

**Tecnológico Nacional de México Campus**

**Milpa Alta II**

**INGENIERÍA EN SISTEMAS COMPUTACIONALES**

*Tema : Modelo MVC*

**Alumnos:**

Barrientos Segura Ariel

**Docente:**

Ing. Julio Cesar Castor Diaz

Milpa Alta, CDMX, Septiembre, 2020

1. **Introducción (Definición de MVC y su relación con los patrones de diseño de software).**

Modelo Vista Controlador (MVC)

Modelo Vista Controlador (MVC) es un estilo de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

Se trata de un modelo muy maduro y que ha demostrado su validez a lo largo de los años en todo tipo de aplicaciones, y sobre multitud de lenguajes y plataformas de desarrollo.

* El **Modelo** que contiene una representación de los datos que maneja el sistema, su lógica de negocio, y sus mecanismos de persistencia.
* La **Vista**, o interfaz de usuario, que compone la información que se envía al cliente y los mecanismos interacción con éste.
* El **Controlador**, que actúa como intermediario entre el Modelo y la Vista, gestionando el flujo de información entre ellos y las transformaciones para adaptar los datos a las necesidades de cada uno.

#### **El modelo es el responsable de:**

* Acceder a la capa de almacenamiento de datos. Lo ideal es que el modelo sea independiente del sistema de almacenamiento.
* Define las reglas de negocio (la funcionalidad del sistema). Un ejemplo de regla puede ser: "Si la mercancía pedida no está en el almacén, consultar el tiempo de entrega estándar del proveedor".
* Lleva un registro de las vistas y controladores del sistema.
* Si estamos ante un modelo activo, notificará a las vistas los cambios que en los datos pueda producir un agente externo (por ejemplo, un fichero por lotes  que actualiza los datos, un temporizador que desencadena una inserción, etc.).

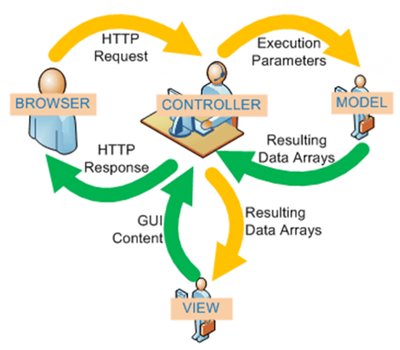
#### **El controlador es responsable de:**

* Recibe los eventos de entrada (un clic, un cambio en un campo de texto, etc.).
* Contiene reglas de gestión de eventos, del tipo "SI Evento Z, entonces Acción W". Estas acciones pueden suponer peticiones al modelo o a las vistas. Una de estas peticiones a las vistas puede ser una llamada al método "Actualizar()". Una petición al modelo puede ser "Obtener\_tiempo\_de\_entrega ( nueva\_orden\_de\_venta )".

#### **Las vistas son responsables de:**

* Recibir datos del modelo y los muestra al usuario.
* Tienen un registro de su controlador asociado (normalmente porque además lo instancia).
* Pueden dar el servicio de "Actualización()", para que sea invocado por el controlador o por el modelo (cuando es un modelo activo que informa de los cambios en los datos producidos por otros agentes).

El flujo que sigue el control generalmente es el siguiente:

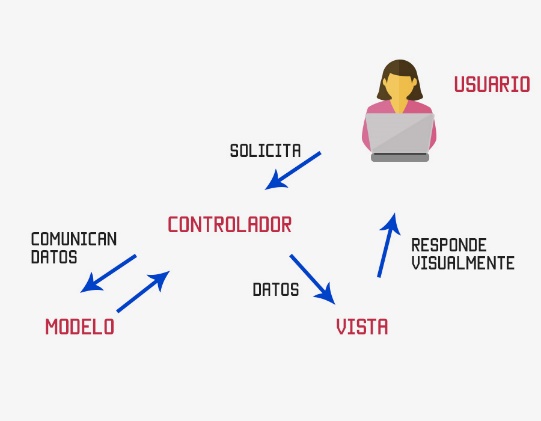


1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.)
2. El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (handler) o callback.
3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario). Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista. Sin embargo, se podría utilizar el patrón Observador para proveer cierta indirección entre el modelo y la vista, permitiendo al modelo notificar a los interesados de cualquier cambio. Un objeto vista puede registrarse con el modelo y esperar a los cambios, pero aun así el modelo en sí mismo sigue sin saber nada de la vista. El controlador no pasa objetos de dominio (el modelo) a la vista aunque puede dar la orden a la vista para que se actualice. Nota: En algunas implementaciones la vista no tiene acceso directo al modelo, dejando que el controlador envíe los datos del modelo a la vista.
5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

**2.¿Que frameworks utiliza el modelo MVC?**

El MVC o Modelo-Vista-Controlador es un patrón de arquitectura de software que, utilizando 3 componentes (Vistas, Models y Controladores) separa la lógica de la aplicación de la lógica de la vista en una aplicación. Es una arquitectura importante puesto que se utiliza tanto en componentes gráficos básicos hasta sistemas empresariales; la mayoría de los frameworks modernos utilizan MVC (o alguna adaptación del MVC) para la arquitectura, entre ellos podemos mencionar a [Ruby on Rails](https://codigofacilito.com/cursos/rails), [Django](https://codigofacilito.com/cursos/django), [AngularJS](https://codigofacilito.com/cursos/angularjs" \t "_blank) y muchos otros más. En este pequeño artículo intentamos introducirte a los conceptos del MVC.

## UNA IMAGEN



## UNA ANALOGÍA

Una que me gusta mucho es la de la televisión. En tu televisión puedes ver distintos canales distribuidos por tu tu proveedor de cable o televisión (que representa al modelo), todos los canales que puedes **ver** son la vista, y tú cambiando de canal, **controlando qué ves** representas al controlador.

## LA EXPLICACIÓN

Los puntos anteriores so sólo para proveer background, y que ojalá puedas utilizar las referencias ahora que vamos a explicar qué es.

Antes que nada, me gustaría mencionar por qué se utiliza el **MVC**, la razón es que nos permite separar los componentes de nuestra aplicación dependiendo de la responsabilidad que tienen, esto significa que cuando hacemos un cambio en alguna parte de nuestro código, esto no afecte otra parte del mismo. Por ejemplo, si modificamos nuestra Base de Datos, sólo deberíamos modificar el modelo que es **quién se encarga de los datos** y el resto de la aplicación debería permanecer intacta. Esto respeta el principio de la responsabilidad única. Es decir, una parte de tu código no debe de saber qué es lo que hace toda la aplicación, sólo debe de tener una responsabilidad.

En web, el MVC funcionaría así. Cuando el usuario manda una petición al navegador, digamos quiere ver el [curso de AngularJS](https://codigofacilito.com/cursos/angularjs), el controlador responde a la solicitud, porque él es el que controla la lógica de la app, una vez que el controlador nota que el usuario solicitó el curso de Angular, le pide al modelo la información del curso.

El modelo, que se encarga de los datos de la app, consulta la base de datos y digamos, obtiene todos los vídeos del curso de AngularJS, la información del curso y el título, el modelo responde al controlador con los datos que pidió (nota como en la imagen las flechas van en ambos sentidos, porque el controlador pide datos, y el modelo responde con los datos solicitados).

Una vez el controlador tiene los datos del curso de AngularJS, se los manda a la vista, la vista aplica los estilos, organiza la información y construye la página que vez en el navegador.

Resumamos entonces los conceptos.

### MODELO

Se encarga de los datos, generalmente (pero no obligatoriamente) consultando la base de datos. Actualizaciones, consultas, búsquedas, etc. todo eso va aquí, en el modelo.

### CONTROLADOR

Se encarga de... controlar, recibe las órdenes del usuario y se encarga de solicitar los datos al modelo y de comunicárselos a la vista.

### VISTAS

Son la representación visual de los datos, todo lo que tenga que ver con la interfaz gráfica va aquí. Ni el modelo ni el controlador se preocupan de cómo se verán los datos, esa responsabilidad es únicamente de la vista

**3 . ¿Qué ventajas ofrece el modelo MVC?**

El uso del patrón MVC ofrece múltiples ventajas sobre otras maneras de desarrollar aplicaciones con interfaz de usuario, y en especial para la Web. Sin entrar en detalles aquí, por que la extensión del artículo ya es grande, comentaré a continuación algunas de ellas:

* La clara separación de responsabilidades impuesta por el uso del patrón MVC hace que los componentes de nuestras aplicaciones tengan sus misiones bien definidas. Por lo tanto, nuestros sistemas serán **más limpios**, **simples**, más fácilmente **mantenibles** y, a la postre, más **robustos**.
* **Mayor velocidad de desarrollo en equipo**, que es consecuencia de lo anterior, ya que al estar separado en tres partes tan diferenciadas, diferentes programadores pueden ocuparse de cada parte en paralelo. Esto la hace **ideal para el desarrollo de aplicaciones grandes**.
* **Múltiples vistas** a partir del mismo modelo, pudiendo reaprovechar mucho mejor los desarrollos y asegurando consistencia entre ellas.
* **Facilidad para realización de pruebas unitarias**.

**4. Patrones de diseño y el patrón MVC.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PATRONES DE DISEÑO  “Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de software.”  Brindan una solución ya probada y documentada a problemas de desarrollo de software que están sujetos a contextos  similares. Debemos tener presente los siguientes elementos de un patrón: su nombre, el problema  (cuando aplicar un patrón), la solución (descripción abstracta del problema) y las consecuencias (costos y beneficios).  Los patrones de diseño se clasifican en:  \* PATRONES DE CREACIÓN: Resuelven problemas con la creación de instancias de objetos.    Tipos de patrones de creación:                                                                                                                                                                                    Abstract factory  Proporciona una clase que delega la creación de una o más clases concretas con el fin de entregar objetos específicos.  Este patrón puede ser utilizado cuando:  - La creación de objetos debe ser independiente del sistema que los utilice.  -Los sistemas deben ser capaces de utilizar múltiples familias de objetos.  - Se usan bibliotecas sin exponer detalles de la implementación.    Estructura de Abstract factory  EJEMPLO:  class EnchantedMazeFactory : public MazeFactory {  public:  EnchantedMazeFactory();  virtual Room\* MakeRoom(int n) const  { return new EnchantedRoom(n, CastSpell()); }  virtual Door\* MakeDoor(Room\* r1, Room\* r2) const  { return new DoorNeedingSpell(r1, r2);  }  protected:  Spell\* CastSpell() const;  };  // Make it possible to have rooms with bombs  Wall\* BombedMazeFactory::MakeWall () const {  return new BombedWall;  }  Room\* BombedMazeFactory::MakeRoom(int n) const {  return new RoomWithABomb(n);  }  Factory Method  Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar.  Permite que una clase delegue en sus subclases la creación de objetos.  Motivación.   Ej.: un framework que pueda presentar distintos tipos de documentos (similar a MFCs).  -Dos abstracciones clave: aplicación y documento (ambas clases abstractas). Hay que subclasificarlas.  -La clase aplicación no sabe qué subclase documento instanciar.       Estrcutura de Factory Method  Singleton  Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.   Este patrón puede ser utilizado cuando:  - Se requiere exactamente una instancia de una clase.  -Es necesario acceso controlado a un solo objeto.    \*  PATRONES ESTRUCTURALES:  -Establecen cómo se componen clases y objetos para formar estructuras mayores que implementan nueva funcionalidad.  -Los patrones de clase usan la herencia para componer interfaces o implementaciones (ej.: Adapter).  Tipos de patrones de creación:  Composite  Propósito:  -Componer objetos en estructuras arborescentes para representar jerarquías parte-conjunto.     Estructura de Composite  Participantes.  \* Component (Grafico).  - Declara la interfaz de los objetos de la composición.  - Implementa el comportamiento por defecto de la interfaz común a todas las clases.  - Declara las interfaces para acceder y gestionar los hijos.  - (opcional) Define una interfaz para acceder al padre de un componente en la estructura recursiva y la implementa, si es  apropiado.  \* Leaf (Linea, Rectangulo, …).  - Representa objetos hoja en la composición. Una hoja no tiene hijos.  - Define el comportamiento de los objetos primitivos en la composición.  \*Composite (Dibujo)  - Define el comportamiento de los objetos con hijos en la composición.  - Almacena componentes hijo.  - Implementa operaciones relacionadas con los hijos de la interfaz de Component.  \* Client  - Manipula objetos en la composición a través de la interfaz de Component.  Ejemplo:  class Equipment {  public:  virtual ~Equipment();  const char\* Name() { return \_name; }  virtual Watt Power();  virtual Currency NetPrice();  virtual Currency DiscountPrice();  virtual void Add(Equipment\*);  virtual void Remove(Equipment\*);  virtual Iterator\* CreateIterator() {return 0;}  protected:  Equipment(const char\*);  private:  const char\* \_name;  };  Proxy  Propósito:  Proporcionar un representante o substituto de otro objeto para controlar el acceso a este.     Estructura de Proxy  Aplicabilidad.  - Cuando hay necesidad de una referencia a un objeto más flexible o sofisticada que un puntero.  - Proxy Remoto: un representante para un objeto que se encuentra en otro espacio de direcciones.  - Proxy Virtual: Crea objetos costosos por encargo (ej.: ImageProxy).  - Proxy de Protección: Controla el acceso al objeto original (permisos de acceso).  - Referencia inteligente: sustituto de un puntero, que lleva a cabo operaciones adicionales cuando se accede a un objeto.    Ejemplo de un proxy virtual:  class ImageProxy : public Graphic { // la clase proxy  public:  ImageProxy(const char\* imageFile);  virtual ~ImageProxy();  virtual void Draw(const Point& at);  virtual void HandleMouse(Event& event);  virtual const Point& GetExtent();  virtual void Load(istream& from);  virtual void Save(ostream& to);  protected:  Image\* GetImage();  private:  Image\* \_image;  Point \_extent;  char\* \_fileName;  };  Adapter  Propósito:  -Convierte el interfaz de una clase en otro que espera el cliente.  -El adapter permite trabajar juntas a clases que de otra forma no podrían por  tener interfaces incompatibles.                                                                                                                                                                                                      Estructura:      \*  Patrones de comportamiento:  - Tratan sobre algoritmos y la asignación de responsabilidades entre objetos.  -  Describen no sólo patrones de clases y objetos, sino patrones de comunicación entre ellos.  - Caracterizan un flujo de control complejo, difícil de seguir en tiempo de ejecución.  - Permiten que el diseñador se concentre sólo en cómo interconectar objetos.  Iterator  Proporciona un medio de acceder a los elementos de un contenedor secuencialmente sin exponer su representación  interna    Motivación.    - Antes de instanciar el ListIterator, se ha de proporcionar la lista.  - Una vez que se tiene la instancia, se puede acceder a los elementos de la lista secuencialmente.  - Separar el mecanismo de recorrido del objeto lista, nos permite definir iteradores que implementen distintas estrategias.  Uso del iterador.  void PrintEmployees (Iterator<Employee\*>& i) {  for (i.First(); !i.IsDone(); i.Next()) {  i.CurrentItem()->Print();  }  }  List<Employee\*>\* employees;  // ...  ListIterator<Employee\*> forward(employees);  ReverseListIterator<Employee\*> backward(employees);  PrintEmployees(forward);  PrintEmployees(backward);  Template Method  Propósito:  - Definir el esqueleto de un algoritmo, delegando en las subclases alguno de sus pasos.  - Permite que las subclases cambien pasos de un algoritmo sin cambiar su estructura.  Aplicabilidad.  -Para implementar las partes de un algoritmo que no cambian, y dejar que las subclases implementen el comportamiento que puede variar.  - Cuando el comportamiento repetido de varias subclases debería factorizarse y localizarse en una clase común, para evitar código duplicado.  - Para controlar las extensiones de las subclases.  MVC  El modelo–vista–controlador (MVC) es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones, donde se potencie la facilidad de mantenimiento, reutilización del código y la separación de conceptos.  Su fundamento es la separación del código de nuestras aplicaciones entre capas diferentes, como son:  Modelos.  Es la capa donde se trabaja con los datos, por tanto contendrá mecanismos para acceder a la información y también para actualizar su estado. Los datos los tendremos habitualmente en una base de datos, por lo que en los modelos tendremos todas las funciones que accederán a las tablas y harán los correspondientes selects, updates,  inserts, etc  Vistas.  Las vistas, como su nombre nos hace entender, contienen el código de nuestra aplicación que va a producir la visualización de  las interfaces de usuario, o sea, el código que nos permitirá enderezar los estados de nuestra aplicación en HTML.  En las vistas nada más tenemos los códigos HTML y PHP que nos permite mostrar la salida.  En la vista generalmente trabajamos con los datos, sin embargo, no se realiza un acceso directo a éstos. Las vistas requerirán los datos a los modelos y ellas se generarán la salida, tal como nuestra aplicación requiera.  Controladores.  Contiene el código necesario para responder a las acciones que se solicitan en la aplicación, como visualizar un elemento, realizar una compra, una búsqueda de información, etc.  En realidad es una capa que sirve de enlace entre las vistas y los modelos, respondiendo a los mecanismos  que puedan requerirse para implementar las necesidades de nuestra aplicación. Sin embargo, su responsabilidad no  es manipular directamente datos, ni mostrar ningún tipo de salida, sino servir de enlace entre los modelos y las vistas  para implementar las diversas necesidades del desarrollo    ¿CÓMO FUNCIONA Y PARTICULARMENTE SUS COMPONENTES?    Estructura de MVC  Como se puede ver, los controladores, con su lógica de negocio, hacen de puente entre los modelos y las vistas. Pero además en  algunos casos los modelos pueden enviar datos a las vistas. En la ilustración 1 se verá paso a paso cómo sería el flujo de trabajo  característico en un esquema MVC.  1.      El usuario realiza una solicitud a nuestro sitio web. Generalmente estará desencadenada por acceder a una página  de nuestro sitio. Esa solicitud le llega al controlador.  2.    El controlador comunica tanto con modelos como con vistas. A los modelos les solicita datos o les manda realizar actualizaciones de los datos. A las vistas les solicita la salida correspondiente, una vez se hayan realizado las operaciones pertinentes según la lógica del negocio.  3.    Para producir la salida, en ocasiones las vistas pueden solicitar más información a los modelos. En ocasiones, el controlador  será el responsable de solicitar todos los datos a los modelos y de enviarlos a las vistas, haciendo de puente entre unos y otros.  4.       Las vistas envían al usuario la salida. Aunque en ocasiones esa salida puede ir de vuelta al controlador y sería éste el que  hace finalmente el envío al cliente.  NOTA:   La “Lógica de negocio” es un conjunto de reglas que se siguen en el software para reaccionar ante distintas situaciones.  En una aplicación el usuario se comunica con el sistema por medio de una interfaz, pero cuando acciona esa interfaz para   realizar acciones con el programa, se ejecutan una serie de procesos que se conocen como la lógica del negocio.    ¿Qué frameworks existen para desarrollo de aplicaciones con este patrón?  Un framework de aplicaciones web es un tipo de framework que permite el desarrollo de sitios web  dinámicos, web services  (servicios web) y aplicaciones web. El propósito de este tipo de framework es permitir a los desarrolladores construir  aplicaciones web y centrarse en los aspectos interesantes, aliviando la típica tarea repetitiva asociada con patrones comunes de  desarrollo web.     |  |  | | --- | --- | |  | | | [Ruby on Rails](http://www.linuxlinks.com/article/20120525000539219/RubyonRails.html) | Framework MVC basado en Ruby, orientado al desarrollo de aplicaciones web | | [CodeIgniter](http://www.linuxlinks.com/article/20120525000531497/CodeIgniter.html) | Poderoso framework PHP liviano y rápido | | [Kohana](http://kohanaframework.org/) | Un fork de CodeIgniter, Gracias a Samuel por mencionarlo en los comentarios. | | [Django](http://www.linuxlinks.com/article/20120525000545879/Django.html) | Framework Python que promueve el desarrollo rápido y el diseño limpio | | [CakePHP](http://www.linuxlinks.com/article/20120525000252509/CakePHP.html) | Framework MVC para PHP de desarrollo rápido | | [Zend Framework](http://www.linuxlinks.com/article/20120525000536311/ZendFramework.html) | Framework para PHP 5, simple, claro y open-source | | [Yii](http://www.linuxlinks.com/article/2012052500054269/Yii.html) | Framework PHP de alto rendimiento basado en componentes | | [Pylons](http://www.linuxlinks.com/article/2012052500055227/Pylons.html) | Framework web para Python que enfatiza la flexibilidad y el desarrollo rápido | | [Catalyst](http://www.linuxlinks.com/article/20120525000602635/Catalyst.html) | Framework para aplicaciones web MVC elegante | | [Symfony](http://www.linuxlinks.com/article/20120525000534344/Symfony.html) | Framework full-stack | |