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:

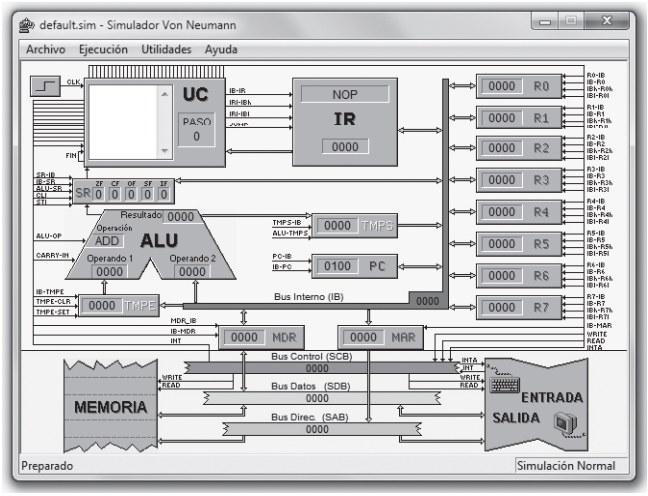

Figura 1.1. *Máquina de von Neumann*

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.

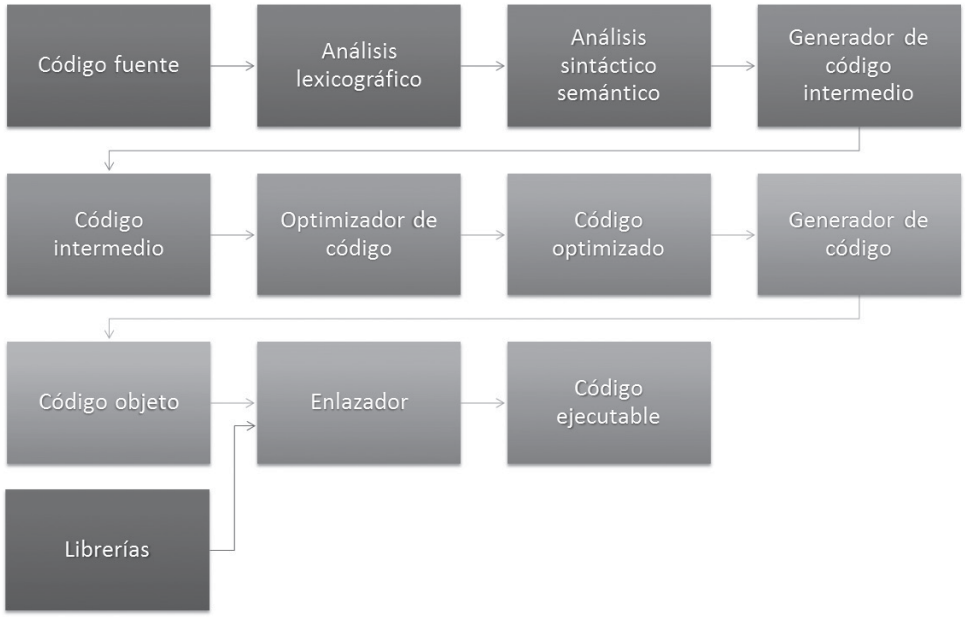

Figura 1.2. *Obtención de código ejecutable*

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.