



# Errores comunes en el trabajo y soluciones prácticas en Java y Spring Boot

**Jhonatan Apaico**
**Systems Engineer**

youtube.com/@JoasDev

# Mala o poca documentación del proyecto

La falta de documentación adecuada o insuficiente en un proyecto de desarrollo puede presentar varios desafíos al ingresar a un nuevo trabajo. Sin una documentación clara y completa, puede resultar difícil comprender la arquitectura, el diseño y la funcionalidad del software existente. La falta de documentación también puede dificultar la identificación y resolución de problemas, ya que no se dispone de información detallada sobre el funcionamiento interno del sistema. Además, la ausencia de documentación puede dificultar la colaboración con otros miembros del equipo y limitar la capacidad de realizar cambios o mejoras de manera eficiente. En general, la mala o poca documentación puede generar una curva de aprendizaje más empinada y retrasar el progreso en el proyecto, por lo que es crucial priorizar y mejorar la documentación en el desarrollo de software.

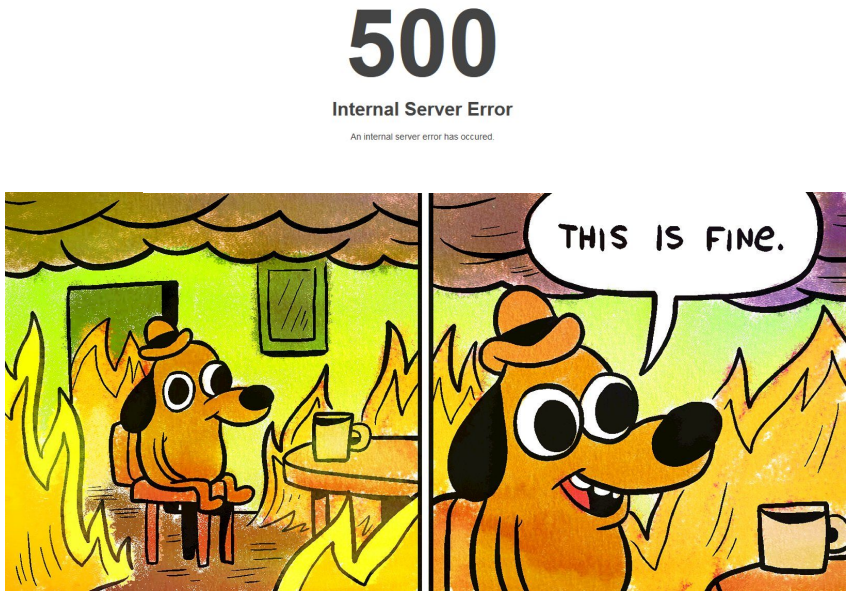

youtube.com/@JoasDev

# Consumo de recursos por no paginar las consultas

Si no se implementa la paginación adecuada en consultas a una base de datos con conjuntos de datos grandes, existe el riesgo de sobrecargar el servidor. Esto puede llevar a problemas como:

**Agotamiento de recursos, tiempos de respuesta prolongados, inestabilidad del servidor**

En resumen, paginar nuestras consultas a la base de datos utilizando Spring Boot nos brinda beneficios significativos en términos de rendimiento, eficiencia y experiencia del usuario. Nos permite administrar conjuntos de datos grandes de manera más efectiva, mejorar la velocidad de respuesta de nuestras aplicaciones y ofrecer una navegación fluida y eficiente a través de los resultados de las consultas.




youtube.com/@JoasDev

# Usar los ambientes de producción para realizar pruebas

Es importante evitar utilizar los ambientes de producción para realizar pruebas debido a los siguientes motivos:

Riesgo de impacto en la disponibilidad y estabilidad del sistema: Las pruebas pueden generar carga adicional, consumir recursos y causar interrupciones en el ambiente de producción. Esto puede afectar la disponibilidad y estabilidad del sistema, lo que podría tener un impacto negativo en los usuarios y en la reputación de la aplicación.

Pérdida de datos y corrupción: Las pruebas pueden implicar la creación, modificación o eliminación de datos. Si se realizan estas acciones en un ambiente de producción, existe el riesgo de pérdida de datos críticos o corrupción de la información. Esto puede resultar en problemas graves y pérdidas irreparables.

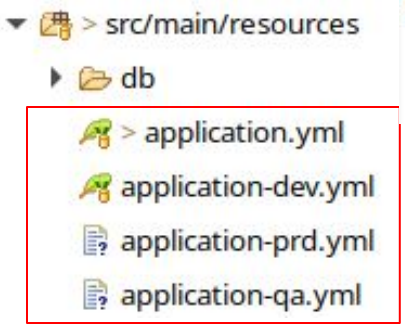
Exposición de información sensible: Los ambientes de producción a menudo contienen datos sensibles y confidenciales. Al realizar pruebas en estos entornos, se corre el riesgo de exponer información sensible a personas no autorizadas, lo que podría tener consecuencias legales y de seguridad significativas.

Dificultad para replicar y aislar problemas: Si se encuentran problemas durante las pruebas en el ambiente de producción, puede resultar complicado aislar y diagnosticar la causa raíz. Los entornos de producción están diseñados para ser estables y funcionales, por lo que la identificación y resolución de problemas puede ser más difícil en comparación con entornos de prueba dedicados.


youtube.com/@JoasDev

# Uso de profile en Spring Boot

En Spring Boot, puedes utilizar varios perfiles en el archivo **application.yml** para configurar diferentes entornos o escenarios de tu aplicación. Cada perfil representa una configuración específica que se activa según el entorno en el que se ejecute la aplicación.




```
spring:
  profiles:
    active: ${PROFILE:dev}
```

Usar varios perfiles te permite tener diferentes configuraciones para diferentes entornos, como desarrollo, pruebas y producción, sin necesidad de modificar el código fuente de la aplicación.


youtube.com/@JoasDev

# Eliminar las tablas y toda la data de la DB

Es cierto que algunos programadores de Spring Boot pueden olvidar establecer la opción de creación automática de tablas en "**none**" al ejecutar el proyecto, lo que puede resultar en la eliminación de las tablas existentes en la base de datos. Esto no se recomienda por varias razones importantes.

En primer lugar, al eliminar las tablas existentes, se pierde toda la información y los datos almacenados en ellas. Esto puede ser extremadamente perjudicial, especialmente en entornos de producción, donde los datos son valiosos y deben conservarse de manera segura. La pérdida de datos puede tener un impacto significativo en la integridad y el funcionamiento de la aplicación y puede resultar en una pérdida de confianza por parte de los usuarios o clientes.

spring.jpa.hibernate.ddl-auto=create-drop 🤣


youtube.com/@JoasDev

# Dependencias innecesarias por copiar de otro proyecto

El archivo POM (Project Object Model) en Maven o Gradle es esencial para gestionar las dependencias de un proyecto. Al copiar el POM de otro proyecto, es común que también se copien todas las dependencias definidas en él. Sin embargo, si no se eliminan las dependencias innecesarias, el proyecto puede terminar con un conjunto de dependencias excesivas que no se utilizan realmente.

```xml
<dependencies>
    <!-- Dependencia principal: Spring Boot Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Dependencia para manejo de tareas y programación asíncrona
    (PERO SI ESTOY HACIENDO UNA API REST QUE HACE ESTO AQUI) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-task</artifactId>
    </dependency>

    <!-- Dependencia para manejo de base de datos con Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- Dependencia para integración con base de datos H2 (opcional, para pruebas) -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```


youtube.com/@JoasDev

# Agregar librerías por un método que tú podrías hacer

En primer lugar, al crear tus propias funciones, tienes un mayor control y comprensión sobre el comportamiento y la lógica de tus métodos. Puedes adaptar las funciones según tus necesidades específicas y asegurarte de que se ajusten exactamente a los requisitos de tu proyecto.

Además, al **evitar la dependencia de bibliotecas externas como por ejemplo org.apache.commons.lang.StringUtils**, reduces la complejidad y la cantidad de código que se necesita en tu proyecto. Esto puede ayudar a mantener tu proyecto más ligero, con menos dependencias y con un menor riesgo de conflictos o problemas relacionados con versiones de bibliotecas.

```
import org.apache.commons.lang.StringUtils;

if (StringUtils.isNotBlank(str)) {
    ...
}

if (StringUtils.isBlank(str)) {
    ...
}
```

youtube.com/@JoasDev

# Usa los métodos utilitarios de Spring Boot

StringUtils proporciona una amplia gama de métodos para realizar operaciones comunes en cadenas, como verificar si una cadena está vacía o nula, comparar cadenas de forma segura, eliminar espacios en blanco, formatear cadenas y mucho más. Estos métodos nos permiten realizar tareas de manipulación y validación de cadenas de manera eficiente y concisa, evitando la necesidad de escribir código personalizado repetitivo.

Por otro lado, CollectionUtils ofrece una variedad de funcionalidades para trabajar con colecciones de manera más efectiva. Podemos encontrar métodos para comprobar si una colección está vacía o nula, buscar elementos específicos, filtrar elementos basados en ciertos criterios, transformar colecciones y mucho más. Estos métodos nos ayudan a simplificar la manipulación y el procesamiento de colecciones, mejorando la legibilidad y mantenibilidad del código.

Además de su funcionalidad, los utilitarios de Spring Boot están bien probados y ampliamente utilizados en la comunidad de desarrollo. Esto significa que podemos confiar en su eficacia y robustez, lo que reduce la necesidad de implementar nuestras propias soluciones y nos permite aprovechar la experiencia y las mejores prácticas compartidas por la comunidad.

```java
import org.springframework.util.StringUtils;

if(StringUtils.hasText(q)) {
    ...
}
```

**Package** org.springframework.util

## Class CollectionUtils

java.lang.Object
    org.springframework.util.CollectionUtils

```
public abstract class CollectionUtils
extends Object
```

Miscellaneous collection utility methods. Mainly for internal use within the framework.


youtube.com/@JoasDev

# Demasiados atributos con sus respectivos getter y setters en el DTO

Cuando un DTO (Objeto de Transferencia de Datos) tiene demasiados atributos con sus correspondientes métodos getter y setter, puede generar un código redundante y dificultar la legibilidad y mantenibilidad del código. Un DTO es una clase utilizada para transferir datos entre capas de una aplicación, y generalmente se crea para encapsular los datos necesarios en una respuesta o solicitud. Sin embargo, cuando un DTO contiene una gran cantidad de atributos, la cantidad de código generado puede volverse abrumadora.

Tener muchos atributos con sus respectivos getter y setter puede dificultar la comprensión del código, ya que se vuelve difícil identificar y seguir la lógica principal del DTO. Además, la necesidad de escribir y mantener una gran cantidad de métodos puede aumentar la probabilidad de errores y afectar el rendimiento de la aplicación, especialmente si no todos los atributos se utilizan en todas las situaciones.

```java
public class ExampleClass {
    private String attribute1;
    private int attribute2;
    private boolean attribute3;
    private double attribute4;
    private List<String> attribute5;

    public String getAttribute1() {
        return attribute1;
    }

    public void setAttribute1(String attribute1) {
        this.attribute1 = attribute1;
    }

    public int getAttribute2() {
        return attribute2;
    }

    public void setAttribute2(int attribute2) {
        this.attribute2 = attribute2;
    }

    public boolean isAttribute3() {
        return attribute3;
    }

    public void setAttribute3(boolean attribute3) {
        this.attribute3 = attribute3;
    }

    public double getAttribute4() {
        return attribute4;
    }

    public void setAttribute4(double attribute4) {
        this.attribute4 = attribute4;
    }

    public List<String> getAttribute5() {
        return attribute5;
    }

    public void setAttribute5(List<String> attribute5) {
        this.attribute5 = attribute5;
    }
}
```

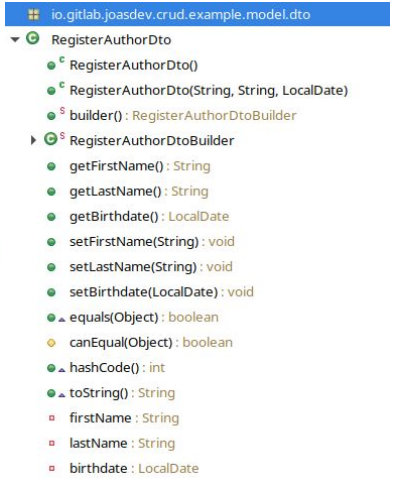
youtube.com/@JoasDev

# Lombok

Lombok es un proyecto que nació en el año 2009 que, mediante contribuciones, ha ido ganando en riqueza y variedad de recursos. Es una librería para Java que a través de anotaciones reduce el código que codificamos, es decir, nos ahorra tiempo y mejora la legibilidad del mismo. **Las transformaciones de código que realiza se hacen en tiempo de compilación.**

**Fuente:** [paradigmadigital.com](paradigmadigital.com)

```java
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Data
public class RegisterAuthorDto {

    private String firstName;
    private String lastName;

    @JsonFormat(pattern="dd/MM/yyyy")
    private LocalDate birthdate;
}
```



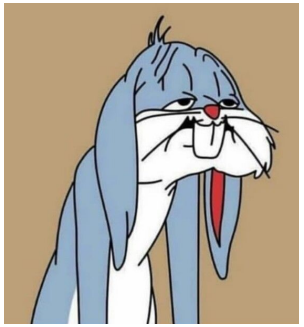


youtube.com/@JoasDev

# Convertir clases (DTO -> Entity)

```java
public class UserDTO {
    private String name;
    private String email;

    // constructor, getters, setters, etc.
}

@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // constructor, getters, setters, etc.

    public static User fromDTO(UserDTO userDTO) {
        User user = new User();
        user.setName(userDTO.getName());
        user.setEmail(userDTO.getEmail());
        return user;
    }
}
```

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // otros métodos del repositorio
}

@Service
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public void saveUser(UserDTO userDTO) {
        User user = User.fromDTO(userDTO);
        userRepository.save(user);
    }
}
```



youtube.com/@JoasDev

# Model Mapper

ModelMapper es una popular biblioteca de mapeo de objetos en Java que facilita la conversión de objetos de un tipo a otro. Proporciona una forma sencilla y conveniente de mapear los campos de un objeto de origen a un objeto de destino, incluso cuando los nombres de los campos o las estructuras de los objetos no coinciden.

Fuente: https://modelmapper.org/getting-started/

```
//Source model
// Assume getters and setters on each class
class Order {
  Customer customer;
  Address billingAddress;
}

class Customer {
  Name name;
}

class Name {
  String firstName;
  String lastName;
}

class Address {
  String street;
  String city;
}
```

```
// Destination Model
// Assume getters and setters
class OrderDTO {
  String customerFirstName;
  String customerLastName;
  String billingStreet;
  String billingCity;
}
```

```
ModelMapper modelMapper = new ModelMapper();
OrderDTO orderDTO = modelMapper.map(order, OrderDTO.class);

assertEquals(order.getCustomer().getName().getFirstName(), orderDTO.getCustomerFirstName());
assertEquals(order.getCustomer().getName().getLastName(), orderDTO.getCustomerLastName());
assertEquals(order.getBillingAddress().getStreet(), orderDTO.getBillingStreet());
assertEquals(order.getBillingAddress().getCity(), orderDTO.getBillingCity());
```

youtube.com/@JoasDev

# Uso de Map en las respuestas REST


El uso de un Map como respuesta puede llevar a una falta de estructura y claridad en la API. Al retornar un Map, se pierde la información sobre la estructura de los datos devueltos, lo que dificulta que los consumidores de la API comprendan y utilicen adecuadamente los datos recibidos. **En cambio, es preferible utilizar objetos de dominio específicos o clases DTO (Data Transfer Objects) que representen la estructura de la respuesta.**

Además, retornar un Map puede dificultar el mantenimiento y evolución de la API en el tiempo. Si se retorna un Map, cualquier cambio en la estructura de la respuesta requerirá modificar el código del cliente para adaptarse a esos cambios. En cambio, si se utiliza una clase DTO o un objeto de dominio, los cambios pueden manejarse de manera más controlada y compatible mediante la versión de la API y sin afectar a los clientes existentes.

```java
@RestController
@RequestMapping("/products")
public class ProductController {

    private final Map<Long, Product> productMap = new HashMap<>();

    @GetMapping("/{id}")
    public ResponseEntity<Map<String, Object>> getProductById(@PathVariable Long id) {
        Product product = productMap.get(id);
        if (product != null) {
            Map<String, Object> response = new HashMap<>();
            response.put("id", product.getId());
            response.put("name", product.getName());
            response.put("price", product.getPrice());
            return ResponseEntity.ok(response);
        } else {
            return ResponseEntity.notFound().build();
        }
    }
}
```



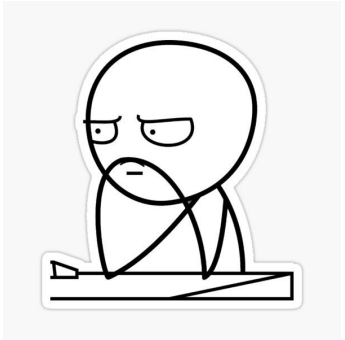
youtube.com/@JoasDev

# Uso de Genéricos

En su esencia, el término genéricos significa tipos parametrizados. Los tipos parametrizados son importantes porque le permiten crear clases, interfaces y métodos en los que el tipo de datos sobre los que operan se especifica como parámetro. Una clase, interfaz o método que funciona con un tipo de parámetro se denomina genérico, como una clase genérica o método genérico.

Fuente: [javadesdecero.es]

```
public interface Pair<K, V> {
  public K getKey();
  public V getValue();
}
public class OrderedPair<K, V> implements Pair<K, V> {

  private K key;
  private V value;

  public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
  }

  public K getKey() { return key; }
  public V getValue() { return value; }
}
```



youtube.com/@JoasDev

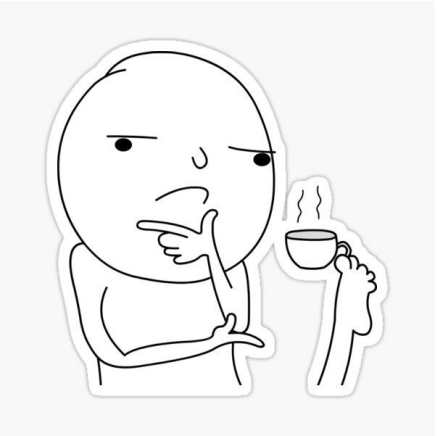
# Uso de Genéricos

Según las convenciones los nombres de los parámetros de tipo usados comúnmente son los siguientes:

- E: elemento de una colección.
- K: clave.
- N: número.
- T: tipo.
- V: valor.
- S, U, V etc: para segundos, terceros y cuartos tipos.

Fuente: [picodotdev.github.io]

```java
// TypeErasure to Object
public class Node<T> {

  private T data;
  private Node<T> next;

  public Node(T data, Node<T> next) }
    this.data = data;
    this.next = next;
  }

  public T getData() { return data; }
  // ...
}

// Node type erased
public class Node {

  private Object data;
  private Node next;

  public Node(Object data, Node next) {
    this.data = data;
    this.next = next;
  }

  public Object getData() { return data; }
  // ...
}
```



youtube.com/@JoasDev

# Evita poner un modelo para recuperar las columnas de tu consulta

Cuándo quieres especificar sólo las columnas que necesitas, puedes enviar un DTO en el Select, pero no se recomienda este método ya que puede traer problemas al darle mantenimiento.

```java
public class ProductDTO {
    private String name;
    private BigDecimal price;

    public ProductDTO(String name, BigDecimal price) {
        this.name = name;
        this.price = price;
    }

    // Getters y setters
}
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {

    @Query("SELECT new com.example.dto.ProductDTO(p.name, p.price) FROM Product p")
    List<ProductDTO> findProductNamesAndPrices();
}
```


youtube.com/@JoasDev

# Projection

Cuando se usa Spring Data JPA para implementar la capa de persistencia, el repositorio generalmente devuelve una o más instancias de la clase raíz. Sin embargo, la mayoría de las veces, no necesitamos todas las propiedades de los objetos devueltos.

En tales casos, es posible que queramos recuperar datos como objetos de tipos personalizados. Estos tipos reflejan vistas parciales de la clase raíz, que contienen solo las propiedades que nos interesan. Aquí es donde las proyecciones son útiles.

Fuente:
https://www.baeldung.com/spring-data-jpa-projections

```java
public interface AuthorProjection {
    Integer getAuthorId();
    String getFirstName();
    String getLastName();
    String getFullName();

    @JsonFormat(pattern="dd-MM-yyyy")
    LocalDate getBirthdate();
}
```

```java
@Query("SELECT a.authorId as authorId, a.firstName as firstName, a.lastName as lastName, "
        + "a.firstName||' '||a.lastName as fullName, a.birthdate as birthdate "
        + "FROM Author a WHERE a.authorId = :authorId")
AuthorProjection findByUserId(@Param("authorId") Integer authorId);
```

youtube.com/@JoasDev

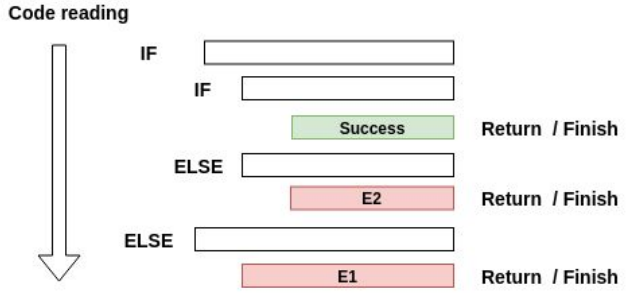
# Los regex en el Controlador

Muchos desarrolladores de Spring Boot pueden no estar al tanto de una funcionalidad interesante al utilizar path variables en las rutas de sus aplicaciones. Al definir una variable de ruta utilizando {nombreVariable}, se puede aprovechar la capacidad de establecer un parámetro con una expresión regular (regex). Esto permite una mayor flexibilidad en la validación de los valores pasados en la URL. Al especificar una expresión regular en el patrón de la variable de ruta, se puede restringir el formato o rango de valores aceptados. Por ejemplo, si se desea que una variable de ruta solo acepte números enteros, se puede utilizar {id:\d+} en lugar de simplemente {id}. Esta característica puede ser muy útil para garantizar que los parámetros de ruta cumplan con ciertas condiciones y mejorar la robustez de las aplicaciones Spring Boot.

```java
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{userId:[A-Za-z0-9]{8}}")
    public String getUserById(@PathVariable("userId") String userId) {
        // Lógica para buscar el usuario por el identificador
        return "Se encontró el usuario con ID: " + userId;
    }
}
```


youtube.com/@JoasDev

# ¿Cómo corregir el uso excesivo de else?





youtube.com/@JoasDev


```java
int x = 10;
int y = 5;

if (x > y) {
    System.out.println("x es mayor que y");

    if (x > 10) {
        System.out.println("x también es mayor que 10");
    } else {
        System.out.println("x es mayor que y, pero no es mayor que 10");
    }
} else {
    System.out.println("x no es mayor que y");
}
```

---

```java
int x = 10;
int y = 5;

if (x <= y) {
    System.out.println("x no es mayor que y");
    return;
}


System.out.println("x es mayor que y");

if(x <= 10){
    System.out.println("x es mayor que y, pero no es mayor que 10");
    return;
}

System.out.println("x también es mayor que 10");
```

# Por último te dejo la tarea de investigar

**Jhonatan Apaico**
**Systems Engineer**

youtube.com/@JoasDev

# SOLID


Son un conjunto de reglas y mejores prácticas a seguir al diseñar una estructura de clase.

Estos cinco principios nos ayudan a comprender la necesidad de ciertos patrones de diseño y arquitectura de software en general. Así que creo que es un tema que todo desarrollador debería aprender.

https://www.freecodecamp.org/espanol/news/los-principios-solid-explicados-en-espanol/

**Principio de Responsabilidad Única (Single Responsibility Principle - SRP):** Un objeto o clase debe tener una única responsabilidad. En otras palabras, cada objeto debe tener una razón para cambiar y debe ser responsable de una única funcionalidad.

**Principio de Abierto/Cerrado (Open/Closed Principle - OCP):** Las entidades de software (clases, módulos, etc.) deben estar abiertas para su extensión pero cerradas para su modificación. Esto significa que se deben poder agregar nuevas funcionalidades sin modificar el código existente.

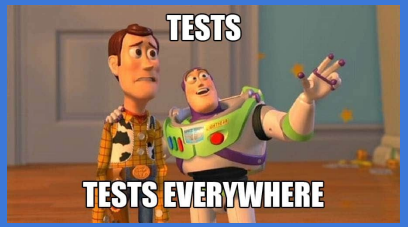
**Principio de Sustitución de Liskov (Liskov Substitution Principle - LSP):** Las instancias de una clase derivada deben poder sustituir a las instancias de la clase base sin alterar el comportamiento del programa. En otras palabras, una clase hija debe ser utilizada de manera transparente como si fuera la clase padre.

**Principio de Segregación de Interfaces (Interface Segregation Principle - ISP):** Los clientes no deben verse obligados a depender de interfaces que no utilizan. En lugar de tener interfaces monolíticas, es preferible tener interfaces más específicas y cohesivas.

**Principio de Inversión de Dependencia (Dependency Inversion Principle - DIP):** Los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones. Este principio promueve el uso de interfaces o clases abstractas para desacoplar las dependencias y facilitar la sustitución de implementaciones.

youtube.com/@JoasDev

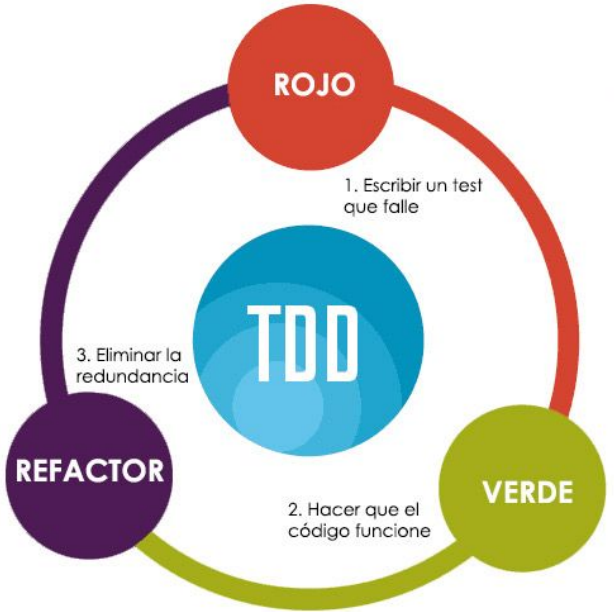
# Test Driver Developer (TDD)




Test Driver Developer (TDD) se refiere a un enfoque de desarrollo de software en el cual se escriben primero las pruebas automatizadas antes de escribir el código de producción. Es una práctica popular en el desarrollo ágil y se basa en el ciclo "red-green-refactor".

En TDD, el desarrollador comienza escribiendo una prueba automatizada que describa la funcionalidad que desea implementar. Esta prueba inicialmente fallará, ya que aún no se ha implementado la funcionalidad. Luego, el desarrollador escribe el código mínimo necesario para que la prueba pase (es decir, que la prueba sea exitosa). Una vez que la prueba pasa satisfactoriamente, el código se refina y se refactoriza según sea necesario para mejorar su diseño y calidad.

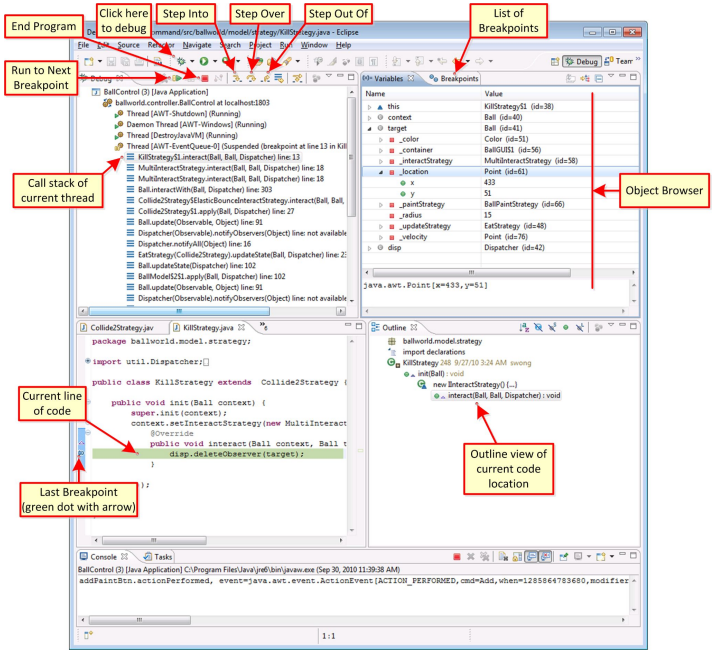
El objetivo principal del TDD es asegurar que el código se mantenga confiable y libre de errores al tiempo que se mejora su diseño. Al escribir las pruebas primero, los desarrolladores pueden tener una mayor confianza en la calidad del código y reducir la probabilidad de introducir errores. Además, el TDD fomenta un enfoque incremental y modular en el desarrollo, lo que facilita la adaptación a cambios y mejoras futuras.




youtube.com/@JoasDev

# Debug

Saber cómo depurar o debuguear el código Java es fundamental para cualquier programador, ya que permite identificar y solucionar los errores o bugs presentes en el programa. La depuración efectiva no solo ahorra tiempo, sino que también mejora la calidad del software y la experiencia del usuario. Al conocer las técnicas de depuración en Java, puedes analizar el flujo de ejecución, identificar problemas de lógica, encontrar errores sintácticos, seguir el valor de las variables y realizar un seguimiento detallado del código en tiempo real. Esto te ayuda a localizar y solucionar rápidamente los errores, garantizando un programa más eficiente y confiable. La capacidad de depurar te convierte en un programador más competente y te permite desarrollar soluciones robustas y libres de errores, lo cual es esencial para el éxito en el desarrollo de software.




youtube.com/@JoasDev

# Aprende GNU Linux

GNU/Linux proporciona una visión más profunda del funcionamiento interno de un sistema operativo. Aprender GNU/Linux implica comprender conceptos fundamentales como el sistema de archivos, la administración de procesos, la gestión de memoria y la administración de red. Estos conocimientos pueden ser transferibles a otros sistemas operativos y mejorar tu comprensión general de la informática.

En resumen, aprender GNU/Linux como desarrollador te permite adaptarte a un entorno de desarrollo ampliamente utilizado, personalizar tu entorno de desarrollo, acceder a herramientas y recursos de calidad, mejorar tu comprensión de los conceptos fundamentales de los sistemas operativos y administrar servidores de manera efectiva. Estas habilidades pueden beneficiar tu carrera profesional y mejorar tus capacidades como desarrollador de software.



youtube.com/@JoasDev

# Actualizarse

En el campo de la tecnología de la información (T.I.), es fundamental que las personas que trabajan en esta área se mantengan constantemente actualizadas. Esto se debe a que el mundo de la tecnología avanza a un ritmo acelerado y las nuevas innovaciones, herramientas y tendencias surgen de manera continua.

La actualización constante es crucial para mantenerse al día con los avances tecnológicos, ya que brinda la oportunidad de adquirir nuevas habilidades y conocimientos que son necesarios para enfrentar los desafíos y las demandas cambiantes de la industria.


youtube.com/@JoasDev