**UNIDAD 1. ELEMENTOS DEL DESARROLLO DE SOFTWARE**

Introducción

En esta unidad vamos a explorar los elementos esenciales del desarrollo de software. Este viaje nos llevará desde los fundamentos básicos, como la diferencia entre software y programas, hasta conceptos más avanzados, como los lenguajes de programación y las máquinas virtuales.

Esta unidad está diseñada para proporcionarles una sólida base teórica y práctica.

Abordaremos:

• Software y Programa: ¿Son lo mismo? ¿Y cuáles son los diferentes tipos de software que existen?

• Relación Hardware-Software: ¿Cómo interactúan entre sí y qué necesitas saber para ser un desarrollador eficaz?

• Desarrollo de Software: Aquí nos sumergiremos en los ciclos de vida del software y las herramientas que facilitan su desarrollo.

• Lenguajes de Programación: Descubrirás los diferentes tipos de lenguajes y cómo se aplican en el mundo real.

• Fases en el Desarrollo y Ejecución del Software: Desde el análisis hasta el mantenimiento, pasando por la codificación y las pruebas.

Para cada uno de estos temas, no sólo aprenderás la teoría, sino que también veremos ejemplos prácticos, tips y mejores prácticas en la industria.

**1. | Software y Programa. Tipos de Software |**

En este segmento, vamos a profundizar en la definición y las diferencias entre "Software" y "Programa." Además, exploraremos los diferentes tipos de software que existen y cuáles son sus aplicaciones específicas en el mundo real.

Este apartado se divide en dos partes clave:

• Software y Programa: ¿Alguna vez te has preguntado si estos términos son intercambiables? Vamos a clarificar las diferencias y similitudes.

• Tipos de Software: Hay diversos tipos de software, cada uno con un propósito y funcionalidad única. Aprenderemos a categorizarlos y entender cómo se usan en diferentes escenarios.

Este conocimiento es fundamental para cualquier persona que desee sumergirse en el campo del desarrollo de software, ya que tener una comprensión clara de estos conceptos básicos establecerá una base sólida para los temas más avanzados que se tratarán más adelante en el curso.

**1.1. | Software y Programa |**

**Software:** Es el conjunto integral de instrucciones y datos que permiten a un ordenador realizar una amplia gama de tareas. Desde ejecutar los procesos básicos para operar la máquina hasta posibilitar la funcionalidad de aplicaciones más avanzadas. El software es el cerebro detrás de cualquier operación que realices en tu computadora. Se escribe en lenguajes de programación que luego se compilan en un formato que la máquina pueda interpretar y ejecutar. Es crucial tener en cuenta que el software no es simplemente el código fuente; también abarca las bibliotecas, las bases de datos y otros archivos necesarios para que el software funcione de manera efectiva.

**Programa:** A diferencia del software, que es un término más amplio y abarcativo, un programa es una unidad individual dentro del software que lleva a cabo una tarea específica. Por ejemplo, un navegador web es un programa diseñado para navegar por Internet, y es parte del software general de tu sistema que permite la funcionalidad de la red. Un programa se ejecuta para realizar una operación específica o un conjunto de operaciones, guiado por un conjunto específico de instrucciones.

Diferencias y Similitudes

Diferencia: El software es una entidad más compleja y puede incluir múltiples programas y componentes, cada uno diseñado para una función específica. Un programa es una sola aplicación enfocada en una tarea en particular.

Similitud: Ambos son fundamentales para el funcionamiento de sistemas informáticos y usualmente se desarrollan utilizando lenguajes de programación.

**1.2. | Tipos de Software |**

• Software de sistema: Este tipo de software actúa como la columna vertebral de un sistema informático. Controla todo, desde la inicialización del hardware durante el arranque hasta la gestión de recursos como la memoria y la CPU. Los sistemas operativos como Windows, MacOS y Linux son ejemplos comunes de software de sistema.

• Software de aplicación: Este software es el que utilizas para tareas específicas. Está diseñado para resolver problemas particulares o facilitar actividades concretas, como procesar texto, editar imágenes o jugar videojuegos. Por ejemplo, Microsoft Word es un software de aplicación diseñado para ayudar en la creación de documentos.

• Software de programación: Son las herramientas que los desarrolladores usan para crear otros tipos de software. Esto abarca desde compiladores y depuradores hasta entornos de desarrollo integrado (IDEs), que son plataformas que proporcionan múltiples herramientas en una sola interfaz para facilitar el proceso de desarrollo de software.

**2. | Relación Hardware-Software |**

En la vida diaria, nos encontramos con múltiples dispositivos electrónicos como smartphones, ordenadores y tablets. ¿Te has preguntado alguna vez cómo funcionan? ¿Cómo saben qué hacer cuando presionas un botón o tocas la pantalla? Todo esto es posible gracias a una interacción compleja pero armoniosa entre el hardware y el software.

**Hardware y Software: Dos Lados de la Misma Moneda**   
Imagina que tu ordenador es como un equipo de fútbol. El hardware sería los jugadores en el campo, físicos y tangibles. Sin embargo, incluso los mejores jugadores necesitan una estrategia para ganar; aquí es donde entra el software, que sería el conjunto de tácticas y jugadas diseñadas por el entrenador.

**Herramientas y la Interacción con el Hardware**

Cuando programas, escribes código fuente en un lenguaje de programación específico. Este código luego se traduce a un "idioma" que el hardware puede entender, gracias a herramientas como los traductores de lenguajes. Estas herramientas se encargan de convertir tu código en instrucciones específicas para el hardware.

Por ejemplo, si estás programando una aplicación para controlar la intensidad de la luz en una habitación, el software enviará comandos al hardware que controla el brillo de las bombillas. Todo esto ocurre de manera coordinada y casi instantánea, lo que nos permite tener una experiencia fluida.

**Máquinas Virtuales: El Puente Entre Dos Mundos**

A veces, el software debe funcionar en diferentes tipos de hardware. Aquí es donde entran las máquinas virtuales. Imagina que sabes hablar español, pero te encuentras en un país donde se habla inglés. Podrías usar un traductor (máquina virtual) para entender y ser entendido. De la misma manera, las máquinas virtuales actúan como un intermediario que permite que el software se ejecute en diferentes tipos de hardware sin tener que modificarlo.

**3. | Desarrollo de software |**

Entramos en el mundo del desarrollo de software, la columna vertebral de casi todas las tecnologías que usamos hoy en día. Desde aplicaciones móviles y videojuegos hasta sistemas de gestión empresarial y plataformas de redes sociales, el desarrollo de software es el proceso creativo y técnico que da vida a estas soluciones digitales. Este apartado se centra en desentrañar los misterios del desarrollo de software, proporcionándote una visión completa de cómo se transforman las ideas en aplicaciones funcionales que impactan en nuestro día a día.

A lo largo de este apartado, exploraremos dos aspectos clave que son fundamentales para cualquier persona interesada en la programación y el desarrollo de software. Primero, nos sumergiremos en los "Ciclos de Vida del Software", donde aprenderás sobre las fases críticas que todo proyecto de software debe pasar, desde su concepción hasta su mantenimiento. Este conocimiento te ayudará a entender cómo se estructuran los proyectos de software y cómo puedes contribuir de manera efectiva en cada etapa.

Luego, abordaremos las "Herramientas de Apoyo al Desarrollo del Software". Aquí descubrirás las diversas herramientas que facilitan el proceso de desarrollo, haciendo que las tareas complejas sean más manejables y permitiendo que los equipos colaboren de manera más eficiente.

**3.1. | Ciclos de Vida del Software |**

¿Sabes cómo se construye una casa? No se empieza por el tejado ni se colocan las ventanas antes que los cimientos. Del mismo modo, el software sigue un proceso estructurado que se conoce como "Ciclo de Vida del Software".

Aunque existen varios modelos, todos ellos incluyen fases como:

Análisis: Es como un arquitecto planificando la casa. Se estudian las necesidades para determinar qué es lo que el software debe hacer.

Diseño: Aquí decidimos cómo se va a hacer. Se establecen las tecnologías a usar, la arquitectura del software, etc.

Codificación: Es el momento de construir. Los programadores escriben el código que hace que el software funcione.

Pruebas: Antes de que la casa sea habitable, hay que asegurarse de que todo funciona como debe. Se realizan distintos tipos de pruebas para garantizar que el software no tenga errores.

Despliegue o Explotación: Es como entregar las llaves de la casa. El software se instala y se pone en funcionamiento.

Mantenimiento: Una casa necesita reparaciones y mejoras, y lo mismo sucede con el software. Se realizan actualizaciones para corregir errores o añadir nuevas funciones.

**3.2. | Herramientas de Apoyo al Desarrollo del Software |**

Desarrollar software no es tarea de un solo programa o lenguaje. Se

utilizan diversas herramientas que ayudan en diferentes etapas del

desarrollo. Algunas de las más comunes son:

• Entornos de Desarrollo Integrados (IDEs): Son como tu estación de trabajo, donde escribes y pruebas el código.

• Control de Versiones: Imagina que estás escribiendo un libro con varios capítulos y de repente quieres volver a una versión anterior.   
Herramientas como Git permiten hacer esto con el código.

• Depuradores: Estos son tus salvavidas cuando algo va mal. Te ayudan a encontrar y corregir errores en el código.

• Automatización de Pruebas: Estas herramientas te permiten probar automáticamente distintas partes del software para asegurarte de que todo funcione correctamente.

**4. | Lenguajes de programación |**

Los programas informáticos son escritos en lenguajes de programación, que actúan como puentes entre la lógica humana y la mecánica del hardware de la computadora. Esencialmente, un lenguaje de programación es un idioma artificial compuesto de un conjunto de símbolos y reglas. Estas reglas se aplican sobre un alfabeto específico para crear códigos que la máquina pueda comprender y ejecutar. En este contexto, los lenguajes de programación son las herramientas que nos permiten instruir a la computadora para que realice tareas específicas según nuestras necesidades.

La importancia de seleccionar el lenguaje de programación adecuado para un proyecto no puede subestimarse. Cada lenguaje tiene su propio conjunto de símbolos y estructuras, y algunos están mejor adaptados para ciertas tareas o áreas que otros. Además, la elección del lenguaje afecta no solo la eficiencia del código sino también su portabilidad y la facilidad de desarrollo.

Los lenguajes de programación han evolucionado a lo largo del tiempo para facilitar la labor de programación y adaptarse a diferentes tipos de necesidades.

1. Inicialmente, se programaba en "lenguaje máquina", que era el único lenguaje que las máquinas podían entender directamente, pero era extremadamente difícil para los humanos.
2. Luego vino el "lenguaje ensamblador", que simplificaba el proceso pero aún era bastante técnico y centrado en el hardware.
3. Los "lenguajes de alto nivel" llegaron más tarde, y son los que utilizamos más comúnmente hoy en día; estos están diseñados para ser más comprensibles para los humanos.
4. Y en los tiempos más recientes, tenemos los "lenguajes visuales", que permiten programar a través de interfaces gráficas, generando automáticamente el código necesario.

**4.1. | Concepto y Características de los Lenguajes de Programación |**

Concepto de Lenguaje de Programación

Un lenguaje de programación es un conjunto específico de reglas y directrices, que permiten a los programadores comunicarse con las computadoras para llevar a cabo tareas particulares. Estas reglas se componen de tres elementos clave:

• Alfabeto: Este es el conjunto de símbolos o caracteres que el lenguaje permite utilizar.

• Sintaxis: Estas son las reglas que dictan cómo se deben ordenar y combinar estos símbolos para crear instrucciones o programas válidos.

• Semántica: Este elemento provee el significado de las combinaciones de símbolos, es decir, las acciones que se realizarán al ejecutar estas combinaciones.

Características

Los lenguajes de programación presentan una serie de características distintivas que los hacen únicos y determinan su aplicabilidad para diferentes tipos de proyectos. Algunas de estas características son:

* **Nivel de Abstracción**:

o Lenguajes de alto nivel: Son más cercanos al razonamiento humano y más fáciles de leer y escribir.

o Lenguajes de bajo nivel: Son más próximos al hardware y ofrecen mayor control sobre la computadora, incluyendo el lenguaje máquina el lenguaje ensamblador.

* **Técnica de Programación Utilizada:**

o Lenguajes Estructurados: Utilizan la programación estructurada y son ejemplos de ello Pascal y C.

o Lenguajes Orientados a Objetos: Como C++ y Java, utilizan la técnica de programación orientada a objetos.

o Lenguajes Visuales: Permiten el desarrollo gráfico de aplicaciones, y generan automáticamente el código correspondiente. Algunos ejemplos son Visual Basic.Net y Borland Delphi.

* **Tipado:**

o Tipado Estático: Requiere la declaración del tipo de dato antes de su uso.

o Tipado Dinámico: Determina el tipo de dato en tiempo de ejecución.

* **Portabilidad:** Algunos lenguajes están diseñados para ser fácilmente transferibles entre diferentes sistemas, mientras que otros están más vinculados a un sistema específico.
* **Ecosistema y Bibliotecas:** La presencia de un ecosistema rico en bibliotecas y frameworks es a menudo un factor crítico en la selección de un lenguaje para un proyecto específico.

**4.2. | Lenguajes de Programación Estructurados |**

Los lenguajes de programación estructurados son una categoría de lenguajes de programación que se centran en claridad, eficiencia y facilidad para el análisis y la modificación del código. Estos lenguajes facilitan la construcción de programas utilizando bloques de código organizados en estructuras como bucles, condicionales e iteraciones.

Concepto

Un lenguaje de programación estructurado es aquel que está diseñado para facilitar el flujo de control en un programa sin necesidad de recurrir a instrucciones de salto como "GOTO", que pueden hacer que el código sea difícil de seguir y mantener. En lugar de saltos, estos lenguajes emplean estructuras de control definidas, como "if-else", "while", y "for", que mejoran la legibilidad y la lógica del código.

Características

• Claridad y Eficiencia: La estructura del código es fácil de seguir, lo que facilita tanto el desarrollo como la depuración y el mantenimiento del software.

• Modularidad: Permiten descomponer un programa en módulos o funciones más pequeñas que se pueden desarrollar y probar de forma independiente.

• Portabilidad: El código generalmente es más fácil de portar a diferentes sistemas operativos o arquitecturas de hardware.

Ejemplos de Lenguajes Estructurados

• Pascal: Diseñado con un fuerte enfoque en la enseñanza de programación estructurada.

• C: Ampliamente utilizado en el desarrollo de sistemas y aplicaciones, es conocido por su flexibilidad y eficiencia.

• Ada: Utilizado principalmente en sistemas críticos donde la seguridad y la estabilidad son cruciales.

Aplicaciones Prácticas

Un ejemplo práctico podría ser el desarrollo de un programa de gestión de inventario para una tienda. En un lenguaje estructurado como C, podría dividir el programa en diferentes módulos: uno para agregar nuevos elementos al inventario, otro para eliminar elementos, uno más para buscar elementos, etc. Cada módulo sería una función separada que se desarrolla, prueba y depura de forma independiente, facilitando la gestión del proyecto.

**4.3. | Lenguajes de Programación Orientados a Objetos |**

Los lenguajes de programación orientados a objetos (OOP, por sus siglas en inglés) representan una evolución en el paradigma de la programación. A diferencia de los lenguajes estructurados, que se centran en procedimientos y funciones, los lenguajes OOP enfatizan la organización del código alrededor de "objetos", que son entidades que combinan tanto datos como comportamientos.

Concepto

En la programación orientada a objetos, un "objeto" es una instancia de una "clase", que actúa como un modelo o plantilla. Las clases definen las propiedades (atributos) y los métodos (funciones) que sus objetos tendrán. Al programar, se crean, modifican, y se interactúa con estos objetos para realizar tareas específicas.

Características

• Encapsulamiento: Los datos y métodos que operan sobre esos datos están agrupados, ocultando la complejidad del funcionamiento interno.

• Herencia: Las clases pueden heredar atributos y comportamientos de clases previamente definidas, lo que promueve la reutilización del código.

• Polimorfismo: Un objeto puede adoptar múltiples formas, generalmente a través de la herencia y la sobrecarga de métodos, lo que hace que el código sea más flexible y fácil de gestionar.

Ejemplos de Lenguajes OOP

• Java: Utilizado ampliamente en aplicaciones empresariales, aplicaciones móviles y en la web.

• C++: Una extensión de C que añade capacidades de programación orientada a objetos, y es común en el desarrollo de software de sistemas.

• Python: Conocido por su simplicidad y legibilidad, Python también soporta OOP, y se usa en una variedad de aplicaciones desde desarrollo web hasta inteligencia artificial.

Aplicaciones Práctica

Suponga que está desarrollando un videojuego. Podría tener una clase llamada "Personaje" con atributos como salud, velocidad y habilidades. A partir de esta clase principal, podría derivar otras clases como "Guerrero", "Mago" o "Arquero", cada una con sus propias habilidades y atributos únicos. Al usar OOP, el código se vuelve más modular, más fácil de gestionar y extender, y más intuitivo en términos de cómo se organizan y se relacionan entre sí los diferentes elementos del juego.

**5. | Fases en el Desarrollo y Ejecución del Software |**

La creación de un software es un proceso complejo y multifacético que va mucho más allá de simplemente escribir código. Similar a cómo un arquitecto necesita un plano antes de construir un edificio, los desarrolladores de software requieren de un conjunto estructurado de pasos o "fases" para asegurar que el producto final sea robusto, eficiente y cumpla con los objetivos para los que fue diseñado.

¿Por qué son importantes las Fases?

Imaginemos que queremos construir una aplicación móvil para un restaurante que permita a los clientes hacer reservas, consultar el menú y hacer pedidos en línea. Sin un enfoque por fases, podríamos encontrarnos escribiendo fragmentos de código sin tener una visión clara de cómo todas las piezas encajarán entre sí. Este enfoque caótico no solo aumenta la probabilidad de errores, sino que también hace que el proyecto sea más difícil de gestionar y escalar.

• Análisis: Es donde determinamos qué necesita el restaurante y qué quieren los usuarios. Omitir esta fase podría llevarnos a desarrollar funciones que nadie va a utilizar.

• Diseño: Aquí definimos la arquitectura de la aplicación. Sin un diseño sólido, podríamos terminar con un código desordenado que es difícil de depurar o extender.

• Codificación: Es el acto de escribir el código en sí, pero sin las fases anteriores, podríamos encontrarnos codificando características que luego tendrían que ser desechadas o reescritas.

• Pruebas: En esta fase, aseguramos que el software funciona como se espera. Ignorar esta fase podría resultar en una aplicación llena de errores que frustran a los usuarios y dañan la reputación del restaurante.

• Documentación, Explotación y Mantenimiento: Estas fases aseguran que el software se pueda usar, entender y mantener a largo plazo. Sin ellas, incluso un proyecto bien codificado podría fallar eventualmente.

**5.1. | Análisis |**

El análisis es la primera y una de las etapas más cruciales en el ciclo de vida del desarrollo de software. En esta fase, se identifican y se documentan las necesidades y requisitos tanto del cliente como de los futuros usuarios del software. Aquí es donde se responden preguntas fundamentales como: ¿Qué problema estamos tratando de resolver? ¿Cuáles son las funcionalidades específicas que debe tener el software? ¿Qué limitaciones existen, como el presupuesto o la tecnología disponible?

Importancia del Análisis

No realizar un análisis exhaustivo es como tratar de construir una casa sin tener un plano detallado. Por ejemplo, si estamos desarrollando un sistema de gestión de inventario para una tienda, necesitamos saber el volumen de productos, cómo y cuándo se actualizan, quiénes son los usuarios del sistema, entre otras cosas. Ignorar estos detalles podría llevar a malas decisiones de diseño que harán que el software sea ineficaz o incluso inútil para sus usuarios.

Tipos de Análisis

• Análisis de Requisitos: Se recopila una lista detallada de las funcionalidades que el cliente y los usuarios finales esperan del

software. Esto puede incluir desde características muy generales hasta requisitos muy específicos.

• Análisis Funcional: Aquí se desglosan los requisitos en funciones más detalladas que el software debe ser capaz de realizar. Este es un mapa conceptual de cómo las diferentes partes del software interactuarán entre sí.

• Análisis Técnico: En esta subfase, se examina la viabilidad técnica de los requisitos. Esto implica evaluar las tecnologías, frameworks y lenguajes de programación más adecuados para desarrollar el software.

Ejemplo Práctico

Supongamos que se nos ha pedido desarrollar una aplicación para realizar seguimiento del progreso del ejercicio físico. En la fase de análisis, identificaríamos requisitos como la necesidad de rastrear diferentes tipos de actividades (caminar, correr, nadar), la capacidad para sincronizar con dispositivos de monitoreo de actividad física y requisitos de seguridad para proteger la información del usuario. Sin un análisis detallado, podríamos terminar con una aplicación que no cumple con las expectativas del usuario, que es difícil de usar o que carece de funcionalidades esenciales.

**5.2. | Diseño |**

Una vez que se han identificado y documentado las necesidades y requisitos en la fase de análisis, el siguiente paso en el desarrollo de software es el diseño. Esta etapa es donde se esbozan las especificaciones técnicas y se elaboran los planos arquitectónicos del software.

Importancia del Diseño

El diseño es al desarrollo de software lo que los planos de arquitectura son a la construcción de un edificio. Si en la fase de análisis identificamos qué queremos construir, en la fase de diseño determinamos cómo lo construiremos. Aquí se toman decisiones críticas que afectarán todo el ciclo de vida del software, desde su desarrollo hasta su mantenimiento.

Componentes del Diseño

• Diseño Arquitectónico: Aquí se decide la estructura general del software. Se define cómo se descompondrá el sistema en subcomponentes y cómo estos subcomponentes interactuarán entre sí.

• Diseño de Interfaz: Este aspecto se centra en cómo los usuarios interactuarán con el software. Esto incluye el diseño de la interfaz de usuario (UI) y la interfaz de programación de aplicaciones (API).

• Diseño de Base de Datos: Si el software necesita almacenar datos, esta subfase implica la planificación de la estructura de la base de datos, cómo se accederá a ella y cómo se mantendrá.

Ejemplo Práctico

Si tomamos el ejemplo anterior de la aplicación para hacer seguimiento del progreso del ejercicio físico, el diseño podría incluir la creación de varios módulos: uno para el seguimiento de actividades, otro para la sincronización con dispositivos externos, y otro para la gestión de cuentas de usuario. En el diseño de la interfaz, podríamos planificar pantallas específicas para la visualización de estadísticas, el inicio de sesión y la configuración del perfil del usuario. Y en el diseño de la base de datos, planificaríamos cómo almacenar y recuperar eficientemente la información relacionada con las actividades físicas, las preferencias del usuario y los registros históricos.

Herramientas de Diseño

Existen diversas herramientas que facilitan el diseño, como software de diagramación (UML), herramientas de diseño de interfaz y software de gestión de bases de datos. Estas herramientas ayudan a visualizar, modelar y documentar las especificaciones del diseño de manera efectiva.

**5.3. | Codificación |**

Después de haber establecido un diseño sólido, llega la fase de codificación, que es donde realmente comienza a construirse el software. Aquí, los desarrolladores escriben el código fuente siguiendo las especificaciones de diseño y utilizando el lenguaje de programación adecuado para el proyecto.

Importancia de la Codificación

La codificación es la etapa que da vida a las ideas y conceptos desarrollados en las fases anteriores. Sin código bien escrito y eficiente, incluso el mejor diseño puede fallar en su ejecución. Es el código el que ejecutará las funciones y ofrecerá las funcionalidades deseadas al usuario.

Tipos de Código

• Código Fuente: Este es el conjunto de instrucciones escritas en un lenguaje de programación que será luego compilado o interpretado para generar el programa ejecutable. Ejemplo: Código en Python para una calculadora simple.

• Código Objeto: Este es el código que resulta después de que el código fuente ha sido compilado por un compilador. Aunque no es directamente legible por humanos, es un paso crucial antes de llegar al código ejecutable.

• Código Ejecutable: Es el archivo final que puede ser ejecutado en un sistema operativo. Este archivo contiene instrucciones que la máquina puede entender y ejecutar directamente.

• Código de Script: Estos son fragmentos de código que se ejecutan en un entorno interpretado en lugar de ser compilados. Scripts en lenguajes como Javascript o Python a menudo manejan tareas específicas dentro de una aplicación más grande.

Ejemplo práctico

Si seguimos con nuestro ejemplo de la aplicación para el seguimiento del progreso en el ejercicio físico, la codificación implicaría escribir el código para cada uno de los módulos diseñados: seguimiento de actividades, sincronización con dispositivos y gestión de cuentas de usuario. Cada uno de estos módulos podría contener varios tipos de código, desde el código fuente que define la lógica del programa hasta scripts para validar la entrada del usuario.

Herramientas de Codificación

Para facilitar la tarea de codificación, los desarrolladores suelen emplear entornos de desarrollo integrados (IDEs), que ofrecen herramientas para escribir, depurar y probar el código de manera más eficiente. Ejemplos populares incluyen Visual Studio, Eclipse y PyCharm.

**5.4. | Fases en la obtención de código |**

La obtención de un programa ejecutable es un proceso multifásico que comienza con la escritura del código fuente y pasa por varias etapas antes de generar el código ejecutable. Cada etapa tiene un propósito específico y es crucial para el funcionamiento correcto del software final. Vamos a abordar estas fases una por una.

Fuente

El Código Fuente es donde todo comienza. Es el conjunto de instrucciones escritas en un lenguaje de programación específico, como Python, Java, C++, entre otros. Este código es legible y comprensible por los humanos y generalmente se escribe utilizando un editor de texto especializado o un Entorno de Desarrollo Integrado (IDE).

Ejemplo: Imaginemos que estamos desarrollando una aplicación para gestionar reservas en un restaurante. En esta fase, escribiremos el código que maneja la interfaz de usuario, las consultas a la base de datos y la lógica para gestionar las reservas.

Objeto

Una vez que el código fuente está completo, el siguiente paso es la compilación. Durante esta fase, un compilador toma el código fuente y lo convierte en Código Objeto. Este es un archivo binario que no es directamente ejecutable, pero es un paso crucial en el proceso de creación del código ejecutable.

Ejemplo: En nuestra aplicación de reservas para restaurantes, el compilador tomará el código que hemos escrito y generará archivos de código objeto para cada uno de los módulos o clases que hemos definido.

Ejecutable

Finalmente, los archivos de código objeto son enlazados juntos en la fase de enlazado para crear un archivo ejecutable. Este es el programa

que los usuarios finales ejecutarán en sus sistemas.

Este archivo contiene todo el código necesario para que la aplicación funcione, incluyendo bibliotecas y dependencias.

Ejemplo: Volviendo a la aplicación de reservas, en esta fase se generarán los archivos .exe o .app (dependiendo del sistema operativo) que los usuarios finales descargarán e instalarán en sus sistemas para hacer y gestionar reservas en el restaurante.

**5.5. | Máquinas Virtuales |**

En el mundo del desarrollo de software, uno de los desafíos más notables es cómo hacer que las aplicaciones sean compatibles con diferentes sistemas operativos y arquitecturas de hardware. Aquí es donde las máquinas virtuales (MV) y los entornos de ejecución entran en juego como soluciones integrales. Estos elementos actúan como puentes entre el código fuente que escribimos y la infraestructura de hardware subyacente, permitiendo que el software se ejecute de forma uniforme en diferentes plataformas.

Las Máquinas Virtuales no son solo una capa de abstracción; son ecosistemas completos que incluyen no solo la máquina virtual en sí sino también una serie de bibliotecas, frameworks y entornos de ejecución que ayudan en la creación y despliegue del software. Estos componentes adicionales facilitan tareas como la gestión de memoria, la optimización de código y la interfaz con servicios del sistema operativo.

¿Por qué son necesarias las máquinas virtuales?

* Portabilidad: Imagina que has desarrollado una aplicación que quieres que funcione tanto en Windows como en macOS. En lugar de escribir versiones específicas del código para cada sistema, puedes escribir una versión que funcione en una máquina virtual compatible con ambos.
* Gestión de Recursos: Las máquinas virtuales pueden gestionar recursos como la memoria y el tiempo de CPU de manera más eficiente, lo que puede llevar a un mejor rendimiento y menor uso de recursos.
* Seguridad: Las máquinas virtuales a menudo incluyen capas de seguridad adicionales, como sandboxing, que pueden ayudar a proteger el sistema anfitrión contra código malicioso.
* Facilidad de Despliegue y Mantenimiento: Las actualizaciones y el mantenimiento se simplifican ya que se realizan en un único entorno, en lugar de tener que ajustar el código para cada tipo de hardware o sistema operativo.

Ejemplo ilustrativo

Supongamos que estás trabajando en una aplicación de gestión de inventario. Los empleados usarán esta aplicación en diferentes sistemas operativos: algunos en Windows, otros en macOS y algunos incluso en Linux. En lugar de crear tres versiones diferentes de tu aplicación, puedes escribir el código una sola vez y hacer que se ejecute en una máquina virtual compatible con todos estos sistemas. Esto no solo te ahorra tiempo, sino que también reduce el margen de error y las complicaciones de mantenimiento.

**5.5.1. | Frameworks |**

En el contexto de la programación y el desarrollo de software, un "framework" es básicamente un marco de trabajo: una estructura o conjunto de prácticas y tecnologías que proporcionan una base sobre la cual construir. Un framework no es un programa en sí mismo; en cambio, es una colección predefinida de bibliotecas y herramientas que sirve como un esqueleto estructural para tu proyecto.

¿Por qué usar un Framework?

• Consistencia: Los frameworks ofrecen un conjunto uniforme de prácticas y directrices que aseguran que todos los desarrolladores en un equipo estén en la misma página.

• Eficiencia: Ahorran tiempo y esfuerzo en tareas rutinarias, como la conexión a bases de datos, el manejo de sesiones o la construcción de interfaces de usuario.

• Escalabilidad: Están diseñados con la escalabilidad en mente, lo que facilita la expansión de tu aplicación en el futuro.

• Seguridad: Incluyen varias características de seguridad de forma predeterminada, lo que reduce el riesgo de errores comunes de seguridad.

• Comunidad: Popularidad significa una gran comunidad de desarrolladores, lo que se traduce en más documentación, más soluciones a problemas comunes y más bibliotecas de terceros.

Tipos de Código

• Web Frameworks: Están diseñados para ayudar en el desarrollo de aplicaciones web. Ejemplos incluyen Django para Python, Rails para Ruby, y Angular para JavaScript.

• Mobile Frameworks: Facilitan el desarrollo de aplicaciones móviles y pueden incluir herramientas para funciones como el acceso a sensores o la cámara. Ejemplos incluyen React Native y Flutter.

• Game Frameworks: Estos frameworks, como Unity y Unreal Engine, ofrecen todo lo necesario para el desarrollo de videojuegos, desde gráficos hasta la física del juego.

• Backend Frameworks: Son útiles para crear la lógica y las operaciones del lado del servidor de una aplicación. Ejemplos son Node.js para JavaScript o Spring Boot para Java.

Ejemplo práctico

Imagina que estás construyendo una aplicación para una tienda en línea. Un framework como Django te proporcionaría herramientas para manejar cosas como el carrito de compras, la autenticación de usuarios, y la gestión de inventario, lo que te permitiría centrarte en características únicas y personalizadas para tu tienda.

**5.5.2. | Entornos de ejecución |**

Un entorno de ejecución es un estado del software que proporciona todo lo que un programa necesita para funcionar correctamente una vez que se ha iniciado. Esto incluye recursos como bibliotecas, variables de entorno, y configuraciones. El entorno de ejecución actúa como un intermediario entre el sistema operativo y el programa, permitiendo que el código se ejecute de manera eficiente y segura.

¿Por qué son importantes los Entornos de Ejecución?

• Aislamiento: Los entornos de ejecución pueden aislar el programa de otros procesos en el sistema, lo que es especialmente útil para garantizar que no haya interferencias ni vulnerabilidades de seguridad.

• Gestión de Recursos: Ayudan en la administración efectiva de recursos como la memoria y el procesador, lo que resulta en un mejor rendimiento de la aplicación.

• Compatibilidad: Un entorno de ejecución bien diseñado puede permitir que un programa se ejecute en diferentes sistemas operativos o plataformas sin necesidad de modificaciones.

• Facilita el Despliegue: Los entornos de ejecución a menudo vienen con herramientas para facilitar la instalación y el despliegue de aplicaciones, lo que simplifica la vida tanto para los desarrolladores como para los usuarios finales.

Tipos de Entornos de Ejecución

• Navegadores Web: JavaScript se ejecuta principalmente en el entorno de ejecución del navegador web, que incluye todo lo necesario para interpretar y ejecutar código JS.

• JVM (Java Virtual Machine): Es el entorno de ejecución para el lenguaje Java, permitiendo que el código compilado se ejecute en cualquier sistema que tenga instalada una JVM.

• Node.js: Es un entorno de ejecución para JavaScript en el lado del servidor, lo que permite que JS se utilice para la lógica del servidor.

• Python Interpreter: Este es el entorno en el cual se ejecutan los scripts de Python. Viene con una biblioteca estándar y facilita la importación de módulos adicionales.

Ejemplo Práctico

Supongamos que estás desarrollando una aplicación de chat en tiempo real. Si utilizas Node.js como tu entorno de ejecución del servidor, podrías aprovechar sus características de manejo de eventos en tiempo real y su capacidad para manejar múltiples conexiones simultáneas. Este entorno de ejecución se asegurará de que todos los recursos necesarios estén disponibles y optimizados para tu aplicación.

**5.5.3. | Java Runtime Environment (JRE) |**

El Java Runtime Environment (Entorno de Ejecución de Java), comúnmente conocido por sus siglas JRE, es un componente específico que proporciona todo lo que una aplicación Java necesita para ejecutarse. El JRE es parte de la Java Development Kit (JDK), pero también se puede instalar de forma independiente.

¿Por qué usar un Framework?

• Java Virtual Machine (JVM): Es el motor que interpreta y ejecuta el bytecode de Java. Es el corazón del JRE y lo que permite la portabilidad entre diferentes sistemas operativos.

• Bibliotecas Java Estándar: Estas son un conjunto de clases precompiladas que proporcionan funcionalidades comunes, como operaciones de entrada/salida, gestión de cadenas y funciones matemáticas, entre otros.

• Archivos de Configuración: Estos archivos incluyen ajustes y parámetros que la JVM y las bibliotecas utilizan para el comportamiento y la optimización.

Importancia del JRE

• Portabilidad: Una de las mayores ventajas de Java es "Escribir una vez, correr en cualquier lugar". JRE es lo que hace posible esta portabilidad.

• Seguridad: JRE tiene varias capas de seguridad que ayudan a proteger las aplicaciones contra código malicioso y actividades no autorizadas.

• Rendimiento: La JVM dentro del JRE utiliza técnicas de optimización como la Just-In-Time compilation para mejorar el rendimiento del código Java.

• Facilidad de Uso: Para los usuarios, la instalación de un JRE compatible es generalmente todo lo que se necesita para ejecutar una aplicación Java, lo que simplifica el proceso de despliegue y uso.

Ejemplo práctico

Imagina que has desarrollado una aplicación Java para gestionar un pequeño negocio de ventas en línea. Los usuarios simplemente necesitarán tener el JRE adecuado instalado en sus sistemas para poder ejecutar tu aplicación. No tendrán que preocuparse por el sistema operativo o las especificaciones del hardware, gracias a la portabilidad que el JRE proporciona.

**5.6. | Pruebas |**

Una vez que el código de un programa ha sido escrito, ¿cómo sabemos si realmente hace lo que se supone que debe hacer? Aquí es donde entran en juego las pruebas de software. Las pruebas son una etapa fundamental en el ciclo de vida del desarrollo de software, y su objetivo principal es evaluar la funcionalidad, rendimiento y seguridad de un programa.

¿Por qué son necesarias las pruebas?

• Verificación y Validación: Las pruebas permiten verificar que cada parte del código cumpla con los requisitos funcionales y de diseño establecidos en las etapas de análisis y diseño. También validan que el programa en su conjunto satisfaga las necesidades del usuario final.

• Identificación de Errores y Defectos: Aunque es posible que un programa se ejecute sin errores aparentes, eso no significa que esté libre de defectos. Las pruebas ayudan a identificar estos problemas antes de que lleguen a los usuarios finales.

• Optimización del Rendimiento: Las pruebas no solo buscan errores, sino que también evalúan el rendimiento del software, como la velocidad de carga y la eficiencia en el uso de recursos.

• Seguridad: En un mundo donde las cuestiones de seguridad son de suma importancia, las pruebas permiten evaluar las vulnerabilidades y aplicar las medidas de seguridad necesarias.

Ejemplo ilustrativo

Imagina que has desarrollado una aplicación para gestionar las citas médicas en un hospital. Sin un proceso de pruebas adecuado, podrías terminar con una aplicación que duplica citas, pierde registros de pacientes o incluso viola la privacidad de los datos médicos. Al realizar pruebas exhaustivas, puedes descubrir estos y otros problemas y corregirlos antes de que la aplicación se implemente en un entorno real y sensible.

**5.7. | Documentación |**

La documentación puede verse a menudo como una tarea secundaria en el desarrollo de software, pero en realidad, es una fase crítica que merece tanto cuidado y atención como el propio código. La documentación es esencialmente el manual de usuario del código, proporcionando una "hoja de ruta" que aclara cómo funciona un programa, cómo interactuar con él y cómo resolver problemas que puedan surgir.

¿Por qué es necesaria la Documentación?

• Claridad y Comprensión: Un código bien documentado es más fácil de entender, lo que facilita tanto el mantenimiento como las mejoras futuras.

• Colaboración: Cuando se trabaja en equipo, la documentación actúa como un recurso compartido que ayuda a todos los miembros a entender el proyecto más rápidamente y a colaborar de manera más eficiente.

• Soporte y Mantenimiento: Una documentación detallada puede servir como una valiosa herramienta de diagnóstico cuando surgen problemas o es necesario actualizar el software.

• Onboarding: Los nuevos miembros del equipo o incluso los futuros desarrolladores que hereden el proyecto encontrarán invaluable una buena documentación, ahorrando tiempo y esfuerzo en el proceso de familiarización.

Ejemplo ilustrativo

Imagina que eres un nuevo desarrollador en un proyecto que ha estado en funcionamiento durante varios años. Sin documentación, te enfrentarías a la desalentadora tarea de descifrar miles de líneas de código para entender cómo funciona cada componente y cómo interactúan entre sí. Con una buena documentación, podrías ponerte al día mucho más rápidamente, comprendiendo los elementos clave del proyecto y cómo abordar tareas específicas.

**5.8. | Explotación |**

Una vez que un software ha sido desarrollado, probado y documentado, entra en la fase de explotación, que es cuando el software se pone en uso operativo y comienza a generar valor real. Esta etapa implica todo, desde la instalación y configuración inicial hasta el monitoreo continuo, actualizaciones y soporte al usuario.

¿Por qué es Importante la Fase de Explotación?

• Generación de Valor: Es en la fase de explotación donde el software demuestra su utilidad, ya sea resolviendo problemas, automatizando tareas o generando ingresos.

• Retroalimentación del Usuario: Una vez que el software está en manos de los usuarios finales, se pueden obtener datos y comentarios vitales que podrían llevar a futuras mejoras y ajustes.

• Evaluación de Rendimiento: Esta etapa también es crítica para evaluar cómo el software se desempeña en condiciones del mundo real, lo cual puede ser muy diferente de las condiciones de prueba.

• Mantenimiento Activo: Durante la explotación, es común que el software necesite actualizaciones para corregir errores, añadir nuevas características o adaptarse a cambios en el entorno en el que opera.

Ejemplo ilustrativo

Supongamos que has creado una aplicación móvil para la gestión de tareas. En la fase de explotación, los usuarios instalarán la app, la usarán diariamente y, con suerte, la encontrarán útil. Aquí es donde podrías empezar a ver descargas en aumento, buenas críticas y quizás ingresos por compras dentro de la app. Pero también podrías enfrentar problemas no anticipados, como incompatibilidades con ciertos modelos de teléfonos, lo que requeriría soluciones rápidas.

**5.9. | Mantenimiento |**

La fase de mantenimiento es el conjunto de actividades que se realizan una vez que el software se encuentra en la etapa de explotación. Aunque muchas personas podrían pensar que el desarrollo de software termina una vez que se ha lanzado, la realidad es que el mantenimiento es una parte crucial y continua del ciclo de vida del software.

¿Por qué es Importante el Mantenimiento?

• Corrección de Errores: Ningún software es perfecto; siempre hay posibilidad de fallos o errores que no se detectaron durante la fase de pruebas.

• Actualizaciones y Mejoras: Las necesidades de los usuarios cambian y también lo hace la tecnología. Mantener el software actualizado es crucial para su éxito a largo plazo.

• Seguridad: Nuevas vulnerabilidades de seguridad pueden surgir con el tiempo, y el mantenimiento regular ayuda a parchear estos agujeros de seguridad.

• Compatibilidad: Con el lanzamiento de nuevos sistemas operativos y hardware, es esencial asegurarse de que el software sigue siendo compatible.

Ejemplo ilustrativo

Imaginemos que tu empresa ha desarrollado un software de gestión de inventario. Tras unos meses de explotación, los usuarios reportan problemas menores y sugerencias para nuevas funcionalidades. Además, se ha lanzado una nueva versión del sistema operativo que muchos de tus usuarios utilizan. Aquí es donde entra en juego el mantenimiento. Se programarán y realizarán actualizaciones para resolver los problemas reportados, implementar las nuevas funcionalidades y asegurar la compatibilidad con el nuevo sistema operativo.