1 Desarrollo de software

OBJETIVOS DEL CAPÍTULO

- ✓ Reconocer los diferentes tipos de lenguajes de programación y las necesidades que cubren cada uno de ellos.
- ✓ Comprender el proceso de desarrollo de un software y las diferentes tareas que se deben realizar en las diferentes fases.
- ✓ Conocer las diferentes técnicas de arquitecturas de software, sus utilidades y las ventajas que ofrece cada una.

1.1 EL PROGRAMA INFORMÁTICO

Definición de 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 es creado por un programador en un lenguaje determinado, que será compilado y ejecutado por un sistema. Cuando un programa es llamado para ser ejecutado, el procesador ejecuta el código compilado del programa instrucción por instrucción.

Se podría llegar a decir también que un programa informático es software, pero no sería una definición muy acertada, pues un software comprende un conjunto de programas.

1.1.1 INTERACCIÓN CON EL SISTEMA

El mejor modo para comprender cómo un programa interactúa con un sistema es revisando la funcionalidad básica y el programa básico, irnos directamente al concepto de programa y de sistema en su mínima expresión. Para ello, vamos a revisar el funcionamiento de una única instrucción dentro del conocido simulador von Neumann.

Como hemos comentado brevemente, el procesador ejecutará las instrucciones una a una, pero no solo eso, para cada instrucción realizará una serie de microinstrucciones para llevarla a cabo.

Vamos a realizar el recorrido que efectúa una instrucción de un modo conceptual, nada técnico, profundizando más y más dentro de la interpretación que hace el sistema de nuestro programa. Imaginemos que tenemos un programa extremadamente sencillo que pide por teclado dos números y los suma. La instrucción que realizará la operación de nuestro programa se podría corresponder con la siguiente línea:

```
c = a + b;
```

El ordenador tendrá reservada una cantidad de posiciones de memoria definidas por el tipo de variable que se corresponden con las variables de nuestra instrucción. Es decir, nuestras variables "a", "b" y "c" tendrán unas posiciones de memoria definidas que es donde el sistema almacenará los valores de las variables.

No obstante, el procesador no puede ejecutar esa instrucción por sencilla que sea de un solo golpe, la ALU (*Aritmetic Logic Unit*) tiene un número muy limitado de operaciones que puede realizar, usualmente SUMAR y RESTAR.

Para esta demostración en concreto, vamos a definir nuestra siguiente máquina de von Neumann:

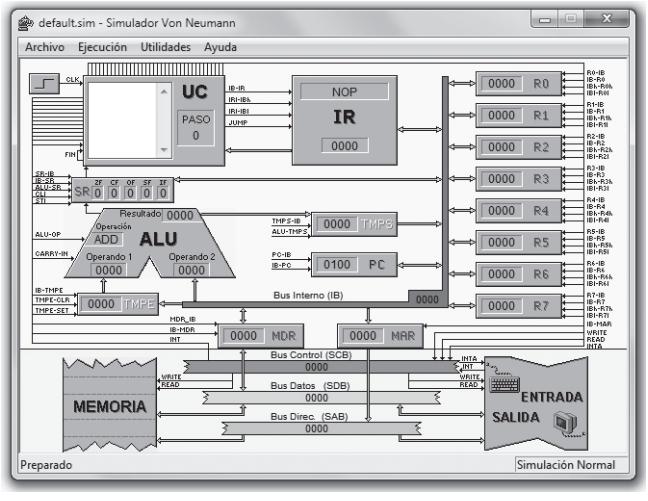


Figura 1.1. *Máquina de von Neumann*

Además, para nuestra demostración deberemos tener en cuenta una serie de reglas intrínsecas del sistema:

- La ALU solo puede realizar una operación a la vez.
- El registro temporal de la ALU, el *bus* y los registros solo pueden almacenar un dato a la vez.

Si definimos (para simplificar el modelo) que las posiciones de memoria de "a", "b" y "c" se corresponden con los registros R1, R2 y R3, las microinstrucciones que tendría que realizar nuestra máquina serían las siguientes:

```
R1 – Bus ; Bus – ALU-Temp ; R2 – Bus ; ALU_SUMAR ; ALU – Bus ; Bus – R3
```

Como es lógico, hoy en día, con los microprocesadores modernos, el funcionamiento, aunque muy similar en esencia, de cómo son interpretadas las instrucciones de nuestro programa por el sistema puede variar, sobre todo en lo que las reglas se refiere.

Sin embargo, hay cosas que no cambian, el programa se sigue almacenando en una memoria no volátil y se sigue ejecutando en la memoria de acceso aleatorio, al igual que todas las variables utilizadas.

La interacción con el sistema no es siempre una relación directa, no todos los programas tienen acceso libre y directo al hardware, es por ello que se definen los programas en dos clasificaciones generales: software de sistema y software de aplicación. Es el software de sistema el que se encarga de controlar y gestionar el hardware, así como de gestionar el software de aplicación, de hecho, en un software de sistema, como el sistema operativo, es en donde se ejecuta realmente el software de aplicación. Será el software de aplicación el que incorpore (gracias al compilador) las librerías necesarias para entenderse con el sistema operativo, y éste a su vez sería el que se comunicase con el hardware.

ACTIVIDADES 1.1

- Suponiendo la existencia de una operación en la ALU llamada DECREMENTAR, que utilizase el valor de la ALU-TEMP y le restase 1, especifique cuáles serían las microinstrucciones que se realizarían para multiplicar el contenido de R1 y R2 guardando el resultado en R3.

1.2 LENGUAJES DE PROGRAMACIÓN

Definición de lenguaje 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 idioma artificial que constituyen los operadores, instrucciones y reglas tiene el objetivo de facilitar la tarea de crear programas, permitiendo con un mayor nivel de abstracción realizar las mismas operaciones que se podrían realizar utilizando código máquina.

En un principio, todos los programas eran creados por el único código que el ordenador era capaz de entender: el código máquina, un conjunto de 0s y 1s de grandes proporciones. Este método de programación, aunque absolutamente efectivo y sin restricciones, convertía la tarea de programación en una labor sumamente tediosa, hasta que se tomó la solución de establecer un nombre a las secuencias de programación más frecuentes, estableciéndolas en posiciones de memoria concretas, a cada una de estas secuencias nominadas se las llamó instrucciones, y al conjunto de dichas instrucciones, lenguaje ensamblador.

Más adelante, empezaron a usar los ordenadores científicos de otras ramas, con muchos conocimientos de física o química, pero sin nociones de informática, por lo que les era sumamente complicado el uso del lenguaje ensamblador; como un modo de facilitar la tarea de programar, y no como un modo de facilitar el trabajo al programador informático, nace el concepto de lenguaje de alto nivel con FORTRAN (*FORmula TRANslation*) como primer debutante.

Los lenguajes de alto nivel son aquellos que elevan la abstracción del código máquina lo más posible, para que programar sea una tarea más liviana, entendible e intuitiva. No obstante, nunca hay que olvidar que, usemos el lenguaje que usemos, el compilador hará que de nuestro código solo lleguen 1s y 0s a la máquina.

1.2.1 CLASIFICACIÓN Y CARACTERÍSTICAS

La cantidad de lenguajes de programación es sencillamente abrumadora, cada uno con unas características y objetivos determinados, tal abrumador elenco de lenguajes de programación hace necesario establecer unos criterios para clasificarlos. Huelga decir que los criterios que clasifican los lenguajes de programación se corresponden con sus características principales.

Se pueden clasificar mediante una gran variedad de criterios, se podrían establecer hasta once criterios válidos diferentes con los que catalogar un lenguaje de programación. Algunos de dichos criterios pudieran ser redundantes, puesto que se encuentran incluidos explícitamente dentro de otros, como el determinismo o el propósito.

En este libro vamos a clasificar los lenguajes de programación siguiendo 3 criterios globales y reconocidos: el nivel de abstracción, la forma de ejecución y el paradigma.

Nivel de abstracción

Llamamos nivel de abstracción al modo en que los lenguajes se alejan del código máquina y se acercan cada vez más a un lenguaje similar a los que utilizamos diariamente para comunicarnos. Cuanto más alejado esté del código máquina, de mayor nivel será el lenguaje. Dicho de otro modo, podría verse el nivel de abstracción como la cantidad de "capas" de ocultación de código máquina que hay entre el código que escribimos y el código que la máquina ejecutará en último término.

Lenguajes de bajo nivel

- **Primera generación**: solo hay un lenguaje de primera generación: el código máquina. Cadenas interminables de secuencias de 1s y 0s que conforman operaciones que la máquina puede entender sin interpretación alguna.

Lenguajes de medio nivel

- **Segunda generación**: los lenguajes de segunda generación tienen definidas unas instrucciones para realizar operaciones sencillas con datos simples o posiciones de memoria. El lenguaje clave de la segunda generación es sin duda el lenguaje ensamblador.
- Aunque en principio pertenecen a la tercera generación, y por tanto serían lenguajes de alto nivel, algunos consideran a ciertos lenguajes de programación procedimental lenguajes de medio nivel, con el fin de establecerlos en una categoría algo inferior que los lenguajes de programación orientada a objetos, que aportan una mayor abstracción.

Lenguajes de alto nivel

- **Tercera generación**: la gran mayoría de los lenguajes de programación que se utilizan hoy en día pertenecen a este nivel de abstracción, en su mayoría, los lenguajes del paradigma de programación orientada a objetos, son lenguajes de propósito general que permiten un alto nivel de abstracción y una forma de programar mucho más entendible e intuitiva, donde algunas instrucciones parecen ser una traducción directa del lenguaje humano. Por ejemplo, nos podríamos encontrar una línea de código como ésta: *IF contador = 10 THEN STOP*. No parece que esta sentencia esté muy alejada de cómo expresaríamos en nuestro propio lenguaje *Si el contador es 10, entonces para*.
- **Cuarta generación**: son lenguajes creados con un propósito específico, al ser un lenguaje tan específico permite reducir la cantidad de líneas de código que tendríamos que hacer con otros lenguajes de tercera generación mediante procedimientos específicos. Por ejemplo, si tuviésemos que resolver una ecuación en un lenguaje de tercera generación, tendríamos que crear diversos y complejos métodos para poder resolverla, mientras que un lenguaje de cuarta generación dedicado a este tipo de problemas ya tiene esas rutinas incluidas en el propio lenguaje, con lo que solo tendríamos que invocar la instrucción que realiza la operación que necesitamos.
- **Quinta generación**: también llamados lenguajes naturales, pretenden abstraer más aún el lenguaje utilizando un lenguaje natural con una base de conocimientos que produce un sistema basado en el conocimiento. Pueden establecer el problema que hay que resolver y las premisas y condiciones que hay que reunir para que la máquina lo resuelva. Este tipo de lenguajes los podemos encontrar frecuentemente en inteligencia artificial y lógica.

Forma de ejecución

Dependiendo de cómo un programa se ejecute dentro de un sistema, podríamos definir tres categorías de lenguajes:

- **Lenguajes compilados**: un programa traductor (compilador) convierte el código fuente en código objeto y otro programa (enlazador) unirá el código objeto del programa con el código objeto de las librerías necesarias para producir el programa ejecutable.
- **Lenguajes interpretados**: ejecutan las instrucciones directamente, sin que se genere código objeto, para ello es necesario un programa intérprete en el sistema operativo o en la propia máquina donde cada instrucción es interpretada y ejecutada de manera independiente y secuencial. La principal diferencia con el anterior es que se traducen a tiempo real solo las instrucciones que se utilicen en cada ejecución, en vez de interpretar todo el código, se vaya a utilizar o no.
- **Lenguajes virtuales**: los lenguajes virtuales tienen un funcionamiento muy similar al de los lenguajes compilados, pero, a diferencia de éstos, no es código objeto lo que genera el compilador, sino un *bytecode* que puede ser interpretado por cualquier arquitectura que tenga la máquina virtual que se encargará de interpretar el código *bytecode* generado para ejecutarlo en la máquina; aunque de ejecución lenta (como los interpretados), tienen la ventaja de poder ser multisistema y así un mismo código *bytecode* sería válido para cualquier máquina.

Paradigma de programación

El paradigma de programación es un enfoque particular para la construcción de software, un estilo de programación que facilita la tarea de programación o añade mayor funcionalidad al programa dependiendo del problema que haya que abordar. Todos los paradigmas de programación pertenecen a lenguajes de alto nivel, y es común que un lenguaje pueda usar más de un paradigma de programación.

- **Paradigma imperativo**: describe la programación como una secuencia de instrucciones que cambian el estado del programa, indicando cómo realizar una tarea.
- **Paradigma 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.
- **Paradigma 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*.
- **Paradigma 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. Permite la herencia e implementación de otras clases, pudiendo establecer *tipos* para los objetos y dejando el código más parecido al mundo real con esa abstracción conceptual.
- **Paradigma 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.
- **Paradigma lógico**: define un conjunto de reglas lógicas para ser interpretadas mediante inferencias lógicas. Permite responder preguntas planteadas al sistema para resolver problemas.

1.3 OBTENCIÓN DE CÓDIGO EJECUTABLE

Como se ha venido comentando a lo largo de todo el capítulo, nuestro programa, esté programado en el lenguaje que esté y se quiera ejecutar en la arquitectura que sea, necesita ser traducido para poder ser ejecutado (con la excepción del lenguaje máquina). Por lo que, aunque tengamos el código de nuestro programa escrito en el lenguaje de programación escogido, no podrá ser ejecutado a menos que lo traduzcamos a un idioma que nuestra máquina entienda.

1.3.1 TIPOS DE CÓDIGO (FUENTE, OBJETO Y EJECUTABLE)

El código de nuestro programa es manejado mediante programas externos comúnmente asociados al lenguaje de programación en el que está escrito nuestro programa, y a la arquitectura en donde queremos ejecutar dicho programa.

Para ello, deberemos definir los distintos tipos de código por los que pasará nuestro programa antes de ser ejecutado por el sistema.

- **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.

Cabe destacar que, si nos encontrásemos programando en un lenguaje de programación interpretado, nuestro programa no pasaría por el compilador y el enlazador, sino que solo tendríamos un código fuente que pasaría por un intérprete interno del sistema operativo o de la máquina que realizaría la compilación y ejecución línea a línea en tiempo real.

1.3.2 COMPILACIÓN

Aunque el proceso de obtener nuestro código ejecutable pase tanto por un compilador como por un enlazador, se suele llamar al proceso completo "compilación".

Todo este proceso se lleva a cabo mediante dos programas: el compilador y el enlazador. Mientras que el enlazador solamente une el código objeto con las librerías, el trabajo del compilador es mucho más completo.

Fases de compilación

- **Análisis lexicográfico**: se leen de manera secuencial todos los caracteres de nuestro código fuente, buscando palabras reservadas, operaciones, caracteres de puntuación y agrupándolos todos en cadenas de caracteres que se denominan lexemas.
- **Análisis sintáctico-semántico**: agrupa todos los componentes léxicos estudiados en el análisis anterior en forma de frases gramaticales. Con el resultado del proceso del análisis sintáctico, se revisa la coherencia de las frases gramaticales, si su "significado" es correcto, si los tipos de datos son correctos, si los *arrays* tienen el tamaño y tipo adecuados, y así consecutivamente con todas las reglas semánticas de nuestro lenguaje.
- **Generación de código intermedio**: una vez finalizado el análisis, se genera una representación intermedia a modo de pseudoensamblador con el objetivo de facilitar la tarea de traducir al código objeto.
- **Optimización de código**: revisa el código pseudoensamblador generado en el paso anterior optimizándolo para que el código resultante sea más fácil y rápido de interpretar por la máquina.
- **Generación de código**: genera el código objeto de nuestro programa en un código de lenguaje máquina relocalizable, con diversas posiciones de memoria sin establecer, ya que no sabemos en qué parte de la memoria volátil se va a ejecutar nuestro programa.
- **Enlazador de librerías**: como se ha comentado anteriormente, se enlaza nuestro código objeto con las librerías necesarias, produciendo en último término nuestro código final o código ejecutable.

Aquí podemos ver una ilustración que muestra el proceso de compilación de un modo gráfico más claro.

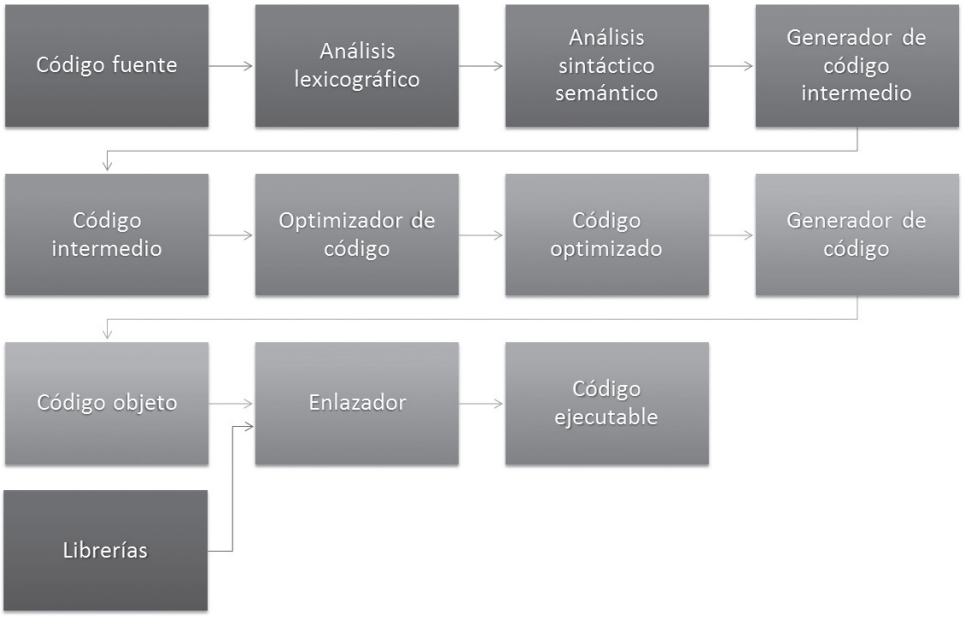


Figura 1.2. *Obtención de código ejecutable*

1.4 PROCESOS DE DESARROLLO

El desarrollo de un software o de un conjunto de aplicaciones pasa por diferentes etapas desde que se produce la necesidad de crear un software hasta que se finaliza y está listo para ser usado por un usuario. Ese conjunto de etapas en el desarrollo del software responde al concepto de ciclo de vida del programa. No en todos los programas ni en todas las ocasiones el proceso de desarrollo llevará fielmente las mismas etapas en el proceso de desarrollo; no obstante, son unas directrices muy recomendadas.

Hay más de un modelo de etapas de desarrollo, que de modo recurrente suelen ser compatibles y usadas entre sí, sin embargo vamos a estudiar uno de los modelos más extendidos y completos, el modelo en cascada.

Figura 1.3. *Modelo en cascada*

1.4.1 ANÁLISIS

La fase de análisis define los requisitos del software que hay que desarrollar. Inicialmente, esta etapa comienza con una entrevista al cliente, que establecerá lo que quiere o lo que cree que necesita, lo cual nos dará una buena idea global de lo que necesita, pero no necesariamente del todo acertada. Aunque el cliente crea que sabe lo que el software tiene que hacer, es necesaria una buena habilidad y experiencia para reconocer requisitos incompletos, ambiguos, contradictorios o incluso necesarios. Es importante que en esta etapa del proceso de desarrollo se mantenga una comunicación bilateral, aunque es frecuente encontrarse con que el cliente pretenda que dicha comunicación sea unilateral, es necesario un contraste y un consenso por ambas partes para llegar a definir los requisitos verdaderos del software. Para ello se crea un informe ERS (Especificación de Requisitos del Sistema) acompañado del diagrama de clases o de Entidad/Relación.

1.4.2 DISEÑO

En esta etapa se pretende determinar el funcionamiento de una forma global y general, sin entrar en detalles. Uno de los objetivos principales es establecer las consideraciones de los recursos del sistema, tanto físicos como lógicos. Se define por tanto el entorno que requerirá el sistema, aunque también se puede establecer en sentido contrario, es decir, diseñar el sistema en función de los recursos de los que se dispone.

En la fase de diseño se crearán los diagramas de casos de uso y de secuencia para definir la funcionalidad del sistema.

Se especificará también el formato de la información de entrada y salida, las estructuras de datos y la división modular. Con todos esos diagramas e información se obtendrá el cuaderno de carga.

1.4.3 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.

Gracias a las etapas anteriores, el programador contará con un análisis completo del sistema que hay que codificar y con una especificación de la estructura básica que se necesitará, por lo que en un principio solo habría que traducir el cuaderno de carga en el lenguaje deseado para culminar la etapa de codificación, pero esto no es siempre así, las dificultades son recurrentes mientras se modifica. Por supuesto que cuanto más exhaustivo haya sido el análisis y el diseño, la tarea será más sencilla, pero nunca está exento de necesitar un reanálisis o un rediseño al encontrar un problema al programar el software.

1.4.4 PRUEBAS

Con una doble funcionalidad, las pruebas buscan confirmar que la codificación ha sido exitosa y 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.

No es un proceso estático, y es usual realizar pruebas después de otras etapas, como la documentación. Generalmente, las pruebas realizadas posteriormente a la documentación se realizan por personal inexperto en el ámbito de las pruebas de software, con el objetivo de corroborar que la documentación sea de calidad y satisfactoria para el buen uso de la aplicación.

En general, las pruebas las realiza, idílicamente, personal diferente al que codificó la aplicación, con una amplia experiencia en programación, personas capaces de saber en qué condiciones un software puede fallar de antemano sin un análisis previo.

1.4.5 DOCUMENTACIÓN

Por norma general, la documentación que se realiza de un software tiene dos caras: la documentación disponible para el usuario y la documentación destinada al propio equipo de desarrollo.

La documentación para el usuario debe mostrar una información completa y de calidad que ilustre mediante los recursos más adecuados cómo manejar la aplicación. Una buena documentación debería permitir a un usuario cualquiera comprender el propósito y el modo de uso de la aplicación sin información previa o adicional.

Por otro lado, tenemos la documentación técnica, destinada a equipos de desarrollo, que explica el funcionamiento interno del programa, haciendo especial hincapié en explicar la codificación del programa. Se pretende con ello permitir a un equipo de desarrollo cualquiera poder entender el programa y modificarlo si fuera necesario. En casos donde el software realizado sea un servicio que pueda interoperar con otras aplicaciones, la documentación técnica hace posible que los equipos de desarrollo puedan realizar correctamente el software que trabajará con nuestra aplicación.

1.4.6 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.

Cabe destacar que en caso de que nuestro software sea una versión sustitutiva de un software anterior es recomendable valorar la convivencia de sendas aplicaciones durante un proceso de adaptación.

1.4.7 MANTENIMIENTO

Son muy escasas las ocasiones en las que un software no vaya a necesitar de un mantenimiento continuado. 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.

Cuando el mantenimiento que hay que realizar consiste en una ampliación, el modelo en cascada suele volverse cíclico, por lo que, dependiendo de la naturaleza de la ampliación, puede que sea necesario analizar los requisitos, diseñar la ampliación, codificar la ampliación, probarla, documentarla, implementarla y, por supuesto, dar soporte de mantenimiento sobre la misma, por lo que al final este modelo es recursivo y cíclico para cada aplicación y no es un camino rígido de principio a fin.

1.5 ROLES QUE INTERACTÚAN EN EL DESARROLLO

A lo largo del proceso de desarrollo de un software deberemos realizar, como ya hemos visto anteriormente, diferentes y diversas tareas. Es por ello que el personal que interviene en el desarrollo de un software es tan diverso como las diferentes tareas que se van a realizar.

Los roles no son necesariamente rígidos y es habitual que participen en varias etapas del proceso de desarrollo.

- **Analista de sistemas**
  - Uno de los roles más antiguos en el desarrollo del software. 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**
  - Comúnmente llamado "desarrollador", domina una visión más amplia de la programación, 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**
  - Se encarga de manera exclusiva de crear el resultado del estudio realizado por analistas y diseñadores. Escribe el código fuente del software.
  - Participa en la etapa de codificación.
- **Arquitecto de software**
  - Es la argamasa que cohesiona el proceso de desarrollo. 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.

1.6 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. La arquitectura de software 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.

El objetivo principal de la arquitectura de software consiste en proporcionar elementos que ayuden a la toma de decisiones abstrayendo los conceptos del sistema mediante un lenguaje común. Dicho conjunto de herramientas, conceptos y elementos de abstracción se organizan en forma de patrones y modelos.

Los resultados obtenidos después de efectuar buenas prácticas de arquitectura de software deben proporcionar capas de abstracción y encapsulado, organizando el software de manera diferente dependiendo de la visión o criterio de la estructura.

1.6.1 PATRONES DE DESARROLLO

Los patrones de desarrollo, también llamados patrones de diseño, establecen los componentes de la arquitectura y la funcionalidad y comportamiento de cada uno.

Las directrices marcadas por los patrones de diseño facilitan la tarea de diseñar un software, aunque no en su totalidad. Los patrones no especifican todas las características o relaciones de los componentes en nuestro software, sino que están centrados en un ámbito específico. Cada patrón determina y especifica los aspectos de uno de los tres ámbitos principales: creacionales, estructurales y de comportamiento.

Nos podríamos encontrar con otro ámbito de patrones, los patrones de interacción, los cuales tienen el objetivo de definir diferentes directrices para la creación de interfaces, en donde prima la usabilidad. El aspecto gráfico y la usabilidad de los controles ofrecidos al usuario para manejar la aplicación son sin duda una parte sumamente importante en un software, y no es algo que se deba menospreciar. Los patrones de interacción se alejan parcialmente del ámbito de los otros patrones de diseño y no entran en su totalidad en los patrones de desarrollo.

Recalcando de nuevo, como se hizo en el párrafo anterior, la mala praxis que es descuidar la usabilidad de nuestras interfaces, nos vamos a centrar en los patrones de desarrollo pertenecientes a los ámbitos creacionales, estructurales y de comportamiento. Realizaremos una visión global de los diferentes patrones que podemos encontrar en cada ámbito mencionado.

Creacionales

Los patrones creacionales definen el modo en que se construyen los objetos, principalmente con el objetivo de encapsular la creación de los mismos haciendo que los constructores sean privados y el modo de crear una instancia sea mediante un método estático que devuelva el objeto. La característica fundamental de la programación orientada a objetos en la que recaen estas prácticas de patrones creacionales es el polimorfismo.

Fábrica abstracta

Se utiliza cuando se necesita crear diferentes objetos pertenecientes a la misma familia. Disponemos de una "factoría abstracta" que define las interfaces de las factorías y de varias "factorías concretas" que interpretan a una familia concreta. La mejor forma de ver este patrón es imaginarlo como factorías tangibles, podríamos pensar en la fábrica de coches como la fábrica abstracta y en el coche como el producto abstracto. Y podemos ver la fábrica de Gijón como una factoría concreta y el coche de Gijón como producto concreto. Aunque el producto sea el mismo, las diferentes fábricas en sus diferentes localizaciones tendrán acceso a unos recursos y proveedores diferentes.

Por tanto, tendríamos una clase con un método para crear coches abstractos, una clase por cada fábrica de coches que tendrá un método creador de coches con un parámetro de materiales. Para crear los coches utilizaremos el constructor abstracto, utilizando el parámetro de los materiales de la factoría concreta. Con ello, tenemos que para crear un coche en la fábrica de Gijón (factoría concreta) utilizaríamos el siguiente método:

Constructor virtual

EJEMPLO 1.1

CONSTRUCTOR VIRTUAL

```
Coche crearCoche() {
    FactoríaMateriales fm = new MaterialesGijon();
    Coche coche = new Coche(fm); // Uso de la factoría
    coche.montar();
    coche.pintar();
    return coche;
}
```

Se utiliza cuando de una misma factoría (utilizando el concepto del patrón anterior) se utilizan diferentes objetos complejos. Se crea un constructor abstracto por cada tipo de producto de la factoría y diferentes constructores concretos para cada producto específico. Consta de una clase producto, una clase abstracta constructor, varios constructores concretos, un director y un cliente.

Si tenemos una empresa que vende diferentes sets de ADSL, podríamos utilizar el patrón de constructor virtual de la siguiente manera:

EJEMPLO 1.2a

PRODUCTO

```
class Adsl {
    private String reuter = "";
    private Int velocidad = "";

    public void setReuter(String reuter) { this.reuter = reuter; }
    public void setVelocidad(Int velocidad) { this.velocidad = velocidad; }
}
```

EJEMPLO 1.2b

CONSTRUCTOR ABSTRACTO

```
abstract class AdslBuilder {
    protected Adsl adsl;

    public Adsl getAdsl() { return adsl; }
    public void crearNuevaAdsl() { adsl = new Adsl(); }

    public abstract void buildReuter();
    public abstract void buildVelocidad();
}
```

EJEMPLO 1.2c

CONSTRUCTOR CONCRETO

```
class BasicoAdslBuilder extends AdslBuilder {
    public void buildReuter()   { adsl.setReuter ("DLink T-504"); }
    public void buildVelocidad()    { adsl.setVelocidad(12); }
}
```

EJEMPLO 1.2d

DIRECTOR

```
class Montador {
    private AdslBuilder adslBuilder;

    public void setAdslBuilder(AdslBuilder ab) { adslBuilder = ab; }
    public Adsl getAdsl() { return adslBuilder.getAdsl(); }

    public void construirAdsl() {
       adslBuilder.crearNuevaAdsl();
       adslBuilder.buildReuter();
       adslBuilder.buildVelocidad();
    }

}
```

EJEMPLO 1.2e

CLIENTE

```
class InstalarAdsl {
    public static void main(String[] args) {
        Montador montador = new Montador();
        AdslBuilder basico_adslbuilder = new BasicoAdslBuilder ();
        AdslBuilder avanzado_adslbuilder = new AvanzadoAdslBuilder ();

        montador.setAdslBuilder(avanzado_adslbuilder);
        montador.construirAdsl();

        Adsl adsl = montador.getAdsl();
    }

}
```

Instancia única

La instancia única o *Singleton* es el patrón de diseño que nos permite 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.

EJEMPLO 1.3

SINGLETON

```
public class Singleton
{
    private static readonly Lazy<Singleton> instance = new Lazy<Singleton>(() =>
new Singleton());

    private Singleton()
    {
    }
    public static Singleton Instance
    {
        get
        {

            return instance.Value;
        }
    }
}
public class Prueba
{
   private static void Main(string[] args)
   {
     Singleton s1 = Singleton.Instance;
     Singleton s2 = Singleton.Instance;
     if(s1==s2)
     {
       Console.WriteLine( "Es el mismo objeto");
     }
   }
}
```

Lo más importante que hay que tener en cuenta cuando se implementa el patrón es privatizar el constructor de la clase, declarar estático el método que nos devuelve la instancia y declarar al atributo de nuestra clase como privado, estático y de solo lectura. Éstas son las claves para implementar correctamente el patrón *Singleton* dentro de cualquier lenguaje.

Estructurales

Los patrones estructurales establecen las relaciones y organizaciones entre los diferentes componentes de nuestro software, resolviendo de una manera elegante diversos problemas que nos encontramos al implementar soluciones sin haber evaluado previamente todas las consecuencias y posibilidades. No hay que olvidar que el uso de patrones no es una cuestión unitaria, para solucionar un problema o eventualidad cuando surja, sino que nos plantean una serie de directrices que pueden solventar problemas futuros, hay que pensar a largo plazo y valorar siempre la expandibilidad del proyecto.

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.

Tenemos una empresa de creación de páginas webs y, a partir del modelo básico, tenemos diferentes funcionalidades que podemos añadir. Para facilitarnos la tarea de crear presupuestos, nos hemos puesto manos a la obra y estamos creando una aplicación para calcularlos. Partimos en un principio de un presupuesto base, correspondiente a la página web básica, y queremos añadir la posibilidad de que esa página web pueda contener un carrito de la compra y un sistema de autentificación de usuarios. Si nos pusiésemos a implementar una subclase para cada tipo de página, tendríamos cuatro clases en total: Página, PáginaConCarrito, PáginaConLogin y PáginaConCarritoYLogin. Si quisiésemos añadir en un futuro un libro de visitas y/o un foro, acabaríamos teniendo ocho y dieciséis clases respectivamente, con lo que tendríamos al final una solución insostenible.

Para solventar ese problema, creamos una subclase abstracta PaginaDecorator de la que heredan las clases CarritoDecorator y LoginDecorator. De este modo, podemos implementar tantas funcionalidades como queramos creando nuevas clases de herencia para cada funcionalidad.

La clase PaginaDecorator emula y encapsula el comportamiento de Página y utiliza composición recursiva para añadir tantos decoradores concretos como se necesiten. Con este patrón, en vez de definir los objetos con la funcionalidad, definimos las funcionalidades y luego se las añadimos al objeto que queramos.

EJEMPLO 1.4

DECORADOR

```
class Program
    {
        static void Main(string[] args)
        {
          CarritoDecorator paginaConCarrito = new CarritoDecorator(new Pagina());
            Console.WriteLine(paginaConCarrito.Calcular());

          LoginDecorator paginaConCarritoYLogin = new LoginDecorator(paginaConCarrito);
            Console.WriteLine(paginaConCarritoYLogin.Calcular());

            LoginDecorator paginaConLogin = new LoginDecorator(new Pagina());
            Console.WriteLine(paginaConLogin.Calcular());

           //En el extraño supuesto de que se necesiten dos sistemas de autentificación
          LoginDecorator paginaConDobleLogin = new LoginDecorator(paginaConLogin);
            Console.WriteLine(paginaConDobleLogin.Calcular());

            Console.Read();
        }
```

En cada decorador incluiríamos la funcionalidad (que en este caso sería añadir a la página el precio por la funcionalidad), pero podría ser cualquier otra cosa.

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.

Supongamos que tenemos una jerarquía de contenedores y párrafos, donde los contenedores pueden albergar tanto párrafos como otros contenedores. Necesitamos poder colorearlos y que hereden el color del contenedor raíz en el que se encuentren.

Para ello creamos una lista de componentes con estructura de composición recursiva en árbol. 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.

EJEMPLO 1.5a

COMPONENTE

```
public interface Componente {

   public void pintar(String color);
}
```

EJEMPLO 1.5b

COMPONENTE CONCRETO

```
public class Parrafo implements Componente{

   private String nombre;

      public Parrafo (String nombre) {
          this.nombre = nombre;
      }

   public void pintar(String color){
       System.out.println("Se pinto a " + nombre + " de color " +color);
}
```

EJEMPLO 1.5c

CONTENEDOR

```
public class Contenedor implements Componente {

   private String nombre;

   private ArrayList <componente> componentes;

   public Contenedor(String nombre) {
      setNombre(nombre);
      setComponentes(new ArrayList());
   }

   public Contenedor() {
      setNombre("");
      setComponentes(new ArrayList());
   }
   public String getNombre() {
      return nombre;
   }

   public void setNombre(String nombre) {
      this.nombre = nombre;
   }

   public void pintar(String color) {
     for (Componente c : componentes) c.pintar(color);
     System.out.println("Se pinto a " + nombre + " de color " + color);
   }

   public void add(Componente c) {
      getComponentes().add(c);
   }
}
```

Fachada

Cuando diseñamos una aplicación, es correcto, lógico y usual crear subdivisiones en el sistema para organizar nuestro código, utilizando diferentes criterios, ya sean por funcionalidad o conceptuales.

La fachada nos permite crear una interfaz que unifique el acceso a dichos subsistemas, de modo que simplifica el acceso y lo hace más fácil de usar. Podemos verlo como un nivel superior en la abstracción de nuestro código, donde para realizar complicadas o sencillas operaciones solo necesitamos tener acceso a la clase fachada para ejecutar nuestro programa.

Veamos un ejemplo conceptual sobre los usos del patrón fachada, emularemos un funcionamiento ficticio a la hora de abrir un archivo.

EJEMPLO 1.6

FACHADA

```
public class Fachada {
    private Memoria mem = new Memoria();
    private DiscoDuro dd = new DiscoDuro();
    private Procesador cpu = new Procesador();

    public void abrirArchivo(String archivo) {
        mem.cargar(dd.getDireccion(archivo));
        cpu.asignarProceso(mem.getPosUltimaCarga());
    }
 }
public class Memoria {
    public Memoria() { }
    public void cargar(String direccion) { /* ... */ }
    public MemPos getPosUltimaCarga() { /* ... */ }
}
public class DiscoDuro {
    public DiscoDuro() { }
    public void getDireccion() { /* ... */ }
}
public class Procesador {
    public Procesador() { }
    public void asignarProceso(MemPos posicion) { /* ... */ }
}
```

Analizando el uso de la fachada, podemos observar que no es un procedimiento muy distinto de lo que hemos tenido en cualquier clase o método. Es decir, combinamos y metodizamos las diferentes rutinas que tenemos que hacer para llevar a cabo una operación concreta. Agrupamos las diferentes operaciones para simplificarlas y solo necesitar de una llamada para realizar un cúmulo de acciones que conllevan la operación que necesitamos.

Gracias a la fachada, ahora si queremos abrir un archivo no necesitamos instanciar 3 clases y realizar las llamadas para abrir el archivo, solo necesitamos invocar al método *abrirArchivo* de nuestra fachada.

Comportamiento

Los patrones de comportamiento describen las comunicaciones entre objetos y clases y establecen directrices sobre cómo utilizar los objetos y clases para optimizar y organizar el comportamiento, interacción y comunicación entre ellos.

Estado

El patrón 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.

Para crear el patrón estado, nos crearemos un contexto, que nos servirá de interfaz con el cliente, una interfaz estado para encapsular las responsabilidades del contexto en el estado en que se encuentre, y una serie de subclases estados concretos para definir los diferentes comportamientos de nuestro objeto.

Supongamos que estamos creando un software para manejar el botón de llamada a un ascensor, obviamente se comportará de forma diferente dependiendo de si se encuentra en el mismo piso desde el que se llama o si se ecuentra por debajo o por encima del piso desde el que se le llama.

EJEMPLO 1.7a

ESTADO

```
public interface Estado
 {
        void irPiso(int piso);
 }
```

EJEMPLO 1.7b

CONTEXTO

```
public class Contexto
 {
        private Estado estado;

        public void setEstado(Estado estado){
                this.estado = estado;
        }
        public Estado getEstado(){
                return estado;
        }
        public void llamar(int piso){
                estado.irPiso(piso);
        }
 }
```

EJEMPLO 1.7c

ESTADO CONCRETO

```
public class estadoAscensorAbajo implements Estado
 {
        public void irPiso(int piso){
          Ascensor ascensor = Ascensor.getAscensor();
          ascensor.subir(piso - ascensor.getPiso());
        }
 }
public class estadoAscensorArriba implements Estado
 {
        public void irPiso(int piso){
          Ascensor ascensor = Ascensor.getAscensor();
          ascensor.bajar(ascensor.getPiso() - piso);
        }
 }
public class estadoAscensorMismo implements Estado
 {
        public void irPiso(int piso){
          Ascensor ascensor = Ascensor.getAscensor();
          ascensor.abrirPuerta();
        }
 }
```

Aun teniendo en cuenta que en el ejemplo mostrado no parece extremadamente útil (ya que el estado viene especificado por una diferencia numérica), se puede observar tanto su funcionamiento como utilidad. Como añadido, podemos observar como nuestro código queda de este modo mejor organizado, y nos puede evitar utilizar condicionales innecesarios, así dejamos nuestro código más entendible y funcional, permitiendo crear estados sin meternos en complejos condicionales anidados.

Visitor

El patrón 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.

Una de las particularidades más remarcables de este patrón reside en el método aceptar de un elemento, donde se define una llamada al método visitar del visitante y el argumento visitante al llamar al método aceptar en los métodos hijos de nuestra estructura de elementos.

Supongamos que tenemos una jerarquía de expresiones aritméticas sobre las que queremos definir visitantes, entre los que queremos que se encuentre un visitante que convierta la expresión aritmética en una cadena de caracteres.

EJEMPLO 1.8a

ELEMENTO

```
public abstract class Expresion {
  abstract public void aceptar(VisitanteExpresion v);
}
```

EJEMPLO 1.8b

ELEMENTO CONCRETO

```
public class Constante extends Expresion {
  public Constante(int valor) { _valor = valor; }
 public void aceptar(VisitanteExpresion v) {
    v.visitarConstante(this);
  }
  int _valor;
}
public class Variable extends Expresion {
  public Variable(String variable) { _variable = variable; }
  public void aceptar(VisitanteExpresion v) {
     v.visitarVariable(this);
  }
  String _variable;
}
public abstract class OpBinaria extends Expresion {
  public OpBinaria(Expresion izq, Expresion der) {
     _izq = izq; _der = der;
  }
  Expresion _izq, _der;
}
public class Suma extends OpBinaria {
  public Suma(Expresion izq, Expresion der) { super(izq, der); }
  public void aceptar(VisitanteExpresion v){v.visitarSuma(this);}
}
public class Mult extends OpBinaria {
  public Mult(Expresion izq, Expresion der) { super(izq, der); }
  public void aceptar(VisitanteExpresion v){v.visitarMult(this);}
}
```

EJEMPLO 1.8c

VISITANTE

```
public interface VisitanteExpresion {
  public void visitarSuma(Suma s);
  public void visitarMult(Mult m);
  public void visitarVariable(Variable v);
  public void visitarConstante(Constante c);
}
```

EJEMPLO 1.8d

VISITANTE CONCRETO

```
public class ExpressionToString implements VisitanteExpresion {
  public void visitarVariable(Variable v) {
        _resultado = v._variable;
  }
  public void visitarConstante(Constante c) {
         _resultado = String.valueOf(c._valor);
  }
  private void visitarOpBinaria(OpBinaria op, String pOperacion){
          op._izq.aceptar(this);
          String pIzq = obtenerResultado();

          op._der.aceptar(this);
          String pDer = obtenerResultado();

          _resultado = "(" + pIzq + pOperacion + pDer + ")";
  }
  public void visitarSuma(Suma s) {
        visitarOpBinaria(s, "+");
  }
  public void visitarMult(Mult m) {
        visitarOpBinaria(m, "*");
  }
  public String obtenerResultado() {
        return _resultado;
  }
  private String _resultado;
}
```

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.

Para ello nos creamos un iterador y un agregado, donde el iterador es una interfaz que permite recorrer el agregado encapsulando las operaciones necesarias y ocultándolas al cliente, algo parecido a lo que hacíamos con el patrón fachada.

EJEMPLO 1.9a

AGREGADO

```
public class Coleccion {
    public int[] _datos;

    public Coleccion(int valores){
        _datos = new int[valores];
        for (int i = 0; i < _datos.length; i++){
            _datos[i] = 0;
        }
    }
    public int getValor(int pos){
        return _datos[pos];
    }
    public void setValor(int pos, int valor){
        _datos[pos] = valor;
    }
    public int dimension(){
        return _datos.length;
    }
 public IteradorVector iterador(){
        return new IteradorColeccion(this);
    }
}
```

EJEMPLO 1.9b

ITERADOR

```
public class IteradorColeccion{
    private int[] _vector;
    private int _posicion;

    public IteradorColeccion(Coleccion vector) {
        _vector = vector._datos;
        _posicion = 0;
    }
    public boolean hayMas(){
        if (_posicion < _vector.length)
                    return true;
            else
                    return false;
    }
    public Object siguiente(){
        int valor = _vector[_posicion];
        _posicion++;
        return valor;
    }
}
```

Antipatrones

Los antipatrones son la contraparte de los patrones que hemos estado viendo anteriormente. Es decir, si los patrones son buenas prácticas de desarrollo de software, los antipatrones son justamente lo contrario, malas prácticas en el desarrollo de software.

La definición más acertada del antipatrón es la aplicación de un patrón de software en un contexto equivocado, aunque el uso que damos a la palabra "antipatrón" no se detiene ahí, y habitualmente la encontramos usada como "malos patrones" o "malas prácticas de desarrollo". Al final, todo se reduce al mismo concepto: situaciones o prácticas que no tenemos que hacer o tenemos que evitar a la hora de desarrollar un software.

La cantidad de antipatrones en los que podemos caer, o nos podemos encontrar, es sencillamente abrumadora, en esta sección vamos a ver una selección de los antipatrones más "populares" o habituales.

Código spaghetti

El código *spaghetti* era muy habitual encontrarlo en los inicios de la programación, antes de que se definiese el concepto de programación orientada a objetos, donde el código no estaba metodizado y era un absoluto caos. 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, con tantos saltos a diferentes fragmentos de código, las líneas se enmarañaban y cruzaban una y otra vez, dejándonos un garabato con aspecto similar al de un plato de *spaghetti*. Hoy en día es habitual encontrarlo en scripts incrustados en el código, todos ellos carentes y necesitados de refactorizar.

Flujo de lava

Grandes cantidades de código desordenado, módulos y añadidos ingentes que rompen la estructura natural del software; documentación paupérrima o código abandonado serían varios de los indicadores o "síntomas" que se verían en un código "enfermado" con este antipatrón. Este antipatrón es más habitual encontrarlo en software codificado bajo una mala gestión, seguramente no sea un problema fruto de un programador descuidado, sino de una mala gestión por el jefe de proyecto y por las necesidades del cliente, que hace que tengamos fragmentos de código sin terminar y anexos a la aplicación fruto de los cambios recurrentes en la urgencia, prioridades, preferencias y necesidades del cliente.

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 puede ocasionar infinidad de problemas. Las plataformas de trabajo, los lenguajes y los paradigmas tienen diferentes capacidades y limitaciones que nos ofrecen funcionalidades diferentes. Por lo que nuestras preferencias pueden hacernos escoger unos recursos inapropiados para el software que necesitamos desarrollar.

Reinventar la rueda

Este antipatrón aparece cuando implantamos soluciones a problemas que ya existen en el *framework* contra el que trabajamos. Al reimplementar esos componentes ya existentes y no reutilizar el código, no solo perdemos tiempo, sino que el software se vuelve más denso de forma innecesaria y en ocasiones podemos empeorar la cohesión del código dentro de la misma plataforma. Puede venir por un desconocimiento del *framework* donde se trabaja o por la injustificada necesidad de personalizar los componentes que utilizamos.

Infierno de las dependencias

En contrapartida al anterior antipatrón, depender de manera abusiva de las librerías y componentes de un entorno de desarrollo o plataforma puede ocasionarnos muchos problemas derivados de las diferentes versiones de las dependencias.

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. De hecho, ni siquiera es un auténtico control de excepciones, ya que es solo fruto de pensar que dicha excepción se puede producir y lanzamos la excepción que necesitamos, además, por otro lado, estamos creando código innecesario, ya que el propio control de excepciones del lenguaje nos ofrece esta funcionalidad sin necesidad de crear condicionales para ello.

Cadenas mágicas

La utilización de cadenas de caracteres explícitas no es una práctica recomendada; en ocasiones, y por requisitos ajenos a nuestro código (ya sea por el *framework*, librería o similar), incluimos cadenas de caracteres a la hora de realizar llamadas o comparaciones de manera recurrente. El problema más obvio que apreciamos en esta mala práctica es que necesitamos modificar y recompilar el código en caso de que necesitemos cambiar la(s) cadena(s) de caracteres.

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. Duplicar el código en vez de reutilizarlo nunca es una práctica adecuada, la necesidad de duplicar código es síntoma de que nuestro código necesita ser refactorizado, por ejemplo con una refactorización de "Extraer método" o pensar un modo de hacer y modificar el método para que sea accesible desde los sitios donde necesitas invocarlo.

ACTIVIDADES 1.2

- ¿Qué patrones podrían usarse para construir una aplicación que se encargase de administrar y crear los extractos de un banco que tiene varias sucursales en una misma ciudad?
- Relacione los patrones y antipatrones vistos especificando qué patrones podrían sustituir al uso de algún antipatrón.

1.6.2 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.

Para solventar esa necesidad, se ideó el desarrollo en 3 capas, que separa la lógica de negocio, el acceso a datos y la presentación al usuario en tres capas que pueden tener cada una tantos niveles como se quiera o necesite.

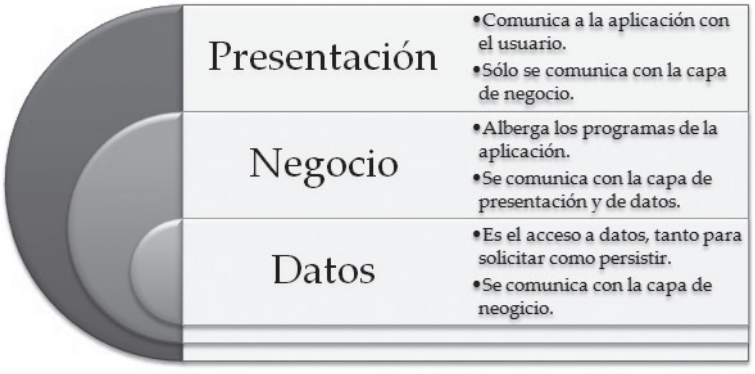


Figura 1.4. *Desarrollo en 3 capas*

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. Por ejemplo, podríamos necesitar acceder y utilizar datos contenidos tanto en nuestra propia base de datos, como en la base de datos de un banco, para ello utilizaríamos la capa de datos, en donde accederíamos a nuestro gestor de base de datos y al servicio que nos ofrezca

el banco para solicitar dichos datos. Esto podría hacerse sin necesitar un desarrollo en tres capas, pero la principal ventaja (aparte de la encapsulación y ocultación de código y datos entre las capas) que nos aporta reside en evitar modificar la lógica de negocio por necesitar acceder a diferentes datos, todo está perfectamente estructurado y nos permite modificar las fuentes y el modo en que accedemos a los datos de los programas que trabajan con ellos.

Modelo vista controlador

Dentro del desarrollo por capas, encontramos diferentes modelos de software, uno de ellos es el MVC (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.

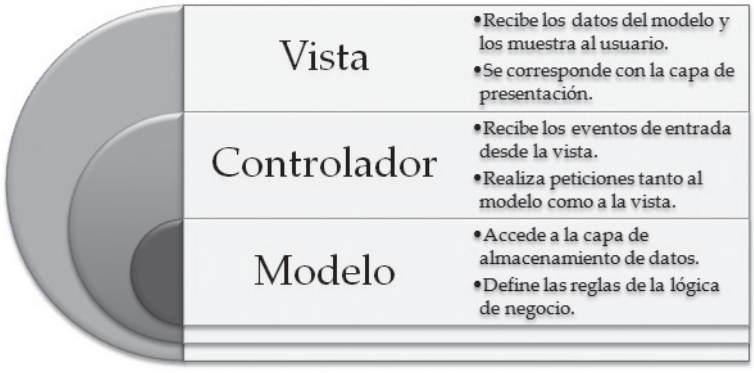


Figura 1.5. *Modelo Vista Controlador*

A la hora de mostrar los datos de modelo en las vistas, se suele establecer una serie de "bindings" que enlacen diferentes componentes de la vista a propiedades y campos de las entidades de los datos a los que tiene acceso el modelo.

Modelo vista vistamodelo

EL MVVM parte de un concepto muy similar al modelo MVC, tanto, que no resulta extraño pensar que es una ampliación o modificación al MVC. De hecho, muchas *frameworks* actuales ofrecen el uso del MVVM a través del MVC *framework* añadiéndole un ViewModel. Siendo objetivos, no es una visión muy apartada de la realidad, ya que en esencia parten de la misma necesidad y concepto.

A diferencia del MVC, la vista del MVVM es un observador que se actualiza cuando cambia la información contenida en el VistaModelo. El componente VistaModelo en MVVM contiene a su vez un controlador al igual que en el MVC, y además utiliza un VistaModelo que actúa como un modelo virtual y personalizado que contiene la información necesaria para mostrarla en la vista. Es decir, al igual que en el MVC, 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, ocultando así al modelo de la vista, dejando al modelo como una mera representación de las entidades para la persistencia de los datos.

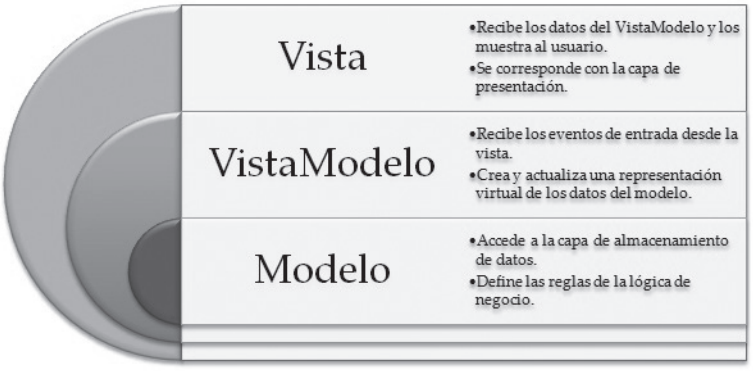


Figura 1.6. *Modelo Vista VistaModelo*

Mediante el uso del VistaModelo, hemos logrado ocultar el modelo a la vista, además de evitar la necesidad de crear "bindings" manuales entre la vista y el modelo, ya que el propio VistaModelo se puede "bindear" directamente.

ACTIVIDADES 1.3

- Relacione los dos modelos de desarrollo en 3 capas con los diferentes patrones, especificando qué patrones podrían incluirse en las diferentes capas de los modelos.

RESUMEN DEL CAPÍTULO

En este tema nos hemos introducido en el mundo del desarrollo de software. Esta primera toma de contacto nos ofrecerá una excelente base sobre la que tratar todos los temas de desarrollo y diseño de proyectos y estrategias de desarrollo de software.

Se han incluido conceptos y técnicas que el alumno no llegará a dominar hasta que profundice en sus conocimientos de programación orientada a objetos, pero aporta un excelente complemento conceptual que se puede practicar por medio del pseudocódigo.

El alumno debería ser capaz de identificar el lenguaje que necesita utilizar para desarrollar un software y reconocer los patrones necesarios para llevar dicha tarea a cabo. Más adelante podrá combinar el conocimiento de antipatrones mencionados en este tema con los "malos olores" de la refactorización, mejorando la calidad de software y consiguiendo una mayor calidad tanto en la aplicación desarrollada como en el código de la misma.

Se recomienda que una vez profundizados y consolidados los conocimientos de programación se vuelva a realizar una lectura de este tema, esto permitirá al alumno comprender mucho mejor los conceptos y técnicas de arquitectura de software.

EJERCICIOS PROPUESTOS

- **1.** Según el esquema de la máquina von Neumann. ¿Cuáles serían las microinstrucciones que se ejecutarían para multiplicar por tres el valor de R1, sumárselo a R2 y guardar el resultado en R3?
- **2.** La pizzería Borde Exterior tiene establecimientos y almacenes en Nueva York, Oslo y Lepe. ¿Qué patrón de desarrollo usaríamos para crear las diferentes pizzas que pueden realizar en dichos establecimientos? Impleméntelo.
- **3.** La posada El Poni Pisador ofrece diferentes servicios a sus clientes, tales como *spa*, piscina y servicios de habitaciones, además de la propia estancia en la habitación. ¿Qué patrón utilizaríamos para desarrollar una aplicación que nos calculase el total a pagar por el cliente? Impleméntelo.

4. Dada la siguiente estrucura, indique a qué patrón corresponde y rellene el código especificado en los comentarios.

CÓDIGO EJERCICIO 4

```
public class Interfaz {
    private LibreriaLibros libros = new LibreriaLibros();
    private LibreriaVideo videos = new LibreriaVideo();
    private LibreriaMusica musica = new LibreriaMusica();

    public void buscarLibros() {
        libros.buscarLibros();
    }

    public void buscarMusica() {
        musica.buscarMusica();
    }

public void buscarVideo() {
        videos.buscarVideo();
    }
}

public class LibreriaLibros {
    public LibreriaLibros() { }
    public void buscarLibros() { /* ... */ }
}

public class LibreriaVideo {

    public LibreriaVideo() { }
    public void buscarVideo() { /* ... */ }
}

public class LibreriaMusica {

    public LibreriaMusica() { }
    public void buscarMusica() { /* ... */ }
}

public class Cliente {

    public static void main(String args[]){
        //Rellenar
    }
}
```

5. Seleccione los diferentes antipatrones que aparecen en el siguiente código. Explique cómo podría evitarlos.

CÓDIGO EJERCICIO 5

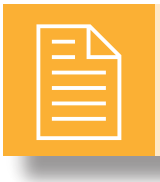
```
public class Cliente {
    public static void main(String args[]){
   Consola consola = new Consola();
   Bucle:
   If (consola.leer()!="*")
       Goto Bucle;
   Else
       System.out.println("¡Bonito asterisco!");
    }
}
```

6. Cree el pseudocódigo necesario para crear una aplicación basada en MVVM que genere movimientos bancarios de ingreso, retirada y traspasos de efectivo.

7. ¿Qué patrón se está utilizando en el siguiente código?

CÓDIGO EJERCICIO 7

```
public class Test extends JFrame {
   public static void main(String args[]) {
      Test frame = new Test();
      frame.setTitle("Swing Actions");
      frame.setSize(500, 400);
      frame.setLocation(400, 200);
      frame.show();
   }
 public Test() {
      JMenuBar mb = new JMenuBar();
      JMenu fileMenu = new JMenu("File");
      fileMenu.add(new ShowDialogAction());
      fileMenu.add(new ExitAction());
      mb.add(fileMenu);
setJMenuBar(mb);
   }
}
class ShowDialogAction extends AbstractAction {
   public ShowDialogAction(){
   super("show dialog");
   }
 public void actionPerformed(ActionEvent e) {
      JOptionPane.showMessageDialog((Component)e.getSource(),
                                "An action generated this dialog");
   }
}
class ExitAction extends AbstractAction {
   public ExitAction() {
      super("exit");
   }
 public void actionPerformed(ActionEvent e) {
      System.exit(0);
   }
}
```



TEST DE CONOCIMIENTOS

1 ¿Ante qué tipo de lenguaje estamos si procesa y traduce las instrucciones en tiempo de ejecución?

a) De tercera generación.
b) Interpretado.
c) Compilado.
d) Todos los anteriores.

2 El código objeto puede ser:

a) Código máquina.
b) *Bytecode*.
c) Las respuestas *a* y *b* son correctas.
d) Ninguna de las anteriores es correcta.

3 ¿Qué se hace durante el proceso de explotación de un software?

a) Desplegar o distribuir nuestro software en el sistema.
b) Comprobar el funcionamiento y seguridad del software.
c) Asegurar y mantener las necesidades del software una vez distribuido.
d) Todas las anteriores son correctas.

4 Un analista programador se encarga de:

a) Codificar el diseño de un software en el lenguaje deseado.
b) Diseñar o mejorar el diseño de un proyecto de software.
c) Las respuestas *a* y *b* son correctas.
d) Todas las anteriores son correctas.

5 ¿Qué antipatrones podemos encontrarnos al diseñar y codificar una aplicación mediante el desarrollo en tres capas MVVM?

a) Ninguno, el desarrollo en 3 capas evita el uso de antipatrones.
b) Cualquiera.
c) Solo los antipatrones relacionados con la codificación.
d) Ninguno de los anteriores.

6 Si creamos una clase cuyo único propósito sea el de declarar unos métodos de acceso más accesibles para la clase cliente… ¿qué estamos haciendo?

a) Aumentar el nivel de abstracción de nuestro código.
b) Una clase fachada.
c) Todas las respuestas anteriores son correctas.
d) Ninguna de las anteriores es correcta.

7 ¿Qué elemento se encarga de observar los cambios e interacciones en la interfaz de usuario?

a) Modelo.
b) Vista.
c) VistaModelo.
d) Controlador.