**Formato para el desarrollo de componente formativo**

|  |  |
| --- | --- |
| **Programa de formación** | Desarrollo de soluciones informáticas aplicando inteligencia artificial |

|  |  |  |  |
| --- | --- | --- | --- |
| **Competencia** | **220501095.** Diseñar la solución de *software* de acuerdo con procedimientos y requisitos técnicos. | **Resultados de aprendizaje** | **220501095-02.** Codificar algoritmo de la solución del problema, de acuerdo con el lenguaje de programación seleccionado.  **220501095-06.** Adoptar arquitectura *software* de acuerdo con estándares para el desarrollo *web*.  **220501095-07.** Emplea las herramientas para el desarrollo de *software* según las necesidades de desarrollo colaborativo. |

|  |  |  |
| --- | --- | --- |
| **Número del componente formativo** | 04 | |
| **Nombre del componente formativo** | Desarrollo de *software*: programación, arquitectura y herramientas. | |
| **Breve descripción** | Este componente formativo aborda los fundamentos del desarrollo de *software*, explorando desde los conceptos básicos de programación hasta las herramientas y entornos de desarrollo modernos. Cubre arquitectura de *software*, control de versiones y prácticas colaborativas, proporcionando una base sólida para el desarrollo de aplicaciones robustas y mantenibles, sin adentrarse en programación orientada a objeto. | |
| **Palabras clave** | Arquitectura de *software*, control de versiones (Git), desarrollo colaborativo, entornos de desarrollo y estructuras de datos fundamentales. | |
| **Área ocupacional** | | Servicios |
| **Idioma** | | Español |

1. Tabla de contenidos

[1 Fundamentos de lenguajes de programación 4](#_Toc182762925)

[1.1 Introducción a los lenguajes de programación 4](#_Toc182762926)

[1.2 Sintaxis y semántica básica 5](#_Toc182762927)

[1.3 Variables, tipos de datos y estructuras de control 5](#_Toc182762928)

[1.4 *Arrays* y estructuras de datos fundamentales 6](#_Toc182762929)

[1.5 Funciones y modularidad básica 6](#_Toc182762930)

[2 Arquitectura y diseño de *software* 7](#_Toc182762931)

[2.1 Fundamentos de arquitectura de *software* 7](#_Toc182762932)

[2.2 Patrones de diseño estructurales básicos 8](#_Toc182762933)

[2.3 Arquitecturas *web* modernas 8](#_Toc182762934)

[2.4 APIs y servicios *web* RESTful 9](#_Toc182762935)

[2.5 Principios SOLID y buenas prácticas 9](#_Toc182762936)

[3 Control de versiones y desarrollo colaborativo 10](#_Toc182762937)

[3.1 Fundamentos de Git 10](#_Toc182762938)

[3.2 Flujos de trabajo con repositorios 11](#_Toc182762939)

[3.3 Branching y merging 11](#_Toc182762940)

[3.4 Herramientas colaborativas y gestión de proyectos 12](#_Toc182762941)

[3.5 Integración continua básica 12](#_Toc182762942)

[4 Herramientas y entornos de desarrollo 13](#_Toc182762943)

[4.1 IDEs y editores de código 13](#_Toc182762944)

[4.2 *Debugging* y *testing* básico 14](#_Toc182762945)

[4.3 Gestión de dependencias y paquetes 14](#_Toc182762946)

[4.4 *Deployment* y entornos de producción 15](#_Toc182762947)

[4.5 Seguridad básica en el desarrollo 15](#_Toc182762948)

[5 Síntesis 16](#_Toc182762949)

[6 Material complementario 19](#_Toc182762950)

[7 Glosario 21](#_Toc182762951)

[8 Referencias bibliográficas 23](#_Toc182762952)

1. Introducción

El desarrollo de *software* moderno requiere una comprensión profunda de principios fundamentales, arquitecturas robustas y herramientas colaborativas que permitan crear aplicaciones escalables y mantenibles. La capacidad de escribir código eficiente y trabajar en equipo son habilidades esenciales en el panorama tecnológico actual.

Este componente aborda de manera integral los fundamentos del desarrollo de *software*, partiendo desde los conceptos básicos de programación y estructuras de datos, hasta las arquitecturas modernas y entornos de desarrollo. Se exploran patrones de diseño estructurales, prácticas de control de versiones con Git, y herramientas esenciales para el desarrollo colaborativo, excluyendo intencionalmente la programación orientada a objetos, que se trata en otro componente.

A lo largo del material, se combina la teoría con ejemplos prácticos y visuales, proporcionando una base inicial para que los estudiantes desarrollen aplicaciones robustas y escalables. Se enfatiza la importancia de las buenas prácticas de desarrollo, la colaboración efectiva y el uso adecuado de herramientas modernas, elementos clave para el éxito en proyectos de *software* reales.

La integración de conceptos arquitectónicos con prácticas de desarrollo colaborativo asegura que los estudiantes no solo aprendan a escribir código, sino también a trabajar efectivamente en equipos de desarrollo. Como dice un principio fundamental en el desarrollo de *software*: "El código es más leído que escrito".

¡Te invitamos a explorar los fundamentos, arquitecturas y herramientas que te permitirán desarrollar *software* de calidad, trabajando colaborativamente y aplicando las mejores prácticas de la industria!

|  |
| --- |
| DI\_ Guion\_Introduccion\_Video\_CF04\_ 228141 |

1. Desarrollo de contenidos

**Desarrollo de *software*: programación, arquitectura y herramientas**

# Fundamentos de lenguajes de programación

Los lenguajes de programación son la base fundamental para el desarrollo de *software* y constituyen el medio a través del cual los programadores pueden comunicar instrucciones precisas a las computadoras. En este capítulo exploraremos los conceptos esenciales que todo desarrollador debe comprender para iniciar su camino en la programación, independientemente del lenguaje específico que decida utilizar en el futuro. La comprensión sólida de estos fundamentos facilita el aprendizaje de cualquier lenguaje de programación y sienta las bases para el desarrollo de *software* robusto y mantenible.

## Introducción a los lenguajes de programación

Un lenguaje de programación es un sistema estructurado de comunicación que permite dar instrucciones a una computadora mediante un conjunto de reglas sintácticas y semánticas. Estos lenguajes han evolucionado significativamente desde sus inicios, pasando de ser muy cercanos al lenguaje de máquina a convertirse en herramientas más intuitivas y cercanas al lenguaje humano.

La historia de los lenguajes de programación es fascinante y refleja la evolución de la computación misma. Desde el código máquina puro hasta los modernos lenguajes de alto nivel, cada evolución ha buscado hacer la programación más accesible y productiva. Esta evolución continúa hoy en día, con nuevos lenguajes y paradigmas que emergen para satisfacer las necesidades cambiantes de la industria del *software*.

Los lenguajes de programación se pueden clasificar según diferentes criterios, siendo uno de los más comunes el nivel de abstracción que ofrecen. A continuación, se presenta una tabla comparativa de los principales tipos de lenguajes según su nivel de abstracción:

Tabla 1. Clasificación de lenguajes de programación

| **Nivel** | **Tipo de lenguaje** | **Características** | **Ejemplos** | **Casos de uso típicos** |
| --- | --- | --- | --- | --- |
| Bajo nivel | Lenguaje de máquina | Instrucciones binarias directas | Código binario (0s y 1s) | Programación de microprocesadores |
| Bajo nivel | Ensamblador | Representación simbólica del lenguaje máquina | Assembly x86, ARM | Drivers, sistemas embebidos |
| Medio nivel | Lenguajes estructurados | Balance entre abstracción y control del hardware | C | Sistemas operativos, aplicaciones de sistema |
| Alto nivel | Lenguajes modernos | Mayor abstracción, más cercanos al lenguaje natural | Python, JavaScript | Desarrollo *web*, aplicaciones empresariales |

Fuente: OIT, 2024.

**Texto alternativo:** La Tabla 1 se denomina «Clasificación de lenguajes de programación». Clasifica los lenguajes de programación en tres niveles según su cercanía al hardware. Los lenguajes de bajo nivel (como el ensamblador) son más cercanos al hardware, mientras que los de alto nivel (como Python) están más cerca del lenguaje humano, ofreciendo mayor abstracción y facilidad de uso. Los lenguajes de medio nivel ofrecen un equilibrio entre ambos.

## Sintaxis y semántica básica

La sintaxis de un lenguaje de programación define las reglas que determinan cómo se deben combinar los diferentes símbolos y palabras para formar expresiones y sentencias válidas. Al igual que en los idiomas naturales, cada lenguaje de programación tiene su propia sintaxis que debe ser respetada para que el código pueda ser interpretado correctamente.

La semántica, por otro lado, se refiere al significado de estas construcciones sintácticas. Una expresión sintácticamente correcta debe también tener sentido semántico para que el programa funcione como se espera. Por ejemplo, la expresión "5 + 3" es tanto sintáctica como semánticamente correcta, mientras que "5 + /" sería sintácticamente incorrecta, aunque los símbolos individuales sean válidos.

Es importante entender que la sintaxis puede variar significativamente entre diferentes lenguajes de programación, pero los conceptos semánticos suelen ser transferibles. Por ejemplo, un bucle while puede escribirse de manera diferente en Python que en JavaScript, pero el concepto fundamental de repetición condicional es el mismo en ambos lenguajes.

## Variables, tipos de datos y estructuras de control

Las variables son espacios de memoria que nos permiten almacenar datos temporalmente durante la ejecución de un programa. Cada variable tiene un nombre que la identifica y un tipo de dato que determina qué clase de información puede almacenar. Los tipos de datos básicos incluyen números enteros, números decimales, caracteres y valores booleanos.

El manejo adecuado de las variables y sus tipos de datos es necesario para escribir programas eficientes y libres de errores. En algunos lenguajes, los tipos de datos deben declararse explícitamente (tipado estático), mientras que en otros se infieren automáticamente (tipado dinámico). Cada enfoque tiene sus ventajas y desventajas en términos de seguridad y flexibilidad.

Las estructuras de control son construcciones del lenguaje que nos permiten alterar el flujo de ejecución de un programa. Las más fundamentales son las estructuras condicionales (if-else) y los bucles (while, for). Estas estructuras nos permiten tomar decisiones y repetir acciones según sea necesario, dotando a nuestros programas de la capacidad de responder de manera diferente según las circunstancias.

## *Arrays* y estructuras de datos fundamentales

Los *arrays*, también conocidos como arreglos o vectores, son estructuras de datos que nos permiten almacenar múltiples valores del mismo tipo en una única variable. Representan una de las estructuras de datos más básicas y fundamentales en programación, sirviendo como base para estructuras más complejas.

Una característica importante de los *arrays* es que permiten el acceso aleatorio a sus elementos mediante índices, lo que los hace muy eficientes para ciertas operaciones. Sin embargo, también tienen limitaciones, como su tamaño fijo en muchos lenguajes y la necesidad de mover elementos cuando se realizan inserciones o eliminaciones en posiciones intermedias.

En el desarrollo de *software*, la elección adecuada de las estructuras de datos es determinante para la eficiencia de nuestros programas. Además de los *arrays*, existen otras estructuras fundamentales como las listas enlazadas, pilas y colas, cada una con sus propias características y casos de uso óptimos. La comprensión de las ventajas y desventajas de cada estructura es esencial para escribir programas eficientes.

## Funciones y modularidad básica

La modularidad es un principio fundamental en la programación que nos permite dividir un programa en partes más pequeñas y manejables llamadas funciones. Una función es un bloque de código que realiza una tarea específica y puede ser reutilizado en diferentes partes del programa.

Las funciones nos ayudan a organizar mejor nuestro código, evitar la repetición y hacer nuestros programas más mantenibles. Cada función debe tener un propósito claro y bien definido, siguiendo el principio de responsabilidad única. Esto significa que una función debe hacer una sola cosa y hacerla bien.

El concepto de alcance (*scope*) es fundamental en el trabajo con funciones. Las variables pueden tener alcance global o local, y entender estas diferencias es importante para escribir código mantenible y evitar efectos secundarios no deseados. Las buenas prácticas generalmente favorecen el uso de variables locales y la minimización del alcance global.

El uso efectivo de funciones también facilita la depuración de errores, ya que podemos probar y verificar cada función de manera independiente. Además, las funciones bien diseñadas pueden ser reutilizadas en diferentes proyectos, lo que aumenta nuestra productividad como desarrolladores. La capacidad de escribir funciones claras, concisas y reutilizables es una habilidad esencial que todo programador debe desarrollar.

La comprensión profunda de estos conceptos fundamentales es esencial para cualquier desarrollador de *software*, ya que constituyen la base sobre la cual se construyen aplicaciones más complejas. En los siguientes capítulos, exploraremos cómo estos conceptos se aplican en el contexto de la arquitectura de *software* y el desarrollo de aplicaciones modernas.

# Arquitectura y diseño de *software*

La arquitectura de *software* constituye la columna vertebral de cualquier sistema informático exitoso. Este capítulo explora los principios fundamentales, patrones y prácticas que guían el diseño de sistemas robustos y escalables. La arquitectura no solo se trata de diagramas y estructuras, sino de decisiones que impactan directamente en la calidad, mantenibilidad y evolución del *software*.

## Fundamentos de arquitectura de *software*

La arquitectura de *software* es el conjunto de decisiones fundamentales sobre la organización de un sistema, incluyendo la selección de elementos estructurales, sus interfaces, comportamientos y la composición de estos elementos en subsistemas más grandes. Una buena arquitectura debe equilibrar múltiples aspectos como el rendimiento, la seguridad, la usabilidad y la mantenibilidad.

Figura 1. Componentes clave de la arquitectura de *software*

Fuente: OIT, 2024.

**Texto alternativo:** La Figura 1 se denomina «Componentes clave de la arquitectura de *software*». Es una infografía circular donde en el centro se encuentra el título "Arquitectura de *Software*" rodeado por cinco círculos interconectados que representan: escalabilidad, mantenibilidad, confiabilidad, seguridad y rendimiento. Los círculos se superponen en parte sobre el círculo central simbolizando la interrelación entre estos componentes.

Los atributos de calidad fundamentales que toda arquitectura debe considerar son:

* **Escalabilidad**: capacidad de manejar crecimiento en carga y usuarios
* **Mantenibilidad**: facilidad para realizar cambios y correcciones
* **Confiabilidad**: consistencia en el funcionamiento bajo condiciones diversas
* **Seguridad**: protección contra vulnerabilidades y amenazas
* **Rendimiento**: eficiencia en el uso de recursos y tiempo de respuesta

## Patrones de diseño estructurales básicos

Los patrones de diseño estructurales son soluciones probadas a problemas comunes en el diseño de *software*. Estos patrones nos proporcionan plantillas para resolver problemas que pueden reutilizarse en muchas situaciones diferentes. Por ejemplo, consideremos el patrón Fachada (Facade), que simplifica una interfaz compleja:

# Ejemplo de patrón Fachada

class SubsistemaA:

def operacion\_a(self):

return "Operación A"

class SubsistemaB:

def operacion\_b(self):

return "Operación B"

class Fachada:

def \_\_init\_\_(self):

self.subsistema\_a = SubsistemaA()

self.subsistema\_b = SubsistemaB()

def operacion\_simple(self):

resultado = []

resultado.append(self.subsistema\_a.operacion\_a())

resultado.append(self.subsistema\_b.operacion\_b())

return " + ".join(resultado)

## Arquitecturas *web* modernas

Las arquitecturas *web* modernas han evolucionado significativamente con la adopción de microservicios, contenedores y computación en la nube. La arquitectura monolítica tradicional ha dado paso a sistemas distribuidos más flexibles y escalables. Un aspecto medular es la separación de responsabilidades entre el *frontend* y el *backend*, permitiendo que cada componente evolucione de manera independiente.

La arquitectura de microservicios, por ejemplo, descompone una aplicación en servicios pequeños e independientes que se comunican a través de APIs bien definidas. Esto permite una mayor flexibilidad en el desarrollo, despliegue y escalamiento de componentes individuales. Sin embargo, también introduce complejidad en términos de coordinación, monitoreo y gestión de datos distribuidos.

## APIs y servicios *web* RESTful

Las APIs (Interfaces de Programación de Aplicaciones) RESTful se han convertido en el estándar de facto para la comunicación entre sistemas distribuidos. REST (*Representational State Transfer)* proporciona un conjunto de principios arquitectónicos que enfatizan la escalabilidad, la independencia entre componentes y la uniformidad de interfaces.

Los principios fundamentales de una API RESTful incluyen:

* **Interfaz uniforme:** uso consistente de URIs y métodos HTTP.
* **Sin estado:** cada solicitud contiene toda la información necesaria.
* **Cacheable:** las respuestas deben indicar si pueden ser cacheadas.
* **Sistema en capas:** el cliente no necesita saber la complejidad detrás de cada solicitud.

## Principios SOLID y buenas prácticas

Los principios SOLID son fundamentales para crear *software* mantenible y adaptable al cambio. Estos principios, introducidos por Robert C. Martin, proporcionan pautas para estructurar funciones y clases de manera efectiva. Aunque originalmente se concibieron para la programación orientada a objetos, sus conceptos subyacentes son valiosos para cualquier paradigma de programación.

Un ejemplo práctico de aplicación del Principio de Responsabilidad Única sería separar la lógica de negocio de la lógica de acceso a datos:

# Buena práctica: Separación de responsabilidades

class UsuarioRepositorio:

def obtener\_usuario(self, id):

# Lógica de acceso a base de datos

pass

class ValidadorUsuario:

def validar(self, usuario):

# Lógica de validación

pass

class ServicioUsuario:

def \_\_init\_\_(self, repositorio, validador):

self.repositorio = repositorio

self.validador = validador

def procesar\_usuario(self, id):

usuario = self.repositorio.obtener\_usuario(id)

self.validador.validar(usuario)

return usuario

La arquitectura y el diseño de *software* son disciplinas en constante evolución que requieren un equilibrio entre principios teóricos sólidos y consideraciones prácticas. En el siguiente capítulo, exploraremos cómo estos conceptos se aplican en el contexto del control de versiones y el desarrollo colaborativo.

# Control de versiones y desarrollo colaborativo

El control de versiones es una parte fundamental del desarrollo de *software* moderno, permitiendo a los equipos trabajar de manera eficiente y coordinada en proyectos complejos. Este capítulo explora las herramientas y prácticas que hacen posible el desarrollo colaborativo, con especial énfasis en Git, el sistema de control de versiones más utilizado en la actualidad.

## Fundamentos de Git

Git es un sistema de control de versiones distribuido que permite realizar un seguimiento de los cambios en el código fuente durante el desarrollo de *software*. A diferencia de los sistemas centralizados anteriores, Git proporciona a cada desarrollador una copia completa del repositorio, lo que permite trabajar de manera offline y realizar operaciones de manera más eficiente.

Los conceptos fundamentales de Git incluyen el área de trabajo (working directory), el área de preparación (staging area) y el repositorio local. Cada uno de estos espacios cumple un papel específico en el flujo de trabajo de Git:

Tabla 2. Espacios de trabajo en Git y sus características principales

| **Espacio** | **Descripción** | **Comandos principales** | **Uso típico** |
| --- | --- | --- | --- |
| Working Directory | Directorio actual donde se realizan los cambios | git status, git add | Desarrollo activo |
| Staging Area | Área intermedia donde se preparan los cambios | git add, git reset | Revisión pre-commit |
| Local Repository | Almacén de todos los commits realizados | git commit, git log | Historial de cambios |
| Remote Repository | Copia compartida en un servidor | git push, git pull | Colaboración |

Fuente: OIT, 2024.

**Texto alternativo:** La Tabla 2 se denomina «Espacios de trabajo en Git y sus características principales». Es una comparativa que muestra los cuatro espacios principales de trabajo en Git: Working Directory, Staging Area, Local Repository y Remote Repository. Para cada espacio se detalla su descripción, los comandos principales asociados y su uso típico. La tabla ayuda a entender cómo el código se mueve a través de los diferentes estados en Git, desde el desarrollo activo hasta la colaboración en equipo.

## Flujos de trabajo con repositorios

Los flujos de trabajo en Git definen cómo los equipos organizan sus procesos de desarrollo y colaboración. El flujo de trabajo más común es Git Flow, que establece una estructura clara para el manejo de características, correcciones y releases. Sin embargo, existen otros modelos como GitHub Flow o GitLab Flow, cada uno adaptado a diferentes necesidades y contextos de desarrollo.

Un flujo de trabajo típico involucra los siguientes pasos:

* Crear una rama para una nueva característica.
* Realizar cambios y commits locales.
* Publicar la rama en el repositorio remoto.
* Crear un pull request para revisión.
* Fusionar los cambios aprobados en la rama principal.

## Branching y merging

El branching (ramificación) es una de las características más poderosas de Git, permitiendo a los desarrolladores trabajar en diferentes características o correcciones de manera aislada. Cada rama representa una línea independiente de desarrollo que puede evolucionar sin afectar a las demás.

# Ejemplo de comandos comunes de branching y merging

git branch feature/nueva-funcionalidad # Crear nueva rama

git checkout feature/nueva-funcionalidad # Cambiar a la nueva rama

git commit -m "Implementación inicial" # Realizar cambios

git push origin feature/nueva-funcionalidad # Publicar rama

git checkout main # Volver a la rama principal

git merge feature/nueva-funcionalidad # Fusionar cambios

## Herramientas colaborativas y gestión de proyectos

El desarrollo moderno de *software* requiere más que solo control de versiones. Las herramientas colaborativas complementan a Git proporcionando capacidades adicionales para la gestión de proyectos, revisión de código y comunicación del equipo. Algunas herramientas populares incluyen:

* Sistemas de gestión de issues (Jira, GitHub Issues).
* Herramientas de revisión de código (GitHub Pull Requests, GitLab Merge Requests).
* Plataformas de documentación colaborativa (Confluence, Wiki).
* Herramientas de comunicación en tiempo real (Slack, Microsoft Teams).

## Integración continua básica

La integración continua (CI) es una práctica de desarrollo que requiere que los desarrolladores integren su código en un repositorio compartido varias veces al día. Cada integración se verifica mediante una construcción automatizada que incluye pruebas, lo que permite detectar errores rápidamente.

Los elementos clave de un sistema de CI incluyen:

* Un repositorio de código centralizado.
* Automatización de la construcción.
* Pruebas automatizadas.
* Un servidor de integración continua.
* Convenciones de trabajo acordadas.

Un *pipeline* básico de CI podría verse así:

# Ejemplo de *pipeline* CI básico

*pipeline*:

stages:

- build

- test

- deploy

build:

script:

- npm install

- npm run build

test:

script:

- npm run test

deploy:

script:

- if [ "$BRANCH" = "main" ]; then

npm run deploy

fi

El desarrollo colaborativo exitoso requiere una combinación de herramientas técnicas y prácticas organizacionales efectivas. La adopción de Git y las prácticas modernas de desarrollo colaborativo no solo mejora la calidad del código, sino que también facilita la escalabilidad de los equipos y proyectos.

En el próximo capítulo, exploraremos las herramientas y entornos de desarrollo que complementan estas prácticas colaborativas y nos ayudan a ser más productivos en nuestro trabajo diario como desarrolladores.

La clave para el éxito en el desarrollo colaborativo no solo radica en el dominio de las herramientas, sino también en la adopción de buenas prácticas de comunicación y coordinación entre los miembros del equipo. La documentación clara, las convenciones de código consistentes y los procesos bien definidos son tan importantes como las herramientas técnicas que utilizamos.

# Herramientas y entornos de desarrollo

El entorno de desarrollo es el espacio donde los programadores pasan la mayor parte de su tiempo, por lo que contar con las herramientas adecuadas y saber utilizarlas eficientemente es indispensable para la productividad. Este capítulo explora los componentes esenciales que conforman un entorno de desarrollo moderno y productivo.

## IDEs y editores de código

Los Entornos de Desarrollo Integrado (IDEs) y los editores de código son las herramientas fundamentales para escribir y modificar código. Mientras que los editores de código son más ligeros y versátiles, los IDEs ofrecen un conjunto más completo de funcionalidades integradas.

Figura 2. Estructura general de un IDE moderno

Fuente: OIT, 2024.

**Texto alternativo:** La Figura 2 se denomina «Estructura general de un IDE moderno». Infografía que representa la interfaz típica de un IDE moderno. Muestra una ventana principal dividida en tres paneles: un explorador de archivos a la izquierda, el editor de código en el centro, y un panel de herramientas a la derecha. Se destacan cuatro características principales: resaltado de sintaxis, autocompletado, depurador integrado y control de versiones.

## *Debugging* y *testing* básico

El *debugging* (depuración) es una habilidad esencial que todo desarrollador debe dominar. Las herramientas modernas de depuración proporcionan capacidades avanzadas para identificar y resolver problemas en el código. El proceso de *debugging* efectivo incluye:

* Identificación del problema.
* Reproducción consistente.
* Localización de la causa raíz.
* Implementación y verificación de la solución.

Las pruebas, por otro lado, ayudan a prevenir errores antes de que ocurran. Un conjunto básico de pruebas debe incluir:

# Ejemplo de pruebas unitarias básicas

import unittest

class CalculadoraTest(unittest.TestCase):

def setUp(self):

self.calc = Calculadora()

def test\_suma(self):

self.assertEqual(self.calc.suma(2, 3), 5)

self.assertEqual(self.calc.suma(-1, 1), 0)

def test\_division(self):

with self.assertRaises(ValueError):

self.calc.division(5, 0)

## Gestión de dependencias y paquetes

La gestión eficiente de dependencias es determinante para mantener proyectos de *software* saludables y actualizados. Las herramientas de gestión de paquetes como npm, pip o Maven nos permiten:

* Instalar y actualizar dependencias de manera consistente
* Mantener un registro de las versiones utilizadas
* Resolver conflictos entre dependencias
* Compartir configuraciones entre equipos

{

"name": "mi-proyecto",

"version": "1.0.0",

"dependencies": {

"express": "^4.17.1",

"lodash": "^4.17.21"

},

"devDependencies": {

"jest": "^27.0.6",

"eslint": "^7.32.0"

}

}

## *Deployment* y entornos de producción

El despliegue de aplicaciones requiere una comprensión clara de los diferentes entornos y sus requisitos específicos. Los entornos típicos incluyen:

Tabla 3. Características y propósitos de los entornos de desarrollo

|  |  |  |  |
| --- | --- | --- | --- |
| **Entorno** | **Propósito** | **Características** | **Consideraciones** |
| Desarrollo | Trabajo local | *Debugging* habilitado, hot reload | Performance no crítica |
| Pruebas | Verificación QA | Datos de prueba, monitoreo | Similar a producción |
| *Staging* | Pre-producción | Configuración idéntica a producción | Validación final |
| Producción | Uso real | Optimizado, seguro | Alta disponibilidad |

Fuente: OIT, 2024.

**Texto alternativo:** La Tabla 3 se denomina «Características y propósitos de los entornos de desarrollo». Describe los cuatro entornos principales en el ciclo de desarrollo de *software*: desarrollo, pruebas, *staging* y producción. Para cada entorno se especifica su propósito principal, características distintivas y consideraciones especiales. La tabla ayuda a comprender cómo evoluciona una aplicación desde el desarrollo local hasta su despliegue en producción, enfatizando las diferencias y requerimientos específicos de cada fase

## Seguridad básica en el desarrollo

La seguridad debe ser una consideración desde el inicio del desarrollo. Algunas prácticas básicas de seguridad incluyen:

* Validación de entradas.
* Sanitización de datos.
* Uso de HTTPS.
* Gestión segura de secretos.
* Actualizaciones regulares de dependencias.

# Ejemplo de validación básica de entradas

def procesar\_entrada(dato):

if not dato:

raise ValueError("El dato no puede estar vacío")

# Sanitización básica

dato = dato.strip()

# Validación de longitud

if len(dato) > 100:

raise ValueError("El dato es demasiado largo")

# Escapar caracteres especiales

dato = html.escape(dato)

return dato

La configuración y mantenimiento adecuados del entorno de desarrollo es fundamental para la productividad y la calidad del *software*. Aunque las herramientas específicas pueden variar según el proyecto o la organización, los principios básicos de desarrollo seguro, *testing* y gestión de dependencias son universales.

El dominio de estas herramientas y prácticas mejora nuestra eficiencia como desarrolladores y contribuye a la creación de *software* más robusto y mantenible. A medida que la tecnología evoluciona, es importante mantenerse actualizado con las nuevas herramientas y mejores prácticas que emergen en la comunidad de desarrollo.

# Síntesis

El diagrama representa la estructura integral del componente sobre desarrollo de *software*, centrado en los fundamentos, arquitectura, control de versiones y entornos de desarrollo. Partiendo del concepto central del desarrollo de *software*, se ramifica en cuatro áreas esenciales: fundamentos de lenguajes de programación, arquitectura de *software*, control de versiones y entornos de desarrollo. Cada una de estas áreas incorpora subtemas específicos que conforman los elementos fundamentales para comprender y aplicar eficazmente las prácticas modernas de desarrollo de *software*.

Esta organización ilustra el flujo lógico del proceso de desarrollo de *software*. Comienza con la comprensión de los fundamentos de programación, proporcionando una base sólida en sintaxis, control de flujo y estructuras de datos. Luego, profundiza en la arquitectura de *software*, abordando patrones de diseño y principios fundamentales que son esenciales para construir aplicaciones robustas y escalables. A continuación, se enfoca en el control de versiones con Git y las prácticas de desarrollo colaborativo, herramientas estratégicas para el trabajo en equipo y la gestión eficiente del código. Finalmente, aborda los entornos de desarrollo y las herramientas modernas, destacando la importancia del *testing*, *debugging* y las consideraciones de seguridad en el despliegue de aplicaciones.

El diagrama funciona como una hoja de ruta visual para comprender la estructura y el alcance del componente, permitiendo al estudiante visualizar rápidamente la progresión del aprendizaje y las conexiones entre los diferentes temas. Los elementos transversales (desarrollo colaborativo, buenas prácticas y herramientas modernas) se entrelazan con todas las áreas, enfatizando su importancia en todo el proceso de desarrollo. Se sugiere utilizarlo como referencia para organizar el estudio y entender cómo se integran los diversos aspectos en el desarrollo de *software* moderno, garantizando la creación de aplicaciones mantenibles y escalables en entornos profesionales.

Figura 3. Síntesis temática del componente

Fuente: OIT, 2024.

**Texto alternativo:** La Figura 3 se denomina «Síntesis temática del componente», es un mapa mental con el tema central “Desarrollo de *software*: programación, arquitectura y herramientas", del cual se desprenden cuatro ramas principales —fundamentos lenguajes, arquitectura *software*, control versiones y entornos desarrollo— cada una detallada con sus subtemas específicos.

1. Actividades didácticas

|  |  |
| --- | --- |
| Descripción de actividad didáctica | |
| Nombre de la Actividad | Prueba de conocimientos sobre desarrollo de *software*: programación, arquitectura y herramientas. |
| Objetivo de la actividad | Validar el conocimiento adquirido sobre desarrollo de *software*: programación, arquitectura y herramientas. Esto se trabaja a partir de un conjunto de preguntas con el propósito de buscar una dinámica de razonamiento ágil sobre opciones cerradas y reafirmar un conocimiento declarado dentro del componente. |
| Tipo de actividad sugerida | Interfaz de usuario gráfica  Descripción generada automáticamente |
| Archivo de la actividad  (Anexo donde se describe la actividad propuesta) | Actividad\_didactica\_CF05\_228141 |

# Material complementario

| **Tema** | **Referencia APA del Material** | **Tipo de material (Video, capítulo de libro, artículo, otro)** | **Enlace del recurso o archivo del documento o material** |
| --- | --- | --- | --- |
| 1. Fundamentos de lenguajes de programación | Ecosistema de Recursos Educativos Digitales SENA. (2022b, octubre 5). *Lenguajes de programación.* | Video | <https://www.youtube.com/watch?v=QpaLtzMsIFw> |
| 1. Fundamentos de lenguajes de programación | Ecosistema de Recursos Educativos Digitales SENA. (2021a, mayo 4). *Introducción a la programación de aplicaciones.* | Video | <https://www.youtube.com/watch?v=7bu6jnb5q8s> |
| 2 .Arquitectura y diseño de *software* | Ecosistema de Recursos Educativos Digitales SENA. (2021c, noviembre 26). *Diseño de patrones de software: introducción.* | Video | <https://www.youtube.com/watch?v=sQHRHhsRUoA> |
| 2. Arquitectura y diseño de *software* | Ecosistema de Recursos Educativos Digitales SENA. (2023c, marzo 25). *Introducción - Análisis, diseño y arquitectura de software.* | Video | <https://www.youtube.com/watch?v=gomlgvATlPU> |
| 2. Arquitectura y diseño de *software* | Ecosistema de Recursos Educativos Digitales SENA. (2022a, agosto 8). *Introducción a la arquitectura de software.* | Video | <https://www.youtube.com/watch?v=plwzcGedFpM> |
| 2. Arquitectura y diseño de *software* | Ecosistema de Recursos Educativos Digitales SENA. (2021b, julio 1). *Componentes de una arquitectura de software.* | Video | <https://www.youtube.com/watch?v=XrjY2iOVR8o> |
| 2. Arquitectura y diseño de *software* | Ecosistema de Recursos Educativos Digitales SENA. (2021d, diciembre 10). *Patrones de diseño de software.* | Video | <https://www.youtube.com/watch?v=ZufBcrlUqak> |
| 3. Control de versiones y desarrollo colaborativo | Platzi. (2024, 17 agosto). *La historia del control de versiones en el código* | Video | <https://www.youtube.com/watch?v=KiZRXFJbG98> |
| 4. Herramientas y entornos de desarrollo | Ecosistema de Recursos Educativos Digitales SENA. (2023b, marzo 24). *Entornos de desarrollo.* | Video | <https://www.youtube.com/watch?v=mZo3seNUNFU> |
| 4. Herramientas y entornos de desarrollo | Ecosistema de Recursos Educativos Digitales SENA. (2023a, marzo 23). *Construcción de algoritmos en un entorno de desarrollo (IDE).* | Video | <https://www.youtube.com/watch?v=U0CPktQHSYw> |

# Glosario

| **Término** | **Significado** |
| --- | --- |
| API (*Application Programming Interface*) | Conjunto de reglas y protocolos que permiten la comunicación entre diferentes componentes de *software* |
| Branch (Rama) | En control de versiones, una línea independiente de desarrollo que permite trabajar en características o correcciones sin afectar el código principal |
| *Commit* | Instantánea del código en un momento específico que registra cambios en el repositorio de control de versiones |
| *Debugging* | Proceso de identificación y corrección de errores en el código fuente |
| *Deploy* (Despliegue) | Proceso de poner una aplicación en producción o hacerla disponible para su uso |
| Git | Sistema de control de versiones distribuido diseñado para manejar proyectos de cualquier tamaño con velocidad y eficiencia |
| IDE (*Integrated Development Environment*) | Entorno de desarrollo integrado que proporciona herramientas comprensivas para la programación |
| *Merge* (Fusión) | Proceso de combinar cambios de diferentes ramas en el control de versiones |
| Microservicios | Arquitectura que estructura una aplicación como un conjunto de servicios pequeños e independientes |
| *Pipeline* | Secuencia automatizada de procesos para llevar el código desde el desarrollo hasta la producción |
| Pull Request | Solicitud para integrar cambios de una rama a otra, típicamente utilizada para revisión de código |
| REST | Estilo de arquitectura de *software* para sistemas distribuidos, especialmente usado en APIs *web* |
| Staging Area | En Git, área intermedia donde se preparan los cambios antes de confirmarlos |
| *Testing* | Proceso de evaluar un sistema o componente para verificar que cumple con los requisitos especificados |
| Versionamiento | Sistema para registrar cambios en archivos de código fuente a lo largo del tiempo |

# Referencias bibliográficas

Alur, S. J., Crupi, J., & Malks, D. (2023). Core J2EE Patterns: Best Practices and Design Strategies (3rd ed.). Prentice Hall.

Báez, M., & Brunner, P. (2022). Fundamentos de programación: Aprende a programar desde cero. Universidad Nacional Autónoma de México.

Chacon, S., & Straub, B. (2024). Pro Git (3rd ed.). Apress. https://git-scm.com/book/es/v2

Fowler, M. (2022). Patterns of Enterprise Application Architecture (2nd ed.). Addison-Wesley Professional.

García, A., & Martínez, J. (2023). Arquitectura de *software*: Fundamentos y patrones de diseño modernos. Revista Española de Ingeniería de *Software*, 15(2), 45-67.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2021). Design Patterns: Elements of Reusable Object-Oriented *Software* (Anniversary ed.). Addison-Wesley Professional.

Luján-Mora, S. (2023). Programación en Internet: Guía completa. Universidad de Alicante. http://gplsi.dlsi.ua.es/~slujan/programacion-internet/

Martin, R. C. (2021). Clean Architecture: A Craftsman's Guide to *Software* Structure and Design. Prentice Hall.

Pressman, R. S., & Maxim, B. R. (2024). Ingeniería del *Software*: Un Enfoque Práctico (9ª ed.). McGraw-Hill Education.

Rodríguez, P., & López, M. (2023). Control de versiones con Git: Manual práctico. Universidad Politécnica de Madrid. https://oa.upm.es/control-versiones/

Sommerville, I. (2021). *Software* Engineering (11th ed.). Pearson.

Sznajdleder, P. A. (2023). Algoritmos y estructuras de datos: Una perspectiva práctica con Python y Java. Alfaomega.

Tanenbaum, A. S., & Van Steen, M. (2023). Distributed Systems: Principles and Paradigms (3rd ed.). Pearson.

Torres, M. (2024). Desarrollo *web* moderno: De principiante a experto. Ra-Ma Editorial.

Vélez Serrano, J. F., & Peña Abril, A. (2023). Introducción práctica al desarrollo de *software*. Universidad Complutense de Madrid.

1. Control del documento

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Nombre | Cargo | Dependencia | Fecha |
| Autor | OIT | - | OIT | Noviembre de 2024 |

1. Control de cambios (diligenciar únicamente si realiza ajustes a la unidad temática)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Nombre | Cargo | Dependencia | Fecha | Razón del Cambio |
|  |  |  |  |  |  |

\*\*\*\*\*