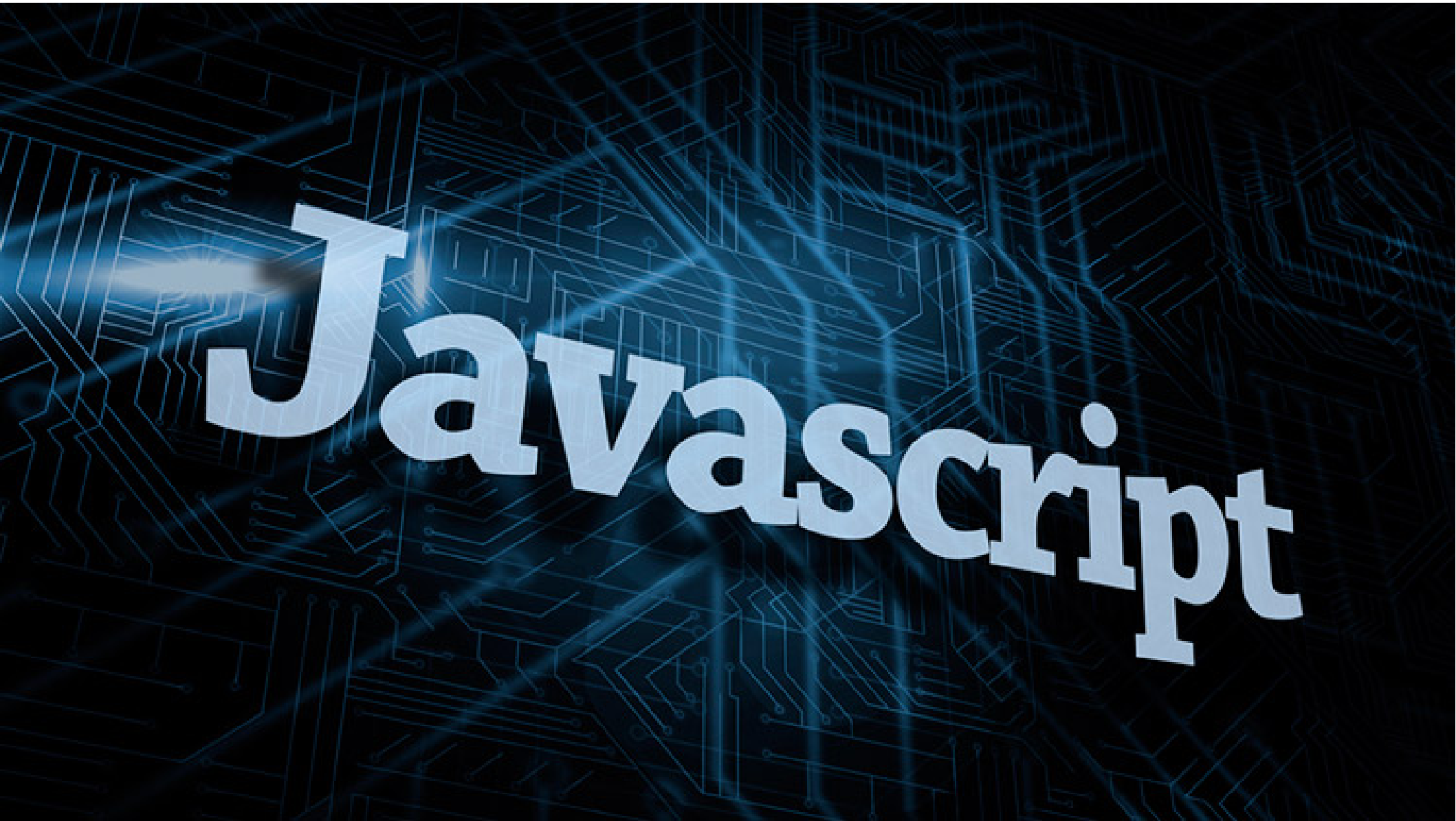
CURSO DESAROLLO WEB CON MEAN

WEB FULL STACK DEVELOPER

Germán Caballero Rodríguez
germanux@gmail.com

Programación avanzada con JS – II
Funciones y patrones



INDICE

Funciones en JavaScript

Funciones variádicas

- Un número indefinido de parámetros pueden ser pasados a la función.
- La función puede acceder a ellos a través de los parámetros o también a través del objeto local **arguments**.

Funciones en JavaScript

Funciones variádicas

- Las funciones variádicas también pueden ser creadas usando el método .apply().
- El método apply() invoca una determinada función asignando explícitamente el objeto this y un array o similar (array like object) como parámetros (argumentos) para dicha función.
    - fun.apply(thisArg[, argsArray])
  - **ThisArg**: El valor del objeto this a utilizar dentro de la llamada a fun.
  - **ArgsArray**: Un objeto similar a un array (array like object), que contiene los parámetro

Funciones en JavaScript

Funciones como métodos

- A diferencia de muchos lenguajes orientados a objetos, no hay distinción entre la definición de función y la definición de método.
- Más bien, la distinción se produce durante la llamada a la función; una función puede ser llamada como un método.
- Cuando una función es llamada como un método de un objeto, la palabra clave this, que es una variable local a la función, representa al objeto que invocó dicha función.
- Esto se puede hacer con **.call()** como ya vimos.

Funciones en JavaScript

Funciones como objetos

- A diferencia de muchos lenguajes orientados a objetos, una función en JS es un tipo de Objeto.
- Cada función en JavaScript es actualmente un objeto **Function**.
- Como todos los demás objetos, Function se puede crear utilizando la instrucción new:
  - **new Function ([arg1[, arg2[, ... argN]],] cuerpo-de-la-función)**

Funciones en JavaScript

Funciones como objetos

- **arg1, arg2, ... argN**
  - Nombres para ser utilizados por la función como nombres formales de argumento.
  - Cada uno de ellos debe ser una cadena que corresponde a un identificador válido JavaScript o una lista de esas cadenas separadas por una coma; por ejemplo "x", "elValor", o "a,b".
- **cuerpo-de-la-función**
  - Una cadena que contiene las instrucciones del código JavaScript y que conforma el cuerpo de la función.
- Invocar el constructor de la Function como una función (sin usar el operador new) tiene el mismo efecto que invocarla como un constructor.

Funciones en JavaScript

Funciones como objetos

- El código siguiente crea un objeto Function que toma dos argumentos.

var multiplicar = new Function("x", "y", "return x * y");

- Este código asigna una función a la variable multiplicar.

Funciones en JavaScript

Funciones como objetos

- Para llamar al objeto Function, puedes especificar el nombre de la variable como si fuera una función

```
var laRespuesta = multiplicar(7, 6);

var miEdad = 50;


if (miEdad >= 39) {
  miEdad = multiplicar(miEdad, .5);
}
```

Funciones en JavaScript

Funciones como objetos

- Los objectos Function creados con el constructor de la Function son evaluados cada vez que son usados.
- Esto no es tan eficiente como declarar una función y llamarla después dentro de tu código, porque las funciones declaradas se pasan una sola vez.

Funciones callbacks

- Una devolución de llamada o retrollamada (en inglés: callback) es una función "A" que se usa como argumento de otra función "B".
- Cuando se llama a "B", ésta ejecuta "A". Para conseguirlo, usualmente lo que se pasa a "B" es el puntero a "A".
- Esto permite desarrollar capas de abstracción de código genérico a bajo nivel que pueden ser llamadas desde una subrutina (o función) definida en una capa de mayor nivel.

Funciones callbacks

- Usualmente, el código de alto-nivel inicia con el llamado de alguna función, definida a bajo-nivel, pasando a esta un puntero, o un puntero inteligente (conocido como handle), de alguna función.
- Mientras la función de bajo-nivel se ejecuta, esta puede ejecutar la función pasada como puntero para realizar alguna tarea.

Funciones callbacks

- En otro escenario, las funciones de bajo nivel registran las funciones pasadas como un handle y luego pueden ser usadas de modo asincrónico.

Funciones callbacks

- Una retrollamada puede ser usada como una aproximación simple al polimorfismo y a la programación genérica, donde el comportamiento de una función puede ser dinámicamente determinado por el paso punteros a funciones o handles a funciones de bajo nivel que aunque realicen tareas diferentes los argumentos sean compatibles entre sí.
- Esta es una técnica de mucha importancia por lo que se la llama código reutilizable.

Uso de funciones callbacks

- Una retrollamada puede ser usada como una aproximación simple al polimorfismo y a la programación genérica, donde el comportamiento de una función puede ser dinámicamente determinado por el paso punteros a funciones o handles a funciones de bajo nivel que aunque realicen tareas diferentes los argumentos sean compatibles entre sí.
- Esta es una técnica de mucha importancia por lo que se la llama código reutilizable.

Uso de funciones callbacks

- Se puede considerar como ejemplo el problema de realizar varias operaciones arbitrarias en una lista.
- Una opción puede ser iterar sobre la lista, o también realizar alguna operación sobre cada uno de los elementos de la lista.
- En la práctica, la solución más común, pero no ideal, es utilizar iteradores como un bucle for) que deberá duplicarse en cada lugar del código donde sea necesario.
- Más aún, si la lista es actualizada por un proceso asíncrono (por ejemplo, si un elemento es añadido o eliminado), el iterador podría corromperse durante el paso a través de la lista.

Uso de funciones callbacks

- Una alternativa podría ser crear una nueva biblioteca de funciones que ejecute la tarea deseada con la sincronización apropiada en cada caso.
- Esta propuesta aún requiere que cada nueva función de la biblioteca contenga el código para ir a través de la lista.
- Esta solución no es aceptable para bibliotecas genéricas que tengan como objetivo varias aplicaciones; el desarrollador de la biblioteca no puede anticiparse a las necesidades de cada aplicación, y el desarrollador de las aplicaciones no debería necesitar conocer los detalles de la implementación de la biblioteca.

Uso de funciones callbacks

- En este caso las retrollamadas resuelven estos problemas.
- Un procedimiento es escribir el paso a través de una lista que provee a la aplicación del código para ir a través de la lista y operando sobre cada elemento.
- Existe una clara distinción entre la biblioteca y la aplicación sin sacrificar la flexibilidad.
- Una retrollamada puede también considerarse un tipo de rutina enlazada por referencia.

Uso de funciones callbacks

- Otro uso es en la señalización de errores.
- Hemos visto ejemplos de funciones callbacks que son llamadas cuando ha ocurrido un error en una transacción en bb.dd., o cuando hay un error en una solicitud de posición GPS.

Patrón Modelo Vista Controlador

- Siguiendo con la secuencia lógica de tu aprendizaje de Javascript, llegarás en este punto al MVC.
- Son las siglas de Modelo, Vista y Controlador y se trata de un paradigma de programación que se usa en lenguajes donde se tiene que trabajar con interfaces gráficas, como es el caso de la Web.
- Propone la separación del código de las aplicaciones por responsabilidades.

Patrón Modelo Vista Controlador

- Los modelos se encargan de trabajar con los datos de la aplicación, las vistas con la presentación y los controladores hacen de conexión entre vistas y modelos.
- MVC no es algo específico de Javascript, sino que lo encontramos en lenguajes del lado del servidor como PHP o incluso en lenguajes de propósito general como es Java.

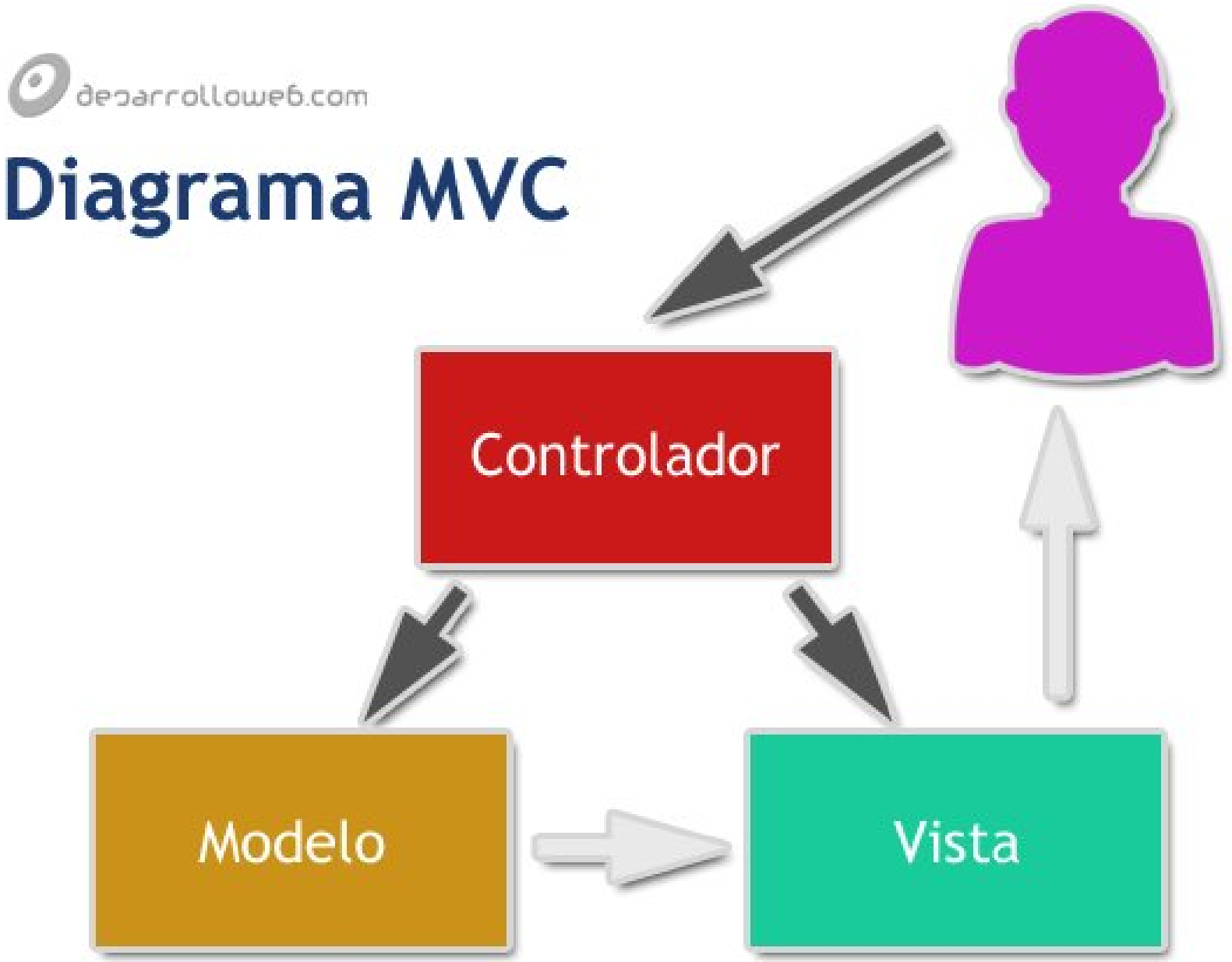
Patrón Modelo Vista Controlador

- Trabajar con paradigmas como MVC es fundamental en el mundo de las aplicaciones web, porque nos permite organizar mejor nuestro código, facilitando el mantenimiento de las aplicaciones.
- Existen diversas librerías para realizar MVC en Javascript, entre las más populares están BackboneJS, EmberJS, AngularJS, KnockoutJS, NodeJS, etc.

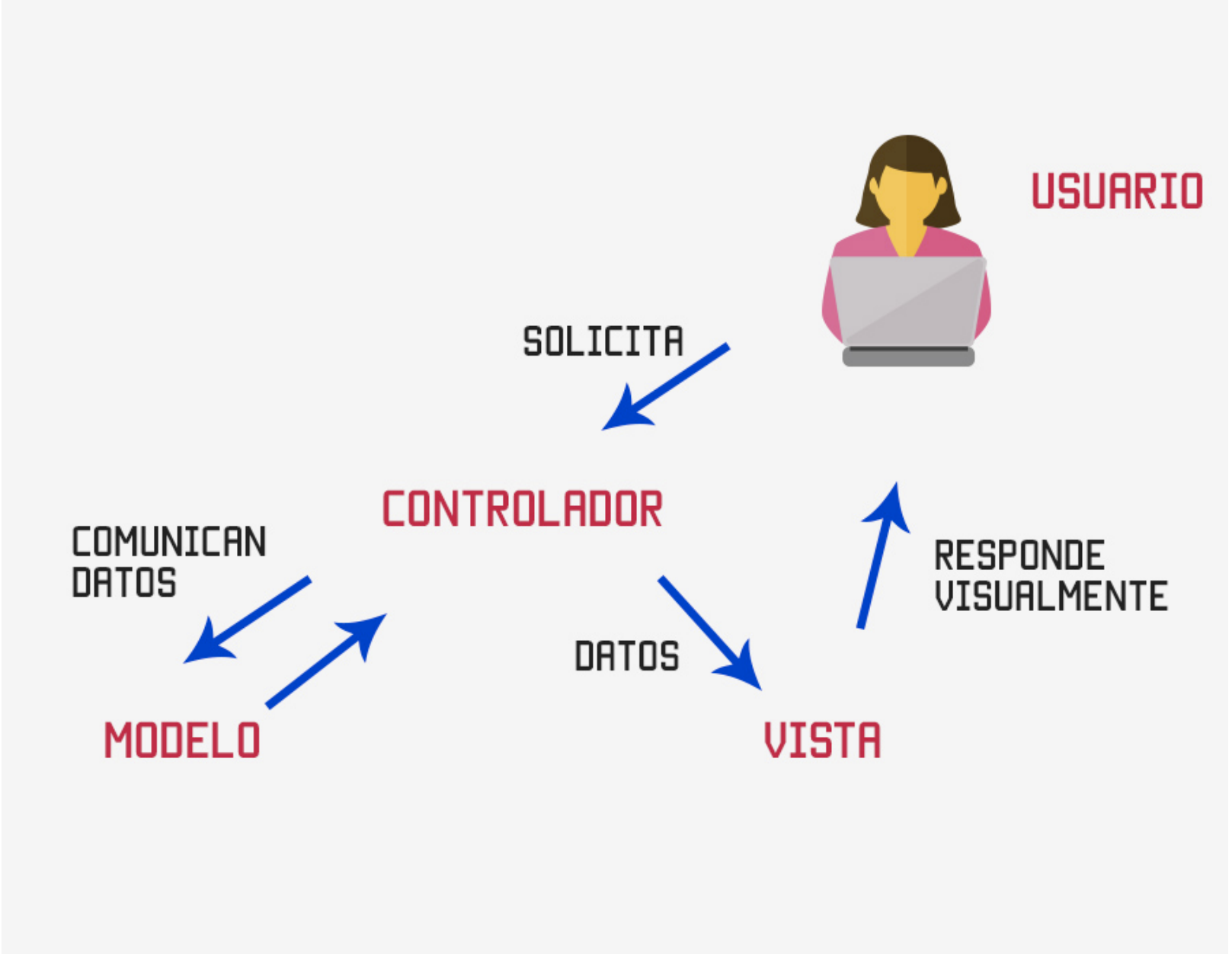
Patrón Modelo Vista Controlador

- 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.
- Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario.

Patrón Modelo Vista Controlador



Patrón Modelo Vista Controlador



Patrón Modelo Vista Controlador

- Este patrón de arquitectura de software se basa en las ideas de **reutilización de código** y la **separación de conceptos**, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento:
  - **La reutilización de código:** se refiere al comportamiento y a las técnicas que garantizan que una parte o la totalidad de un programa informático existente se pueda emplear en la construcción de otro programa.
  - **La separación de conceptos:** es un principio de diseño para separar un programa informático en secciones distintas, tal que cada sección enfoca un interés delimitado.

Patrón Modelo Vista Controlador

HOW MODEL VIEW CONTROLLER DESIGN WORKS

Upwork

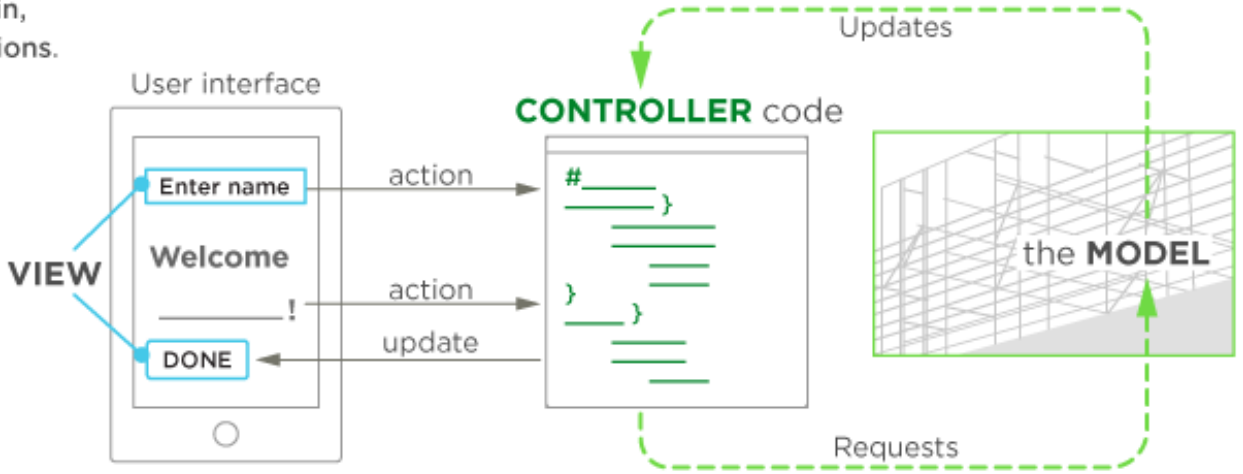
The MVC approach allows for clear separation of a web application's user interface (UI) code ("View") from its data and business logic ("Model") with scripts that send requests and updates between the two ("Controller"). Separating objects into these three bins allows for clean code that's easy to maintain.

Each object should be written for **one bin, and one bin only—no overlapping functions.**

MODEL
data and server-side applications

VIEW
format and layout of data (UI)

CONTROLLER
communicates between the two



Patrón Modelo Vista Controlador

- **El Modelo:**
  - Es la representación de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio).
  - Envía a la 'vista' aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario).
  - Las peticiones de acceso o manipulación de información llegan al 'modelo' a través del 'controlador'.

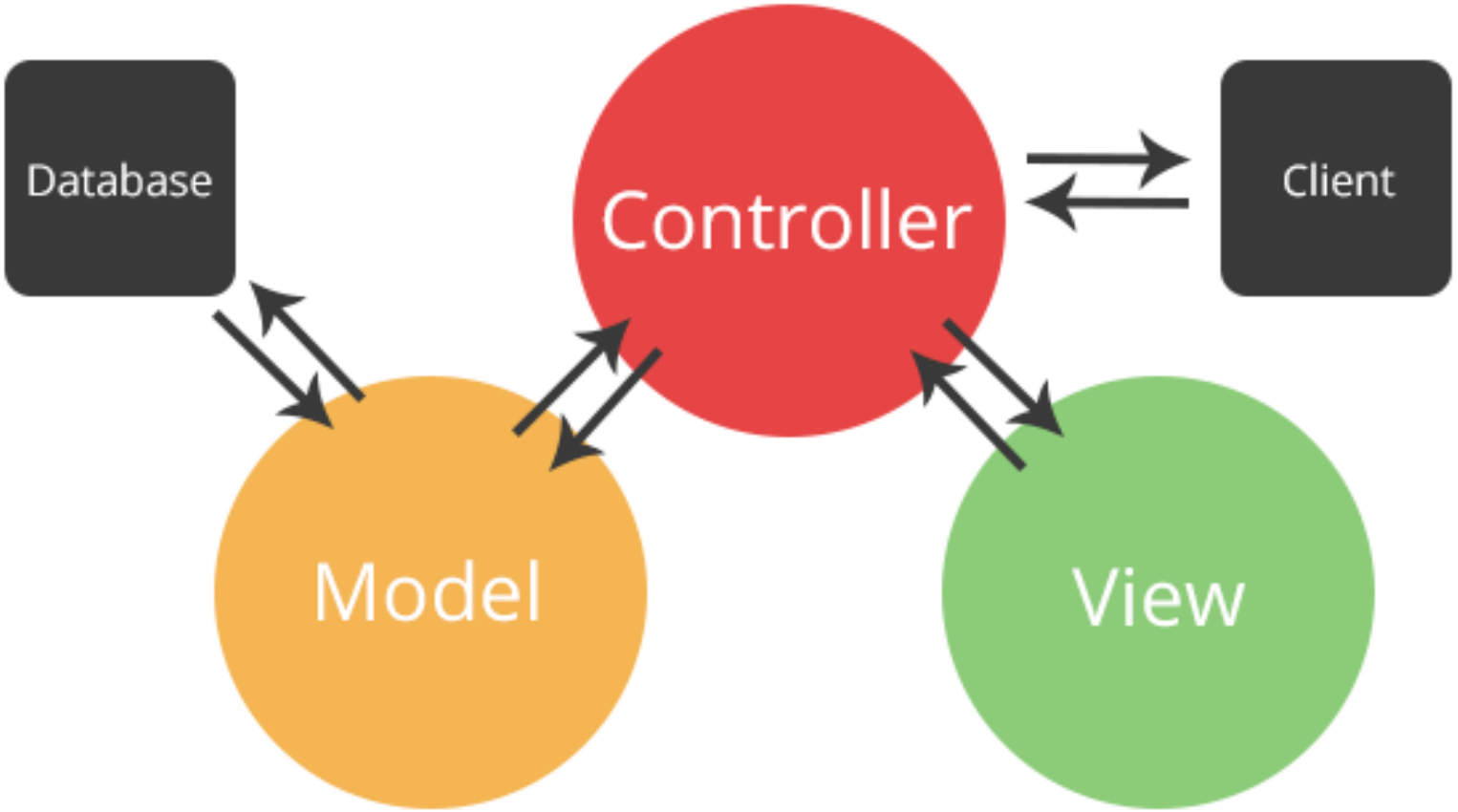
Patrón Modelo Vista Controlador

- **El Controlador:**
  - Responde a eventos (usualmente acciones del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento o un registro en una base de datos).
  - También puede enviar comandos a su 'vista' asociada si se solicita un cambio en la forma en que se presenta el 'modelo' (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos), por tanto se podría decir que **el 'controlador' hace de intermediario entre la 'vista' y el 'modelo'**

Patrón Modelo Vista Controlador

- **La Vista:**
  - Presenta el 'modelo' (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz de usuario) por tanto requiere de dicho 'modelo' la información que debe representar como salida.

Patrón Modelo Vista Controlador



AtomsNetwork

Patrón Modelo Vista Controlador

Interacción de los componentes

- El flujo de control en el MVC que se sigue generalmente (no siempre) 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.

Patrón Modelo Vista Controlador

Interacción de los componentes

- El flujo de control en el MVC:
  - 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.

Patrón Modelo Vista Controlador

Interacción de los componentes

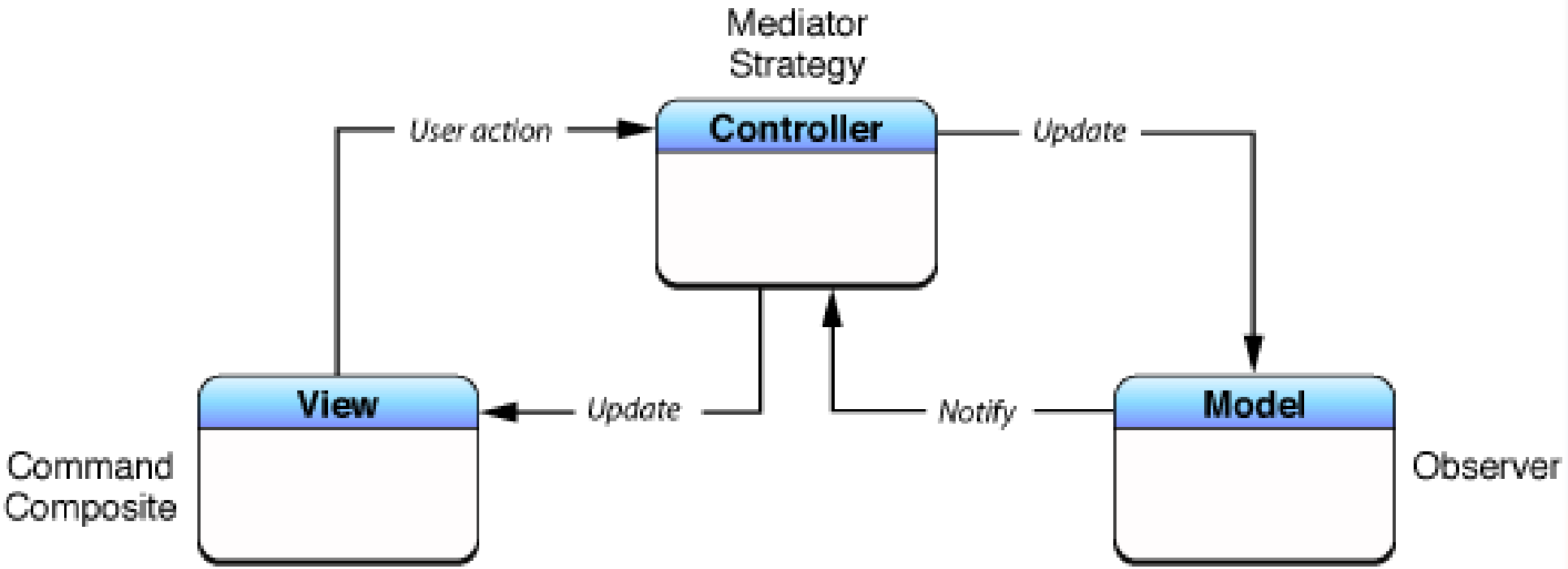
- El flujo de control en el MVC:
  - 4)
    - El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario.
    - La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo (por ejemplo, produce un 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.

Patrón Modelo Vista Controlador

Interacción de los componentes

- El flujo de control en el MVC:
  - 5)
    - La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente....

Patrón Modelo Vista Controlador



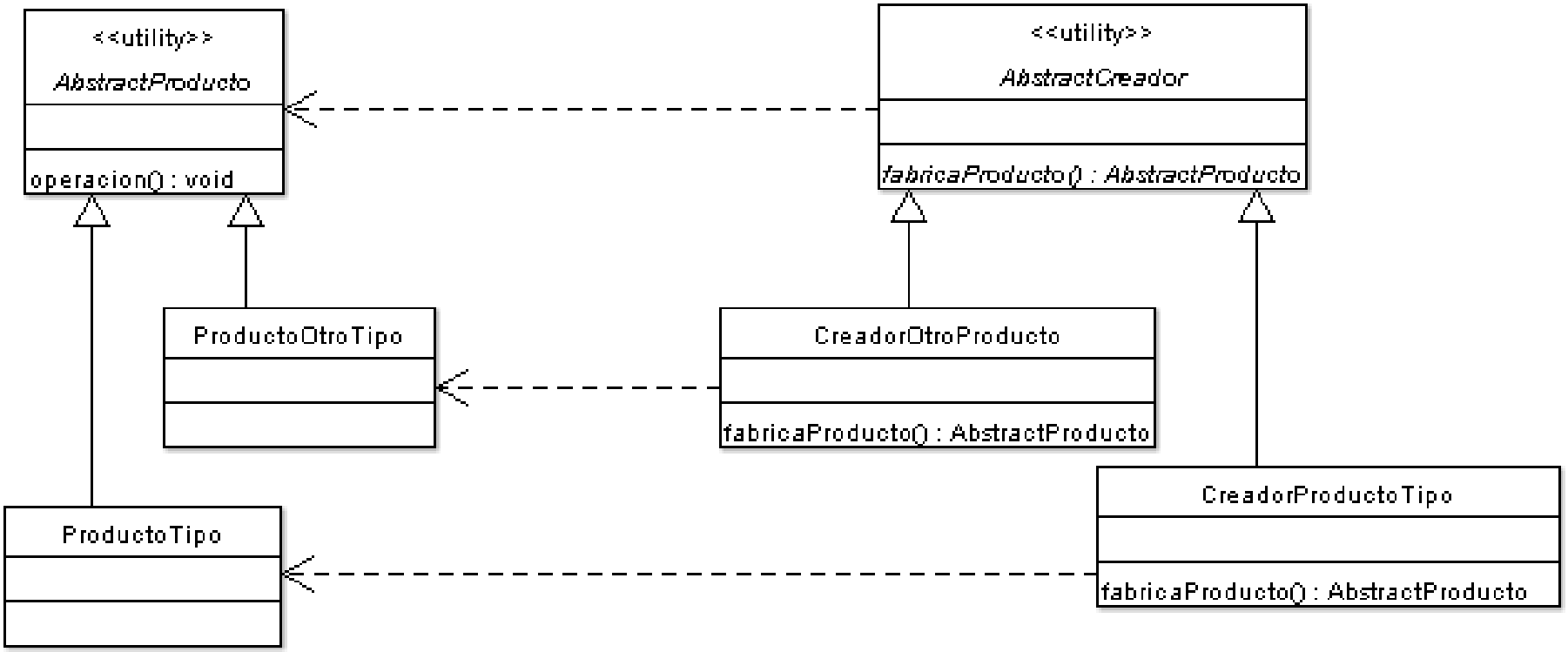
Patrón Factoría

- En diseño de software, el patrón de diseño Factory Method consiste en utilizar una clase constructora abstracta con unos cuantos métodos definidos y otro(s) abstracto(s): el dedicado a la construcción de objetos de un subtipo de un tipo determinado.
- Es una simplificación del Abstract Factory, en la que la clase abstracta tiene métodos concretos que usan algunos de los abstractos; según usemos una u otra hija de esta clase abstracta, tendremos uno u otro comportamiento.

Patrón Factoría

- Las clases principales en este patrón son el creador y el producto.
- El creador necesita crear instancias de productos, pero el tipo concreto de producto no debe ser forzado en las subclases del creador, porque las posibles subclases del creador deben poder especificar subclases del producto para utilizar.

Patrón Factoría



Patrón Factoría

- La solución para esto es hacer un método abstracto (el método de la fábrica) que se define en el creador.
- Este método abstracto se define para que devuelva un producto.
- Las subclases del creador pueden sobrescribir este método para devolver subclases apropiadas del producto...

Patrón Factoría

1) Ejemplo de código (en Java)

```
abstract class Creator{
    // Definimos método abstracto
    public abstract Product factoryMethod();
}
```

Patrón Factoría

1) Ejemplo de código (en Java)

Ahora definimos el creador concreto:

```
public class ConcreteCreator extends Creator{
    public Product factoryMethod() {
        return new ConcreteProduct();
    }
}
```

Patrón Factoría

1) Ejemplo de código (en Java)

Definimos el producto y su implementación concreta:

```
public interface Product{
    public void operacion();
}
public class ConcreteProduct implements Product{
    public void operacion(){
        System.out.println("Una operación de
este producto");
    }
}
```

Patrón Factoría

1) Ejemplo de código (en JavaScript)

Ejemplo de uso:

```
public static void main(String args[]){
    Creator aCreator;
    aCreator = new ConcreteCreator();
    Product producto = aCreator.factoryMethod();
    producto.operacion();
}
```

Patrón Factoría

1) Ejemplo de código (en JavaScript)

```
function crearCoche(marca, modelo, anyo){
    var o = new Object();
    o.marca = marca;
    o.modelo = modelo;
    o.anyo = anyo;
    o.returnMarca = function(){
        alert(this.marca);
    };
    return o;
}
var coche1 = crearCoche("Seat", "Ibiza", 2012);
var coche2 = crearCoche("Opel", "Astra", 2002);
```

Patrón Factoría

2) Ejemplo de código (en JavaScript)

```
function returnMarca(){
    alert(this.marca);
}
function crearCoche(marca, modelo, anyo){
    var o = new Object();
    o.marca = marca;
    o.modelo = modelo;
    o.anyo = anyo;
    o.returnMarca = returnMarca;
    return o;
}
var coche1 = crearCoche("Seat", "Ibiza", 2012);
var coche2 = crearCoche("Opel", "Astra", 2002);
```

Patrón Factoría

3) Ejemplo de código (en JavaScript)

```
function CrearCoche(marca, modelo, anyo){
    this.marca = marca;
    this.modelo = modelo;
    this.anyo = anyo;
    this.returnMarca = function(){
        alert(this.marca);
    };
}
var coche1 = new CrearCoche("Seat", "Ibiza", 2012);
var coche2 = new CrearCoche("Opel", "Astra", 2002);
```

Patrón Factoría

4) Ejemplo de código (en JavaScript)

```
// Types.js - Constructors used behind the scenes

// A constructor for defining new cars
function Car( options ) {

  // some defaults
  this.doors = options.doors || 4;
  this.state = options.state || "brand new";
  this.color = options.color || "silver";

}

// A constructor for defining new trucks
function Truck( options){

  this.state = options.state || "used";
  this.wheelSize = options.wheelSize || "large";
  this.color = options.color || "blue";
}
```

Patrón Factoría

4) Ejemplo de código (en JavaScript)

```
// FactoryExample.js

// Define a skeleton vehicle factory
function VehicleFactory() {}

// Define the prototypes and utilities for this factory

// Our default vehicleClass is Car
VehicleFactory.prototype.vehicleClass = Car;

// Our Factory method for creating new Vehicle instances
VehicleFactory.prototype.createVehicle = function ( options ) {

  switch(options.vehicleType){
    case "car":
      this.vehicleClass = Car;
      break;
    case "truck":
      this.vehicleClass = Truck;
      break;
    //defaults to VehicleFactory.prototype.vehicleClass (Car)
  }

  return new this.vehicleClass( options );

};
```

Patrón Factoría

4) Ejemplo de código (en JavaScript)

```
// Create an instance of our factory that makes cars
var carFactory = new VehicleFactory();
var car = carFactory.createVehicle( {
            vehicleType: "car",
            color: "yellow",
            doors: 6 } );

// Test to confirm our car was created using the vehicleClass/prototype Car

// Outputs: true
console.log( car instanceof Car );

// Outputs: Car object of color "yellow", doors: 6 in a "brand new" state
console.log( car );
```

Patrón Factoría

4) Ejemplo de código (en JavaScript)

```
var movingTruck = carFactory.createVehicle( {
                    vehicleType: "truck",
                    state: "like new",
                    color: "red",
                    wheelSize: "small" } );

// Test to confirm our truck was created with the vehicleClass/prototype Truck

// Outputs: true
console.log( movingTruck instanceof Truck );

// Outputs: Truck object of color "red", a "like new" state
// and a "small" wheelSize
console.log( movingTruck );
```