ENTORNOS DE DESARROLLO

ciclos formativos de grado superior de
desarrollo de aplicaciones multiplataforma

CAPÍTULO 1

Carlos Casado Iglesias

Ra-Ma®

DESARROLLO DE SOFTWARE

1. EL PROGRAMA INFORMÁTICO

Un **programa informático** es un conjunto de instrucciones que se ejecutan de manera secuencial con el objetivo de realizar una o varias tareas en un sistema.

Un programa informático interactúa con el sistema ejecutando las diferentes instrucciones del programa en la ALU, mediante instrucciones que sea capaz de entender.

Cada instrucción se divide en instrucciones más pequeñas que se ejecutarán individual y secuencialmente en cada ciclo del procesador. A esas instrucciones más pequeñas se las conoce por el nombre de microinstrucciones.

2. LENGUAJES DE PROGRAMACIÓN

Un lenguaje de programación es un conjunto de instrucciones, operadores y reglas de sintaxis y semánticas, que se ponen a disposición del programador para que éste pueda comunicarse con los dispositivos de hardware y software existentes.

El sistema sólo es capaz de entender código escrito en código máquina (1s y 0s), programar directamente en **código máquina** es una tarea ardua y tediosa.

El uso de un lenguaje de programación tiene el objetivo de facilitar la tarea a los programadores permitiendo escribir programas utilizando un mayor **nivel de abstracción** en el código.

2. LENGUAJES DE PROGRAMACIÓN

➢ NIVEL DE ABSTRACCIÓN

El nivel de abstracción de un lenguaje implica cuan alejado está del código máquina, cuanto más parecido sea a nuestro lenguaje y menos al código máquina, de mayor nivel será el lenguaje.

Bajo nivel: Sólo hay un lenguaje de primera generación, el **código máquina**.

Medio nivel: Tienen definidas una serie de instrucciones sencillas para trabajar con datos simples y posiciones de memoria. El lenguaje clave de éste nivel es el **lenguaje ensamblador**.

Alto nivel: La mayoría de los lenguajes que se utilizan hoy en día para programar aplicaciones son de alto nivel. Son muy cercanos a nuestro propio lenguaje y permiten la realización de patrones de diseño complejos como **Java** o **C#**. También nos encontramos en éste nivel de abstracción a los lenguajes destinados a un propósito específico, como podría ser el **SQL**.

2. LENGUAJES DE PROGRAMACIÓN

➢ FORMA DE EJECUCIÓN

Compilados: Un programa traductor (**compilador**) convierte el código fuente en código objeto para crear un programa ejecutable.

Interpretados: Ejecutan las instrucciones directamente, sin generar código objeto. La propia máquina se encargará de interpretarlo instrucción a instrucción.

Virtuales: Con un comportamiento similar al de los lenguajes compilados con una peculiaridad, el compilador no genera código objeto sino código ***bytecode***. Una **máquina virtual** se encargará de interpretar el ***bytecode*** para ejecutarlo en el sistema.

DESARROLLO DE SOFTWARE

2. LENGUAJES DE PROGRAMACIÓN

➢ PARADIGMA DE PROGRAMACIÓN

Imperativo: describe la programación como una secuencia de instrucciones que cambian el estado del programa, indicando cómo realizar una tarea.

Declarativo: especifica o declara un conjunto de premisas y condiciones para indicar qué es lo que hay que hacer y no necesariamente cómo hay que hacerlo.

Procedimental: el programa se divide en partes más pequeñas, llamadas funciones y procedimientos, que pueden comunicarse entre sí. Permite reutilizar código ya programado y solventa el problema de la programación *spaghetti*.

Orientado a objetos: encapsula el estado y las operaciones en objetos, creando una estructura de clases y objetos que emula un modelo del mundo real, donde los objetos realizan acciones e interactúan con otros objetos.

Funcional: evalúa el problema realizando funciones de manera recursiva, evita declarar datos haciendo hincapié en la composición de las funciones y en las interacciones entre ellas.

Lógico: define un conjunto de reglas lógicas para ser interpretadas mediante inferencias lógicas. Permite responder preguntas planteadas al sistema para resolver problemas.

3. OBTENCIÓN DE CÓDIGO EJECUTABLE

Código fuente: el código fuente de un programa informático es un conjunto de instrucciones escritas en un lenguaje de programación determinado. Es decir, es el código en el que nosotros escribimos nuestro programa.

Código objeto: el código objeto es el código resultante de **compilar** el código fuente. Si se trata de un lenguaje de programación compilado, el código objeto será código máquina, mientras que si se trata de un lenguaje de programación virtual, será código ***bytecode***.

Código ejecutable: el código ejecutable es el resultado obtenido de enlazar nuestro código objeto con las librerías. Este código ya es nuestro programa ejecutable, programa que se ejecutará directamente en nuestro sistema o sobre una máquina virtual en el caso de los lenguajes de programación virtuales.

3. OBTENCIÓN DE CÓDIGO EJECUTABLE

➢ FASES DE COMPILACIÓN

DESARROLLO DE SOFTWARE

4. PROCESOS DE DESARROLLO

El proceso más habitual en el desarrollo de un software es el modelo de desarrollo en cascada, el cual consta de 7 etapas.

Análisis: La fase de análisis define los requisitos del software que hay que desarrollar.

Diseño: En esta etapa se pretende determinar el funcionamiento de una forma global y general, sin entrar en detalles.

Codificación: La fase más obvia en el proceso de desarrollo de software es sin duda la codificación. Es más que evidente que una vez definido el software que hay que crear haya que programarlo.

Pruebas: Con una doble funcionalidad, las pruebas buscan confirmar que la codificación ha sido exitosa y que el software no contiene errores, a la vez que se comprueba que el software hace lo que debe hacer, que no necesariamente es lo mismo.

Documentación: Debe mostrar una información completa y de calidad que ilustre mediante los recursos más adecuados cómo manejar la aplicación. También se debe realizar una documentación técnica destinada a ser leída por los demás desarrolladores que trabajen en la aplicación.

Explotación: Una vez que tenemos nuestro software, hay que prepararlo para su distribución. Para ello se implementa el software en el sistema elegido o se prepara para que se implemente por sí solo de manera automática.

Mantenimiento: En esta fase del desarrollo de un software se arreglan los fallos o errores que suceden cuando el programa ya ha sido implementado en un sistema y se realizan las ampliaciones necesitadas o requeridas.

DESARROLLO DE SOFTWARE

5. ROLES QUE INTERACTÚAN EN EL DESARROLLO

Durante el proceso de desarrollo de un software interviene un personal con diferentes roles.

Analista de sistemas: Su objetivo consiste en realizar un estudio del sistema para dirigir el proyecto en una dirección que garantice las expectativas del cliente determinando el comportamiento del software. Participa en la etapa de **análisis**.

Diseñador de software: Nace como una evolución del analista y realiza, en función del análisis de un software, el diseño de la solución que hay que desarrollar. Participa en la etapa de **diseño**.

Analista programador: Aporta una visión general del proyecto más detallada diseñando una solución más amigable para la codificación y participando activamente en ella. Participa en las etapas de **diseño y codificación**.

Programador: Escribe el código fuente en función al estudio realizado por analistas y diseñadores. Participa en la etapa de **codificación**.

Arquitecto de software: Conoce e investiga los *frameworks* y tecnologías revisando que todo el procedimiento se lleva a cabo de la mejor forma y con los recursos más apropiados. Participa en las etapas de **análisis, diseño, documentación y explotación**.

DESARROLLO DE SOFTWARE

6. ARQUITECTURA DE SOFTWARE

La arquitectura de software es el diseño de nivel más alto de la estructura de un sistema, enfocándose más allá de los algoritmos y estructuras de datos. Es un conjunto de decisiones que definen a nivel de diseño los componentes computacionales y la interacción entre ellos para garantizar que el proyecto llegue a buen término.

Se encuentra principalmente vinculado a la programación orientada a objetos, aunque podemos encontrar arquitecturas de software en otros paradigmas.

Las diferentes técnicas de arquitectura de software se aplican mediante el uso de patrones o modelos de diseños que aportan soluciones a diferentes problemas estableciendo unos objetivos y unas restricciones para cada patrón.

6. ARQUITECTURA DE SOFTWARE

➢ PATRONES CREACIONALES

Fábrica abstracta: Se utiliza cuando se necesita crear diferentes objetos pertenecientes a la misma familia. Para ello, disponemos de una **factoría abstracta** que define las **interfaces** de las factorías concretas, que interpretan una factoría concreta, encargada de crear el **producto (objeto) concreto**.

Podemos observar la utilidad de éste patrón cuando necesitamos crear diferentes objetos con las mismas propiedades pero con diferentes valores. Cada grupo de valores pertenecería a una factoría concreta. De éste modo, en vez de tener diferentes instancias de la misma clase cambiando el valor de sus propiedades, hemos usado el **polimorfismo** para crear una instancia con unos valores específicos que definen de manera concreta al objeto.

Instancia única: Se utiliza para asegurar que solo pueda existir una única instancia de una clase, regulando para ello el acceso al **constructor**. Se deberá tener en cuenta la posibilidad de *multihilos* y controlar dicha eventualidad mediante la exclusión mutua.

6. ARQUITECTURA DE SOFTWARE

➢ PATRONES ESTRUCTURALES

Decorador: Se utiliza este patrón cuando necesitamos añadir de manera dinámica diferentes funcionalidades a un objeto.

Permite además retirar la funcionalidad si se necesita. Evitamos con este patrón definir cada funcionalidad mediante una clase heredada, utilizando para ello clases que implementan las funcionalidades necesitadas y que se asocian con la clase que necesita dicha funcionalidad.

Objeto compuesto: Mediante el uso de una **clase abstracta** o de un **interface**, generaremos jerarquías de objetos que efectúen la misma acción tanto a sí mismos como a los objetos hijos que contienen. De éste modo, se crean objetos complejos formados por otros más pequeños, aplicando la acción en todos ellos y consiguiendo que se comporten como si fuesen un único objeto. Para enlazar ésta jerarquía de objetos sencillos en el objeto compuesto, utilizamos una estructura de **árbol**.

Fachada: Nos permite crear una interfaz que **unifique el acceso** a los diferentes subsistemas, de modo que simplifica el acceso y lo hace más fácil de usar. Básicamente consiste en añadir un nivel adicional de abstracción donde para realizar cualquier operación de los subsistemas para los que hemos creado la fachada sólo necesitamos acceder a la fachada y será ésta quien se encargue de acceder a los subsistemas para realizar la operación.

6. ARQUITECTURA DE SOFTWARE

➢ PATRONES DE COMPORTAMIENTO

Estado: Nos permite definir un comportamiento diferente dependiendo del estado interno en que se encuentre un objeto. Es decir, mediante la invocación del mismo método, el objeto se comporta de modo diferente dependiendo del estado.

Podemos apreciar la utilidad de éste patrón cuando necesitamos manejar una conexión, en donde nuestro objeto se comportará de forma diferente cuando realicemos una petición de conexión dependiendo del estado en el que se encuentre actualmente la conexión.

Visitor: Pretende separar las operaciones de la estructura del objeto, para ello, se definen unas clases elemento con un método "*aceptar*" que recibirá al "*visitante*", teniendo un visitador por clase. De este modo, utilizamos las clases elemento para definir la estructura del objeto, y los visitantes para establecer los algoritmos y operaciones del objeto visitado.

El uso más evidente en donde se utiliza el patrón visitor es en los analizadores sintácticos o gramáticos como por ejemplo en las **expresiones regulares**.

Iterador: El patrón iterador establece una interfaz, cuyos métodos permiten acceder a un conjunto de objetos de una **colección**. Los métodos necesarios para recorrer la colección pueden variar dependiendo de las necesidades que tengamos y de lo que necesitemos hacer, pero es habitual crear o necesitar métodos que permitan acceder al primer elemento, obtener el elemento actual y saber si hemos llegado al final de la colección.

Cuando recorremos colecciones con estructuras de bucle como ***foreach***, se utiliza un **enumerador** que es una implementación del patrón iterador.

6. ARQUITECTURA DE SOFTWARE

➢ ANTIPATRONES

Código spaguetti: Código no metodizado. Cualquier cambio o movimiento en el código desmoronaba toda la aplicación, llena de saltos constantes a modo de llamadas o bucles.

Se le dio este nombre por el dibujo resultante de tomar un lápiz y dibujar líneas mostrando la vida del programa.

Flujo de lava: Grandes cantidades de código desordenado, módulos y añadidos ingentes que rompen la estructura natural del software.

Es habitual encontrarlo en software codificado bajo una mala gestión y no necesariamente fruto de un programador descuidado.

Martillo dorado/Varita mágica: El apego injustificado e irresponsable a un paradigma, a un lenguaje o a un *framework* concreto para solucionar todos los problemas.

Reinventar la rueda: Aparece cuando desarrollamos soluciones a problemas que ya tienen una solución.

Reinventar la rueda cuadrada: Aparece cuando reinventamos la rueda de un modo incorrecto o ineficiente.

Infierno de las dependencias: Depender de manera abusiva de las librerías y componentes de un entorno de desarrollo o plataforma. En muchas ocasiones esas dependencias nos generan problemas al actualizar la versión de la librería, como sucede hoy en día con algunos *ORM*.

Manejo de excepciones inútil: Si establecemos condicionales con el fin de evitar que surjan excepciones para lanzar manualmente una excepción, estamos utilizando un control de excepciones problemático.

Cadenas mágicas: Utilizar cadenas de caracteres explícitas en el código no es una práctica recomendada.

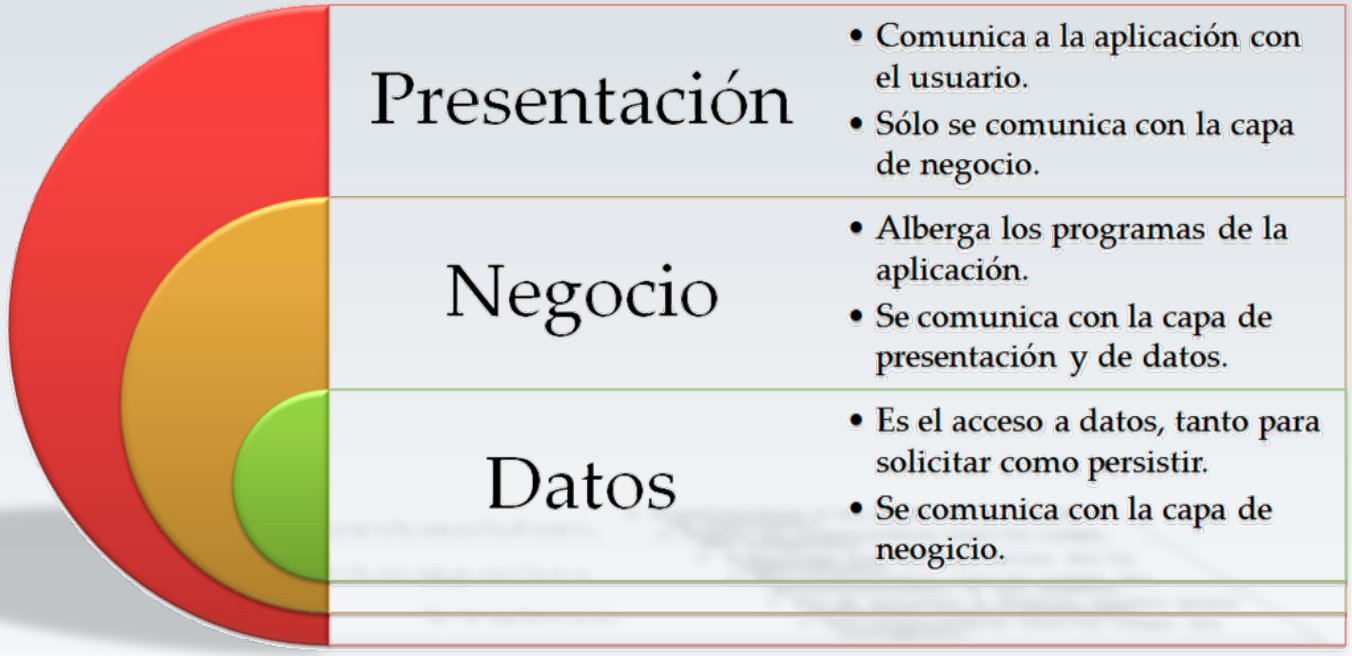
Copiar & Pegar: Siempre que a la hora de crear una nueva clase o método copiemos y peguemos código para ello, debemos tener en cuenta que, sin excepción, estamos haciendo algo mal.

DESARROLLO DE SOFTWARE

6. ARQUITECTURA DE SOFTWARE

➢ DESARROLLO EN TRES CAPAS

El desarrollo en capas nace de la necesidad de separar la lógica de la aplicación del diseño, separando a su vez los datos de la presentación al usuario.



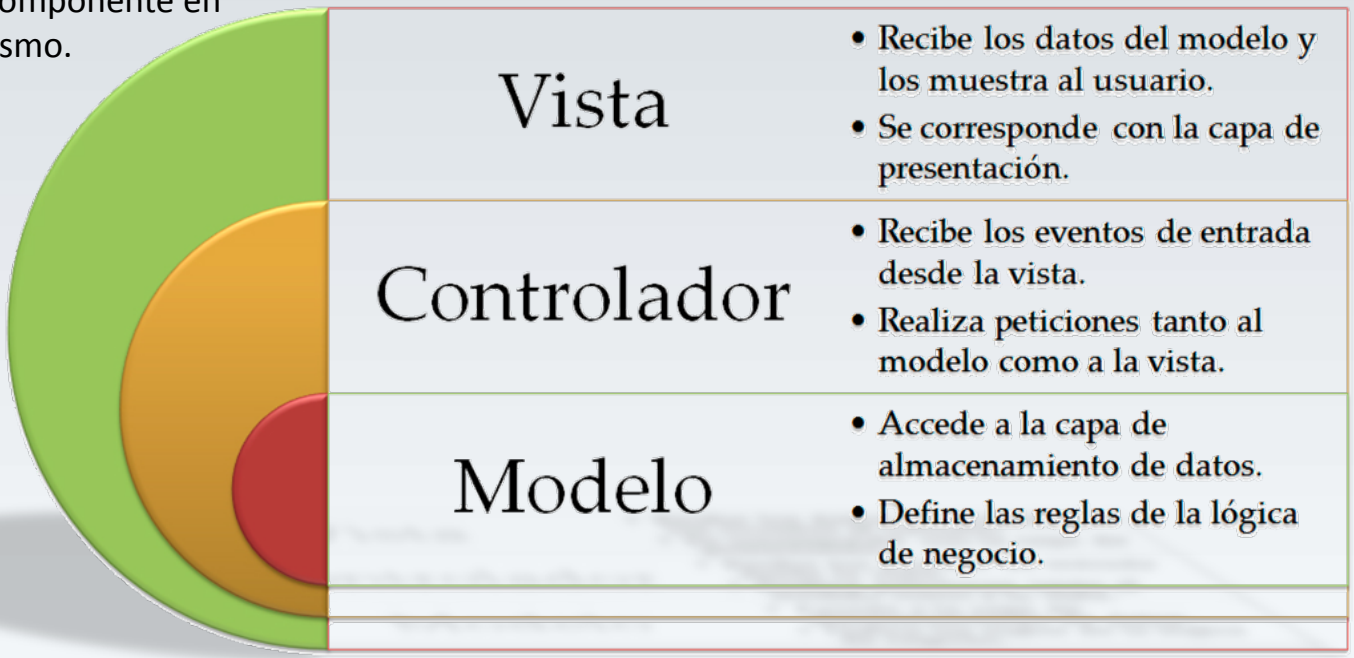
El desarrollo por capas no solo nos mejora y facilita la estructura de nuestro propio software, sino que nos aporta la posibilidad de interoperar con otros sistemas ajenos a nuestra aplicación.

6. ARQUITECTURA DE SOFTWARE

➢ DESARROLLO EN TRES CAPAS

Modelo Vista Controlador

El MVC define tres componentes para las capas del desarrollo del software, organiza el código mediante unas directrices específicas utilizando un criterio basado en la funcionalidad y no en las características del componente en sí mismo.



Se suele establecer una serie de "bindings" que enlacen diferentes componentes de la vista a propiedades y campos de las entidades.

6. ARQUITECTURA DE SOFTWARE

➢ DESARROLLO EN TRES CAPAS

Modelo Vista VistaModelo

EL MVVM parte de un concepto muy similar al modelo MVC. A diferencia del MVC, la vista del MVVM es un observador que se actualiza cuando cambia la información contenida en el VistaModelo.

Los eventos de la vista son recogidos por el controlador, pero a diferencia del MVC los datos de la vista son obtenidos y actualizados a través del VistaModelo, dejando al modelo como una mera representación de las entidades.