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.

Diciembre 2024

Tabla de contenido

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.

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.

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

Video 1. Desarrollo de software: programación, arquitectura y herramientas

Enlace de reproducción del video

Síntesis del video: Desarrollo de software: programación, arquitectura y herramientas
En el componente formativo «Desarrollo de software: programación, arquitectura y herramientas» se abordan los fundamentos esenciales para construir aplicaciones robustas y escalables. Comprender y aplicar adecuadamente estos

conceptos es crucial para desarrollar software de calidad, trabajando eficientemente en equipos y siguiendo las mejores prácticas de la industria.

Durante el desarrollo de este componente, se busca una comprensión integral de los pilares fundamentales del desarrollo de software moderno. Se inicia con una introducción a los fundamentos de los lenguajes de programación, explorando sintaxis, estructuras de control y manejo de datos, estableciendo así una base sólida para el desarrollo.

El componente profundiza en la arquitectura de software, abordando patrones de diseño estructurales y principios SOLID, que son fundamentales para construir aplicaciones mantenibles y adaptables. Se exploran las arquitecturas web modernas y APIs RESTful, elementos clave en el desarrollo de aplicaciones distribuidas actuales.

Asimismo, se abordan las herramientas y prácticas de desarrollo colaborativo, con especial énfasis en el control de versiones con Git y la integración continua. Estos conocimientos son esenciales para trabajar eficientemente en equipos de desarrollo. Finalmente, se exploran los entornos de desarrollo modernos, incluyendo técnicas de debugging, testing y consideraciones de seguridad fundamentales para el despliegue de aplicaciones.

¡Bienvenido al mundo del desarrollo de software profesional! Te invitamos a descubrir cómo construir aplicaciones robustas y escalables, utilizando las herramientas y prácticas más relevantes de la industria actual.

1. 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.

1.1. 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 cercana al lenguaje natural.	Python, JavaScript.	Desarrollo web, aplicaciones empresariales.

Fuente. OIT, 2024.

1.2. 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.

1.3. 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.

1.4. 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.

1.5. 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.

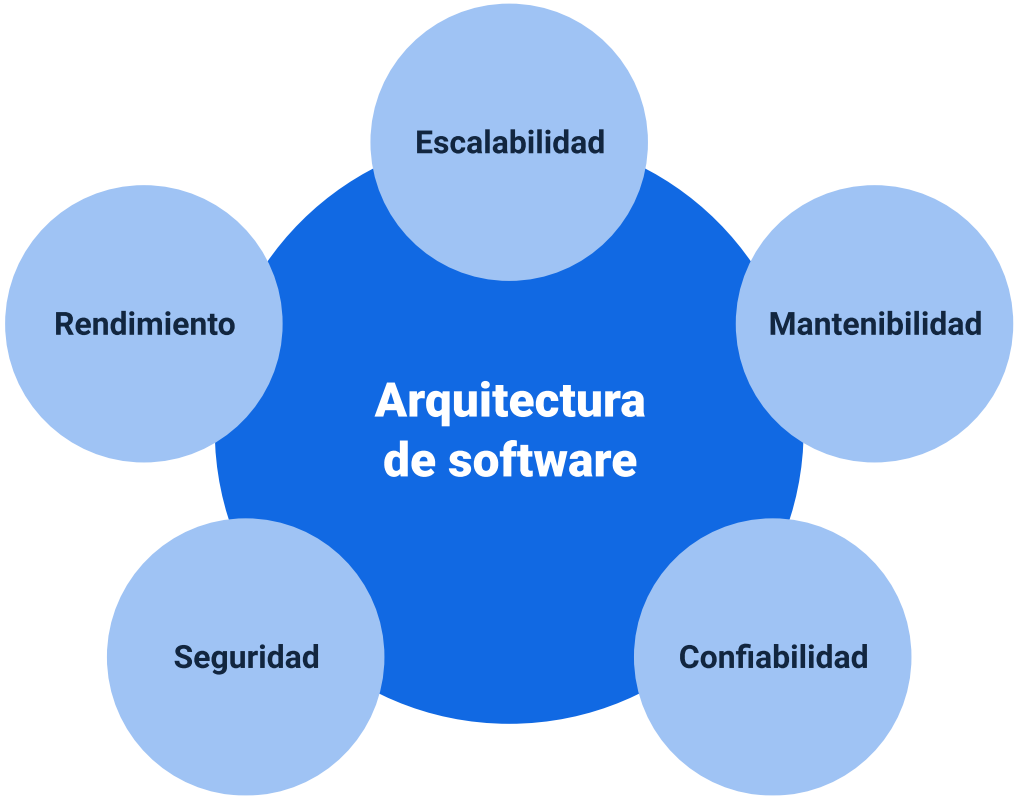
2. Arquitectura y diseño de software

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.

2.1. 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.

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.

2.2. 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)
```

2.3. 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.

2.4. 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.

2.5. 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.

3. 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.

3.1. 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.

Espacio	Descripción	Comandos principales	Uso típico
Remote Repository.	Copia compartida en un servidor.	git push, git pull.	Colaboración.

Fuente. OIT, 2024.

3.2. 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.

3.3. 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
```

3.4. 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).

3.5. 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.

4. 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.

4.1. 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.

4.2. 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)
```

4.3. 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"
```

```
  }
}
```

4.4. 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.

4.5. 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.

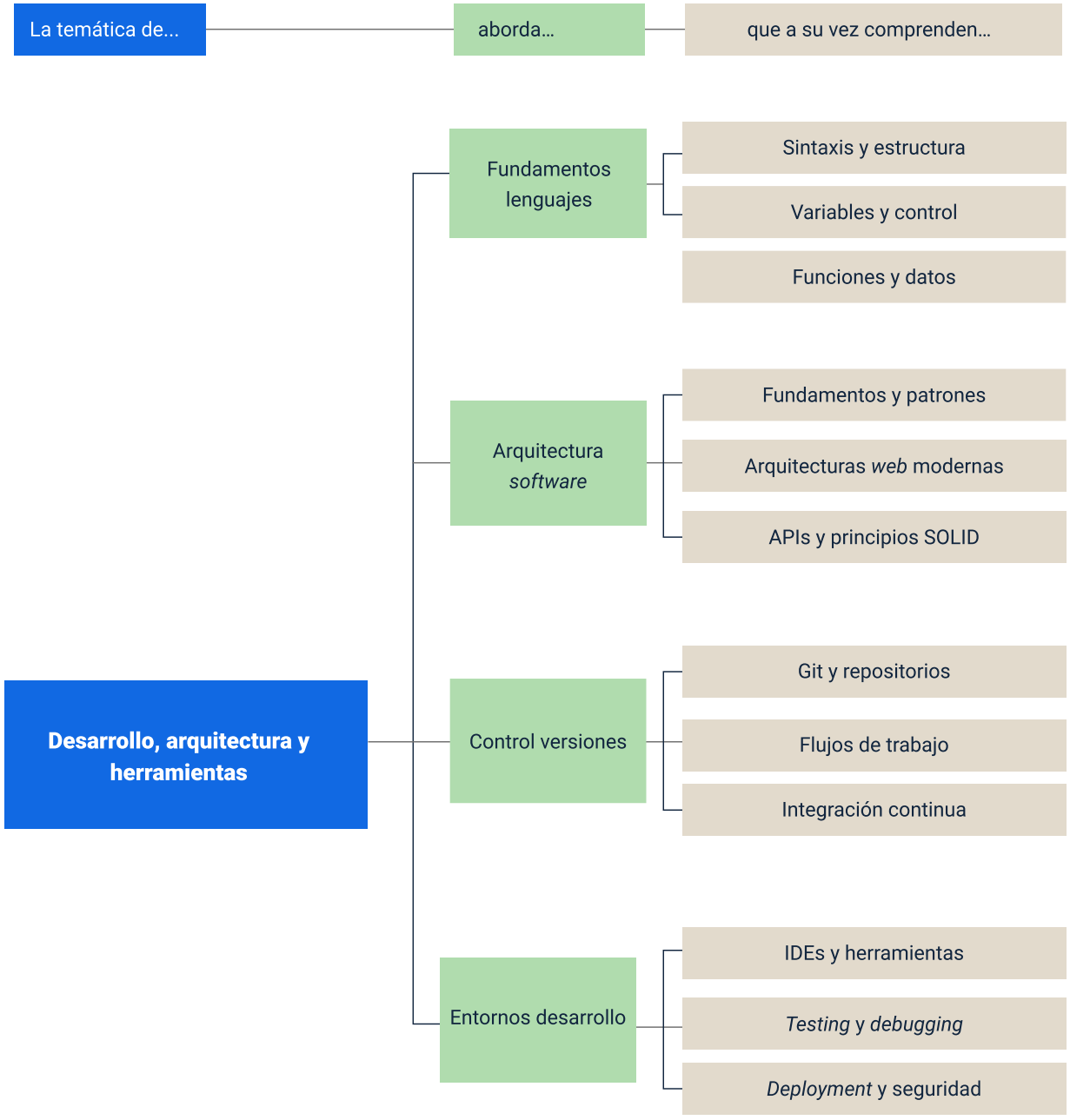
Síntesis

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.



Fuente. OIT, 2024.

Material complementario

Tema	Referencia	Tipo de material	Enlace del recurso
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

Tema	Referencia	Tipo de material	Enlace del recurso
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

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.

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

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.

Créditos

Elaborado por:

Organización Internacional del Trabajo