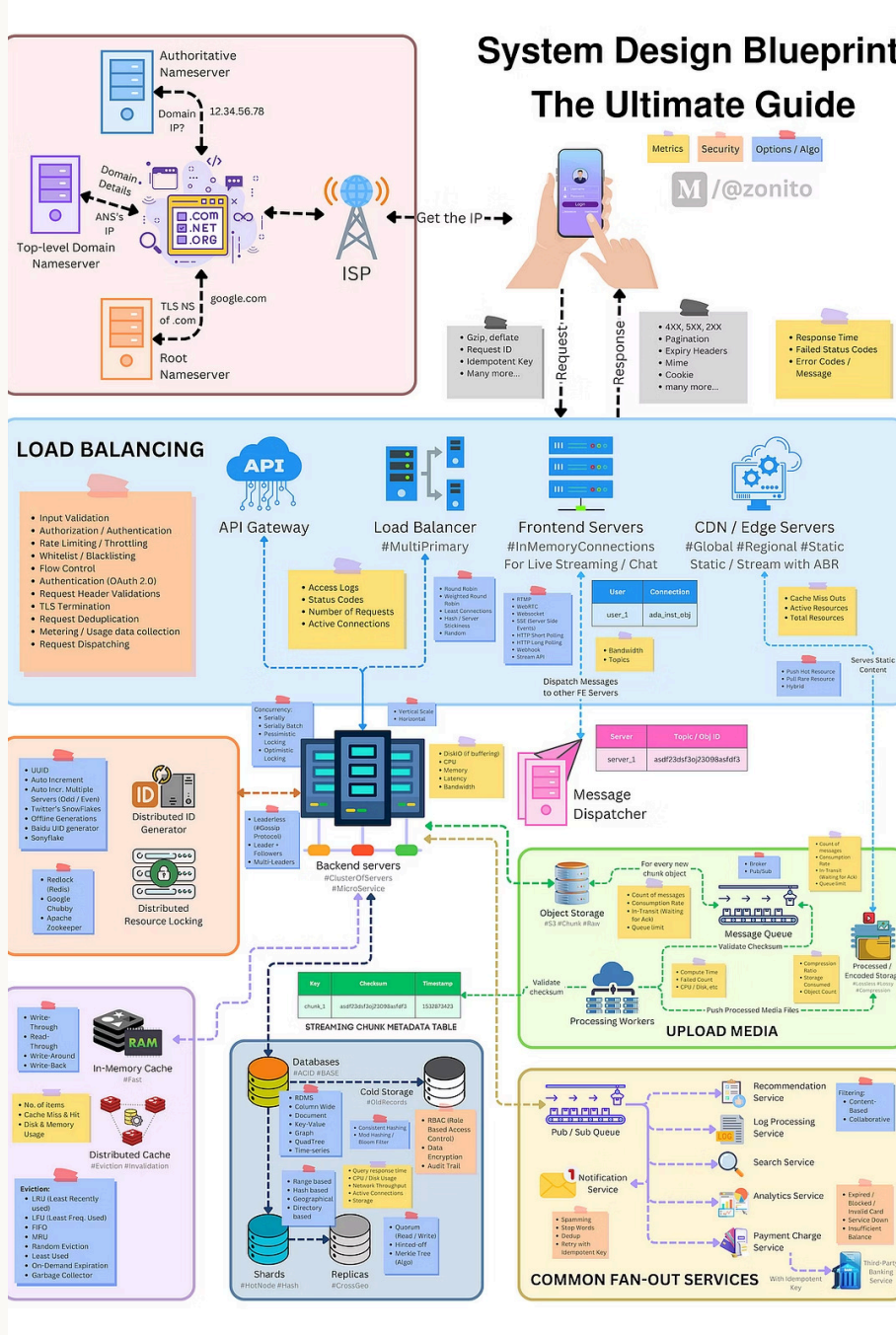
Patrones Creacionales en Java con Lombok

¡Bienvenidos al curso donde dominaréis los patrones de diseño creacionales! A lo largo de estas sesiones, exploraremos cómo construir objetos de manera eficiente y flexible en Java, potenciando nuestro código con Lombok y convirtiéndonos en verdaderos maestros del diseño de software.



¡Vuestra Misión: Biblioteca Digital!

Vamos a construir una mini biblioteca online con funcionalidades avanzadas:

1

Gestión de Libros y Usuarios

Control total sobre quién lee qué, con seguimiento detallado de préstamos

2

Creación de Objetos Nivel Experto

Implementación de objetos complejos con precisión quirúrgica

3

Clonación Avanzada

Duplicación de objetos manteniendo o modificando propiedades según necesidades

4

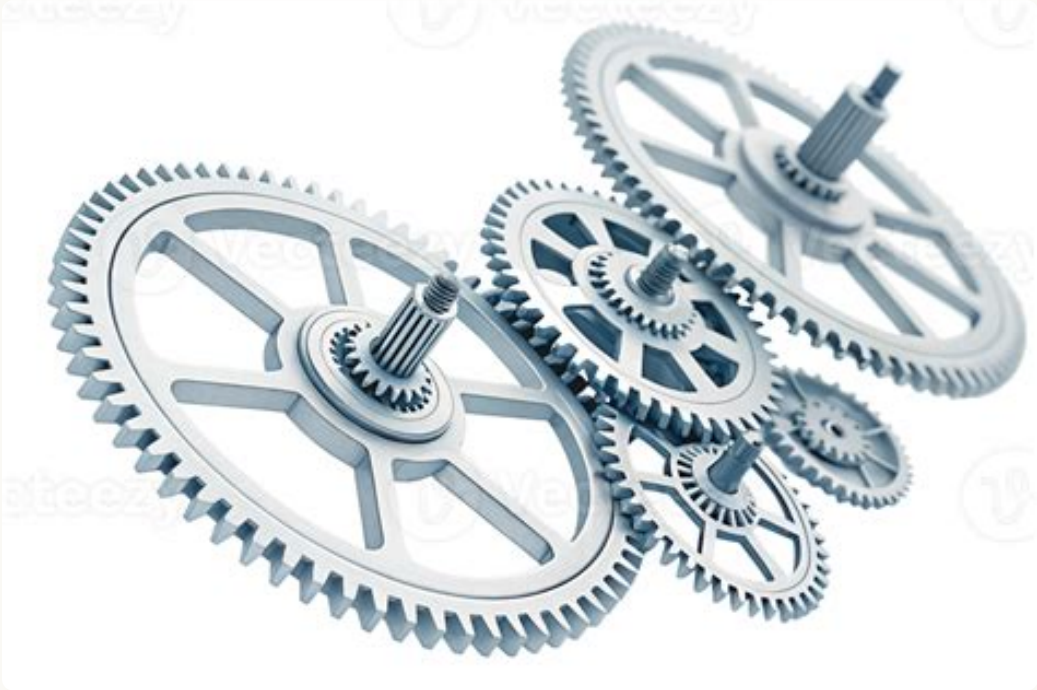
Interfaces Personalizadas

Diferentes interfaces adaptadas al tipo de usuario

¿Por qué los Patrones Creacionales?

Los patrones creacionales nos proporcionan soluciones probadas para:

- Abstraer el proceso de instanciación
- Ocultar la lógica de creación de objetos
- Hacer el código más mantenible y flexible
- Favorecer el principio de responsabilidad única
- Facilitar pruebas unitarias



Los patrones son herramientas, no leyes. Aprender cuándo aplicarlos es tan importante como conocer su implementación.

El Poder de Lombok en Nuestro Proyecto

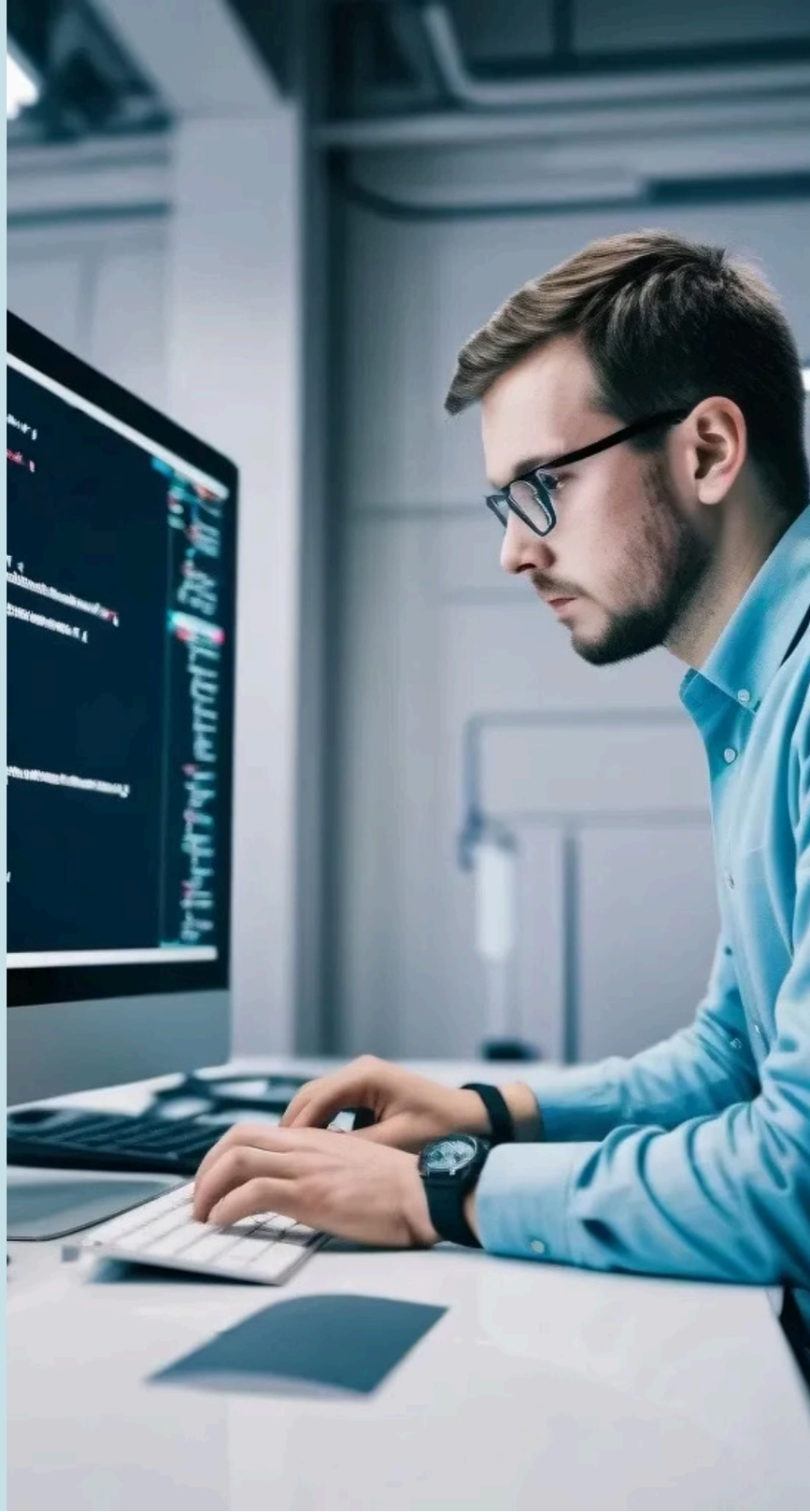
Lombok nos permitirá reducir drásticamente el código repetitivo y centrar nuestra atención en lo importante: la implementación de los patrones.

Sin Lombok

```
public class Libro {
    private String titulo;
    private String autor;

    public Libro(String titulo, String
autor) {
        this.titulo = titulo;
        this.autor = autor;
    }

    public String getTitulo() { return
titulo; }
    public String getAutor() { return
autor; }
    public void setTitulo(String titulo)
{ this.titulo = titulo; }
    public void setAutor(String autor)
{ this.autor = autor; }
}
```

Con Lombok

```
@Getter @Setter
@AllArgsConstructor
public class Libro {
 private String titulo;
 private String autor;
}
```



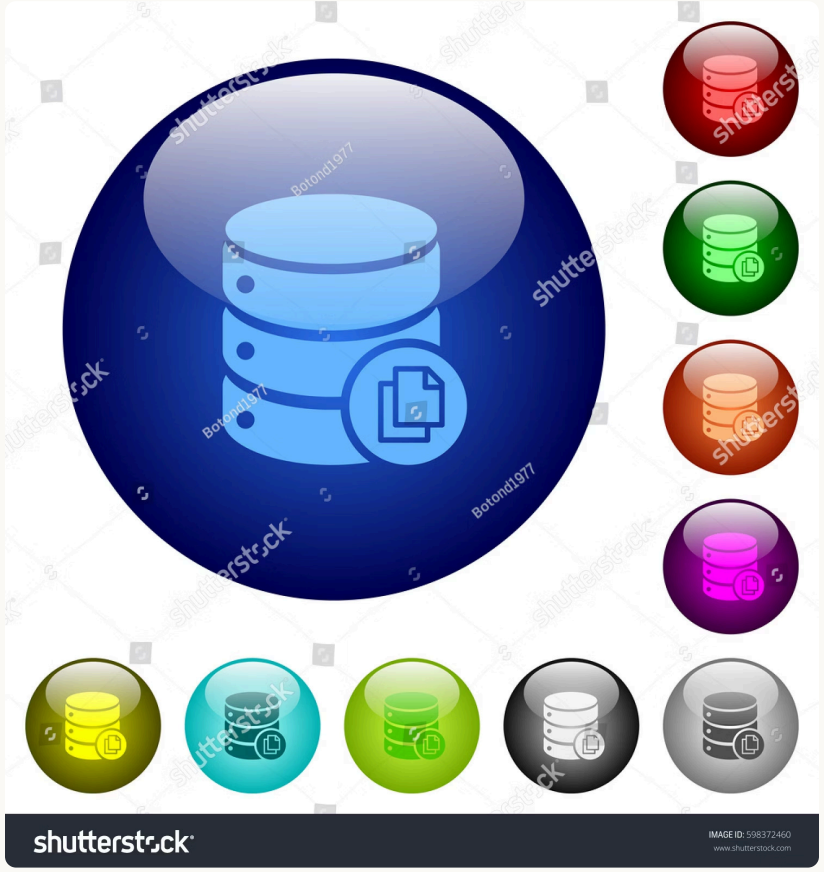
Patrón Singleton: El Guardián de la Unicidad

El patrón Singleton garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a ella.

Características clave:

- Constructor privado que previene instanciación externa
- Método estático que devuelve la instancia única
- La instancia se almacena en un campo estático

Perfecto para gestionar recursos compartidos como una base de datos, pools de conexiones o caches.



Implementando Singleton en Nuestra Biblioteca

Implementación Básica

```
public class Database {
    private static Database instance;

    private Database() {
        // Constructor privado
    }

    public static Database getInstance() {
        if (instance == null) {
            instance = new Database();
        }
        return instance;
    }
}
```

Mejoras Recomendadas

Thread-Safe: Usa synchronized o inicialización estática

Lazy Initialization: Crea la instancia solo cuando sea necesario

Enum Singleton: La forma más concisa y a prueba de errores

Versiones Avanzadas de Singleton

Thread-Safe con Synchronized

```
public static synchronized Database
getInstance() {
    if (instance == null) {
        instance = new Database();
    }
    return instance;
}
```

Inicialización Estática

```
public class Database {
    private static final Database
INSTANCE =
        new Database();

    private Database() {}

    public static Database getInstance()
{
        return INSTANCE;
    }
}
```

Enum Singleton

```
public enum Database {
    INSTANCE;

    private List libros = new ArrayList<>
();

    public void addLibro(Libro libro) {
        libros.add(libro);
    }
}
```

Para nuestro proyecto, la versión con enum es la más recomendada por su concisión y seguridad.

Patrón Factory Method: El Creador Flexible

Factory Method define una interfaz para crear objetos, pero permite a las subclases decidir qué clase instanciar. Este patrón transfiere la responsabilidad de instanciación a las subclases.

¿Cuándo usarlo?

- Cuando una clase no puede anticipar qué tipo de objetos debe crear
- Cuando queremos que las subclases especifiquen los objetos a crear
- Cuando hay "familias" de objetos relacionados

Ventajas

- Reduce el acoplamiento
- Principio de responsabilidad única
- Principio de abierto/cerrado
- Código más limpio y organizado

Implementando Factory Method para Nuestra Biblioteca

Interfaz Libro

```
public interface Libro {
    void mostrarInfo();
    String getTitulo();
}
```

Implementaciones Concretas

```
@Getter @AllArgsConstructor
public class LibroFisico implements Libro {
    private String titulo;
    private String autor;
    private int paginas;

    @Override
    public void mostrarInfo() {
        System.out.println("Libro físico: " + titulo);
    }
}
```

Factory Method

```
public abstract class LogisticaLibro {
 public abstract Libro crearLibro(String titulo, String autor);

 public void entregarLibro(String titulo, String autor) {
 Libro libro = crearLibro(titulo, autor);
 libro.mostrarInfo();
 }
}
```

Creando Fábricas Concretas

```
public class LogisticaLibroFisico extends LogisticaLibro {
    @Override
    public Libro crearLibro(String titulo, String autor) {
        return new LibroFisico(titulo, autor, 300);
    }
}

public class LogisticaLibroDigital extends LogisticaLibro {
    @Override
    public Libro crearLibro(String titulo, String autor) {
        return new LibroDigital(titulo, autor, "PDF");
    }
}
```

Uso en la Aplicación

```
// Código del cliente
LogisticaLibro logistica;

// Configuramos la fábrica según necesidad
logistica = new LogisticaLibroFisico();
logistica.entregarLibro("Don Quijote", "Cervantes");

// Cambiar a otro tipo
logistica = new LogisticaLibroDigital();
logistica.entregarLibro("La Celestina", "F. de Rojas");
```

Este enfoque nos permite cambiar el tipo de libro creado modificando solo la fábrica que utilizamos, sin alterar el código del cliente.

Patrón Abstract Factory: Familias de Objetos

Abstract Factory proporciona una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas.

Diferencia con Factory Method

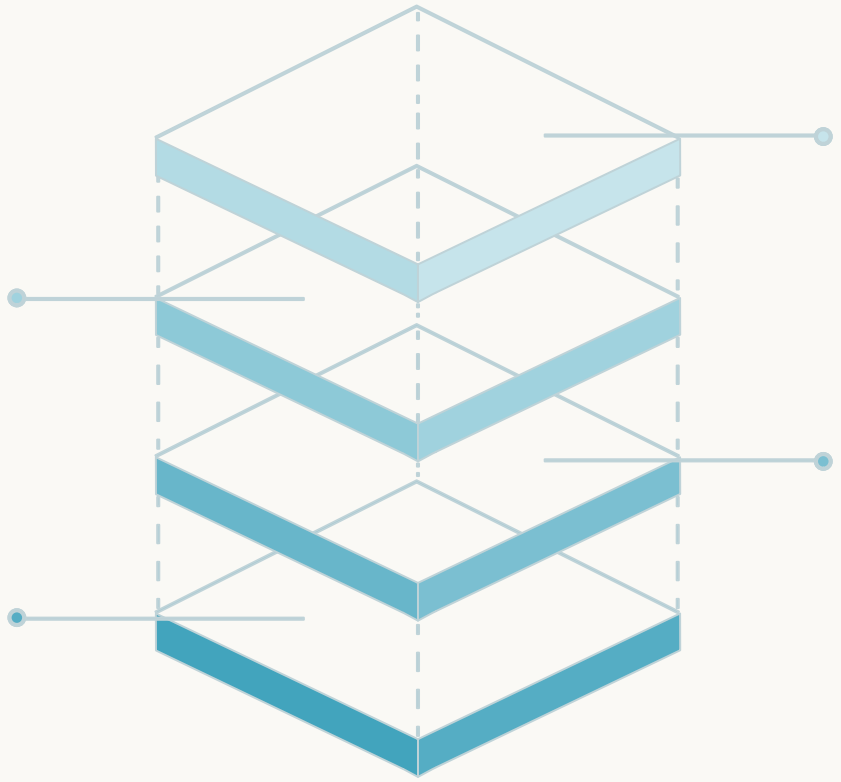
Mientras Factory Method se enfoca en crear un tipo de producto, Abstract Factory crea familias de productos relacionados.

Ventajas Clave

Aislamiento de clases concretas, facilidad para intercambiar familias de productos, y garantía de compatibilidad entre productos.

En nuestra biblioteca, lo usaremos para crear interfaces y métodos de envío adaptados a diferentes tipos de usuarios.

Arquitectura de Abstract Factory para Nuestra Biblioteca



Productos Concretos

AdminUI, UsuarioUI, EnvioExpress, EnvioNormal

UsuarioFactory

Fábrica para componentes de usuario

AdminFactory

Fábrica para componentes administrador

Abstract Factory

Define interfaces para UI y envío

El patrón Abstract Factory nos permitirá crear interfaces de usuario y métodos de envío coherentes según el tipo de usuario (administrador o usuario normal).

Implementando Abstract Factory

Productos y sus Interfaces

```
public interface InterfazUI {
    void mostrar();
}

@AllArgsConstructor
public class AdminUI implements InterfazUI {
    private String tema;

    @Override
    public void mostrar() {
        System.out.println("UI Administrador con tema " + tema);
    }
}

public interface MetodoEnvio {
    void enviar(String item);
}
```

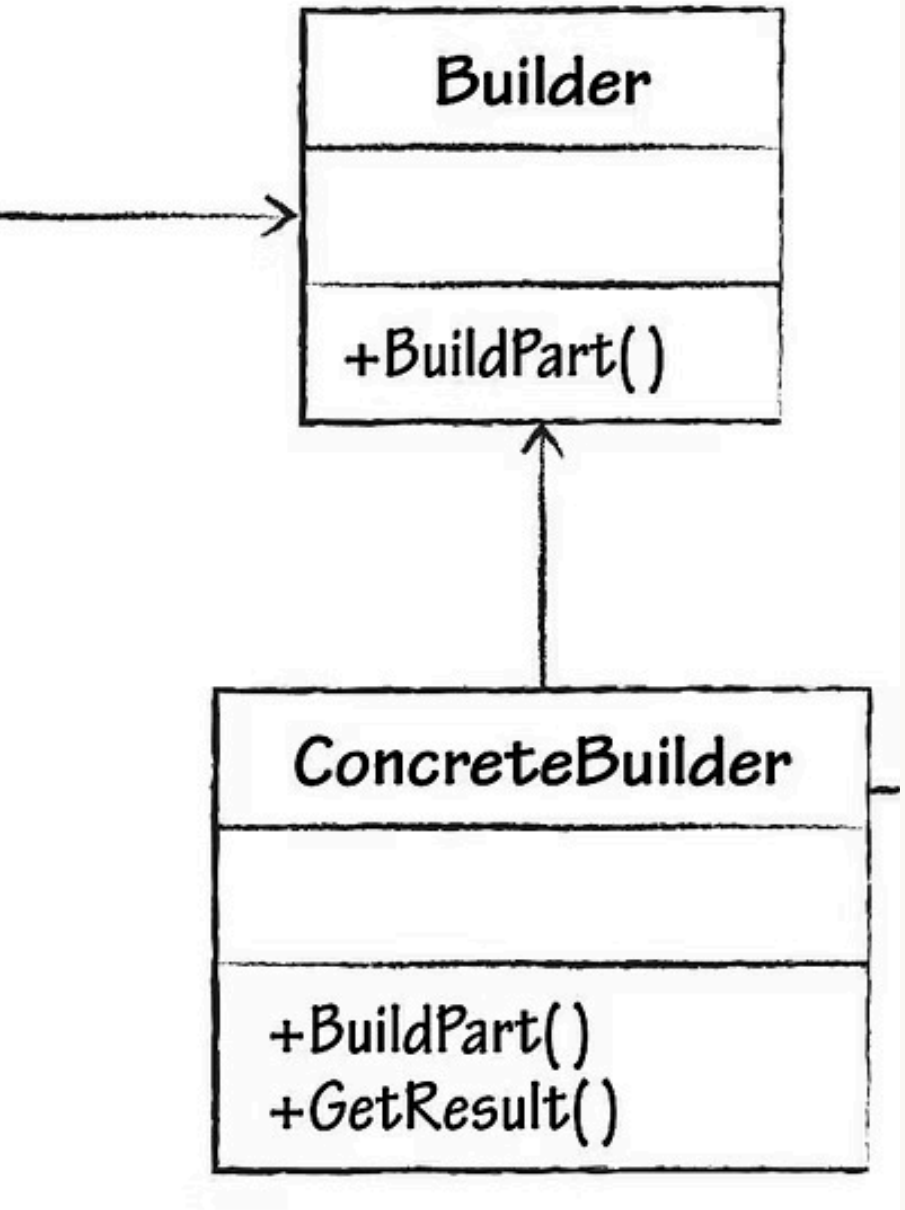
Fábrica Abstracta

```
public interface BibliotecaFactory {
 InterfazUI crearUI();
 MetodoEnvio crearMetodoEnvio();
}

public class AdminFactory implements BibliotecaFactory {
 @Override
 public InterfazUI crearUI() {
 return new AdminUI("Oscuro");
 }

 @Override
 public MetodoEnvio crearMetodoEnvio() {
 return new EnvioExpress();
 }
}
```

Utilizando Abstract Factory en el Cliente

```
public class Cliente {
    private InterfazUI interfaz;
    private MetodoEnvio envio;

    public Cliente(BibliotecaFactory factory) {
        interfaz = factory.crearUI();
        envio = factory.crearMetodoEnvio();
    }

    public void usarSistema() {
        interfaz.mostrar();
        envio.enviar("Libro: 1984");
    }

    public static void main(String[] args) {
        // Cliente administrador
        Cliente admin = new Cliente(new AdminFactory());
        admin.usarSistema();

        // Cliente usuario normal
        Cliente usuario = new Cliente(new UsuarioFactory());
        usuario.usarSistema();
    }
}
```

Con Abstract Factory, cambiamos toda la "familia" de objetos simplemente cambiando la fábrica, manteniendo la coherencia entre interfaz y funcionalidades.



Patrón Builder: Constructor de Complejidad

El patrón Builder separa la construcción de un objeto complejo de su representación, permitiendo crear diferentes configuraciones con el mismo proceso.

Ventajas clave:

- Construcción paso a paso
- Reutilización del código de construcción
- Inmutabilidad posible
- Legibilidad del código mejorada
- Prevención de objetos inconsistentes

Implementando Builder con Lombok

Clase Usuario con Builder Manual

```
public class Usuario {
    private final String nombre;
    private final String email;
    private final int edad;
    private final String direccion;
    private final String telefono;

    private Usuario(Builder builder) {
        this.nombre = builder.nombre;
        this.email = builder.email;
        this.edad = builder.edad;
        this.direccion = builder.direccion;
        this.telefono = builder.telefono;
    }

    public static class Builder {
        // Campos obligatorios
        private final String nombre;
        private final String email;

        // Campos opcionales
        private int edad;
        private String direccion;
        private String telefono;

        // Constructor con campos obligatorios
        public Builder(String nombre, String email) {
            this.nombre = nombre;
            this.email = email;
        }

        // Métodos para campos opcionales
        public Builder edad(int edad) {
            this.edad = edad;
            return this;
        }

        // Otros métodos setter...

        public Usuario build() {
            return new Usuario(this);
        }
    }
}
```

Con Lombok (¡Mucho más simple!)

```
@Builder
@Getter
public class Usuario {
    // Campos obligatorios
    @NonNull
    private String nombre;

    @NonNull
    private String email;

    // Campos opcionales
    private int edad;
    private String direccion;
    private String telefono;
}
```

Uso del Builder

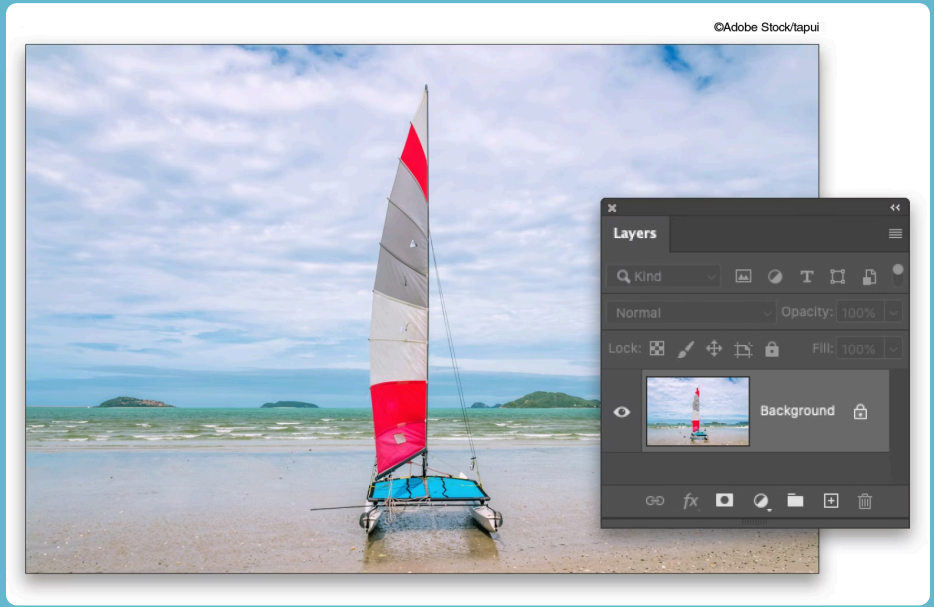
```
// Creación de objetos
Usuario usuario1 = Usuario.builder()
 .nombre("Carlos")
 .email("carlos@mail.com")
 .edad(25)
 .build();

Usuario usuario2 = Usuario.builder()
 .nombre("Ana")
 .email("ana@mail.com")
 .edad(30)
 .direccion("Calle Mayor 1")
 .telefono("666123456")
 .build();
```

Patrón Prototype: El Arte de la Clonación

El patrón Prototype permite crear nuevos objetos clonando un objeto existente (prototipo) en lugar de crear uno desde cero.

¿Cuándo usarlo?

- Cuando la creación de un objeto es costosa
- Cuando las clases a instanciar se especifican en tiempo de ejecución
- Para evitar construir una jerarquía de fábricas paralela
- Cuando las instancias pueden tener solo un conjunto limitado de estados



Tipos de Clonación

Shallow Copy: Copia los valores de los campos, pero reutiliza las referencias a objetos

Deep Copy: Copia recursivamente todos los objetos referenciados

Implementando Prototype para Préstamos

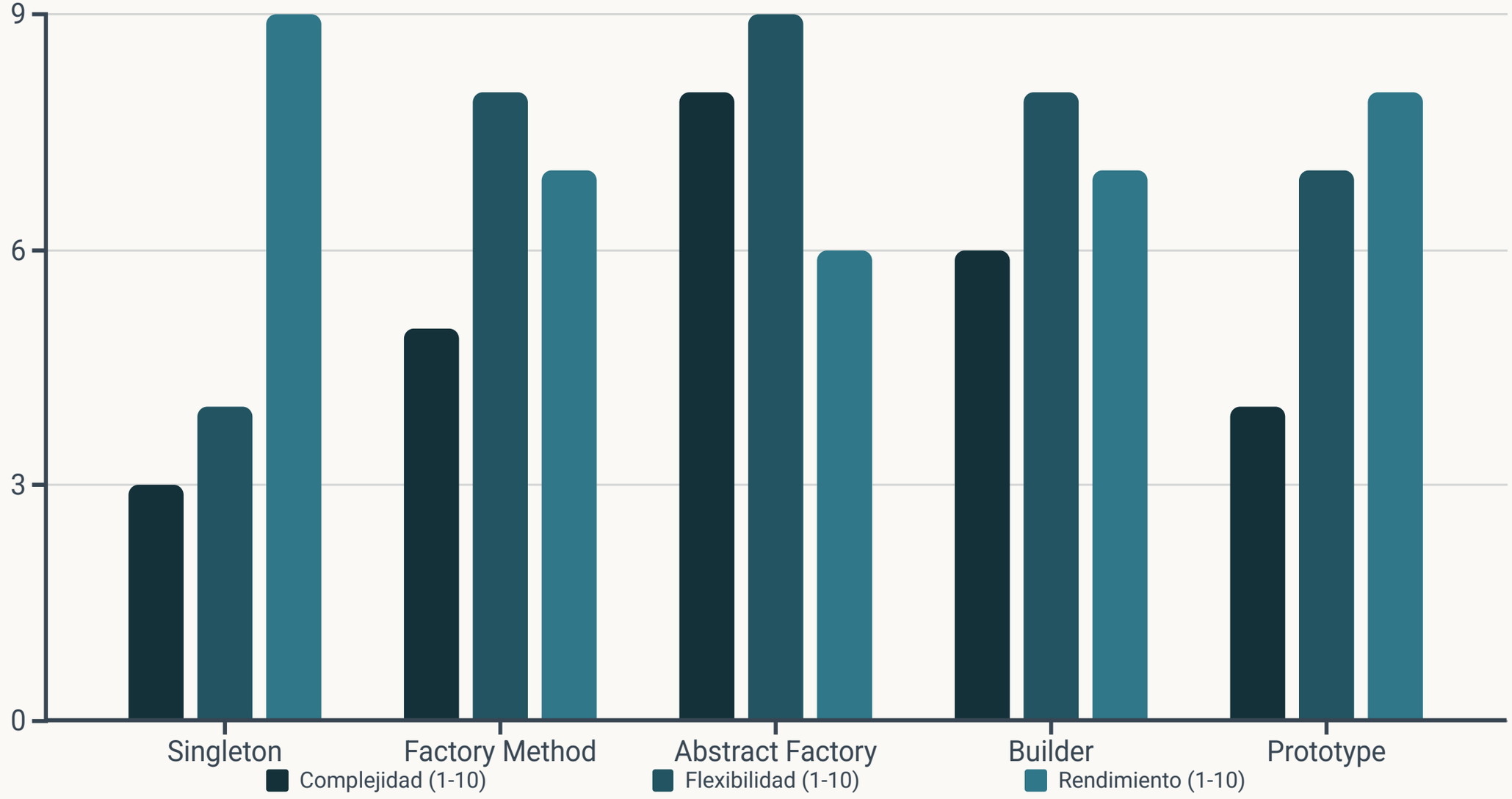
```
@Getter @Setter
@AllArgsConstructor
public class Prestamo implements Cloneable {
    private Libro libro;
    private Usuario usuario;
    private LocalDate fechaInicio;
    private LocalDate fechaFin;

    @Override
    public Prestamo clone() {
        try {
            // Shallow copy
            return (Prestamo) super.clone();
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }

    // Para deep copy
    public Prestamo deepClone() {
        try {
            Prestamo cloned = (Prestamo) super.clone();
            // Clonar objetos internos si es necesario
            return cloned;
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
}
```

Uso del patrón Prototype

```
// Crear prototipo base
Prestamo prestamoPrototipo = new Prestamo(
    libro,
    usuario,
    LocalDate.now(),
    LocalDate.now().plusDays(14)
);

// Clonar y modificar
Prestamo prestamo1 = prestamoPrototipo.clone();
prestamo1.setUsuario(otroUsuario);

Prestamo prestamo2 = prestamoPrototipo.clone();
prestamo2.setFechaFin(
    LocalDate.now().plusDays(30)
);

// Verificar independencia
System.out.println(prestamo1.getUsuario().getNombre());
System.out.println(prestamoPrototipo.getUsuario().getNombre());
```

Este enfoque es especialmente útil cuando tenemos préstamos con características similares pero algunas variaciones.

Comparativa de Patrones Creacionales



Cada patrón tiene sus propias fortalezas y casos de uso ideales. La clave está en saber cuándo aplicar cada uno según los requisitos específicos de nuestro proyecto.

Resumen y Próximos Pasos

Hemos explorado los cinco patrones creacionales fundamentales y cómo implementarlos en Java con Lombok:

Singleton

Para recursos únicos compartidos como nuestra base de datos

Factory Method

Para crear diferentes tipos de libros con una interfaz común

Abstract Factory

Para familias de objetos relacionados según el tipo de usuario

Builder

Para construir objetos complejos como usuarios paso a paso

Prototype

Para clonar préstamos existentes y modificarlos según necesidades

Ahora es vuestro turno de aplicar estos conocimientos en el trabajo práctico de la biblioteca digital. ¡Convertíos en maestros Jedi de los patrones!