerializer()

>>> serializer.serialize(song, 'JSON')

'{"id": "1", "title": "Water of Love", "artist": "Dire Straits"}'

>>> serializer.serialize(song, 'XML')

'<song id="1"><title>Water of Love</title><artist>Dire Straits</artist></song>'

>>> serializer.serialize(song, 'YAML')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "./serializer\_demo.py", line 30, in serialize

raise ValueError(format)

ValueError: YAML