



# Patrones de diseño

---

UNJu - Programación Orientada a Objetos



¿Así programamos?

---





## ¿Y así lo documentamos?

---

```
//  
// Querido programador:  
//  
// Cuando escribí este código, sólo Dios y yo  
// sabíamos cómo funcionaba.  
// Ahora, ¡sólo Dios lo sabe!  
//  
// Así que si está tratando de 'optimizar'  
// esta rutina y fracasa (seguramente),  
// por favor, incremente el siguiente contador  
// como una advertencia  
// para el siguiente colega:  
//  
// total_horas_perdidas_aquí = 189  
//
```



# Christopher Alexander

---

- Según Christopher Alexander, “cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces”. Inspiraciones
- Regiones independientes
  - fomenta la modularidad y la autonomía de componentes
- Comunidad de 7.000 personas
  - escalabilidad humana, como en equipos ágiles o microservicios.
- Pequeñas plazas públicas
  - interfaces centradas en el usuario, espacios de colaboración digital.
- Luz solar interior
  - diseño centrado en la experiencia del usuario, claridad y legibilidad.



04/10/1936 - 17/03/2022



# Clasificación

---

- Creacionales
  - Builder
  - Singleton
  - Abstract Factory, otros
- Estructurales
  - Decorator
  - DAO
  - Service Layer, otros
- Comportamiento
  - State
  - Strategy
  - otros



# Partes de un patrón de diseño

---

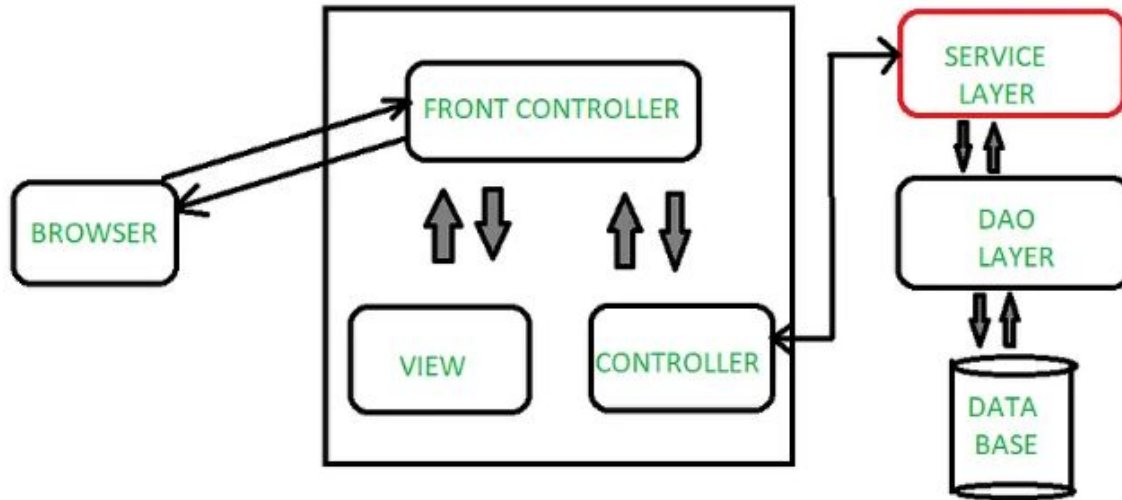
En general, un patrón tiene 4 partes esenciales:

- **Nombre del patrón:** describir en 1 o 2 palabras, un problema de diseño junto con sus soluciones y consecuencias.
- **El problema:** describe cuándo aplicar el patrón.
- **La solución:** describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones
- **Las consecuencias:** son los resultados así como las ventajas e inconvenientes de aplicar el patrón



# Arquitectura con capa de Servicios

- Gestiona las peticiones del controlador



Flow of an application using Service layer



# Consecuencias

---

- Separación de lógica de negocio
  - La lógica de negocios y las reglas se especifican en la capa de servicio que a su vez llama capa DAO, la capa DAO solo es responsable de interactuar con DB.
- Seguridad
  - Solo es posible acceder a la base de datos o a través del servicio.
- Bajo nivel de acoplamiento
  - Dado que un servicio puede contener varias operaciones de la base de datos y modificarlos puede ser imperceptible para quien consume el servicio



# Ejemplo conceptual

---

- Implementación en Java

```
@Service
public class CursoService {

    private final CursoRepository cursoRepository;

    public CursoService(CursoRepository cursoRepository) {
        this.cursoRepository = cursoRepository;
    }

    public List<Curso> obtenerCursosActivos() {
        return cursoRepository.findByActivoTrue();
    }
}
```



# Ventajas de la inyección por constructor en Spring Boot

---

- **Inmutabilidad y claridad:** los atributos se vuelven final por lo tanto no cambian
- **Facilita el testing:** no depender del contexto de spring para mocks o stubs
- **Mejora la legibilidad y el mantenimiento:** evidencia que se requiere de ese atributo para funcionar
- **Compatibilidad con herramientas de análisis estático:** sonarQube u otras
- **Evita errores de inyección silenciosos:** si una dependencia falta, el constructor falla al instanciar el bean, lo que facilita detectar errores temprano



# Patrón de diseño Inyección de Dependencia (DI)

---

- En este patrón de diseño se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos.
- Esos objetos cumplen contratos que necesitan nuestras clases para poder funcionar
- Existe un contexto o contenedor que es el encargado de inyectar las implementaciones deseadas
- Analizado por primera vez por Martin Fowler



# Motivación

---

- Alto nivel de acoplamiento entre componentes
- Aparecen nuevos conceptos en el diseño como la Inversión de Control (IoC) implementada a través de Inyección de Dependencias.

## Implementación en java

- Contenedor (spring framework)
- Declaración de Inyecciones



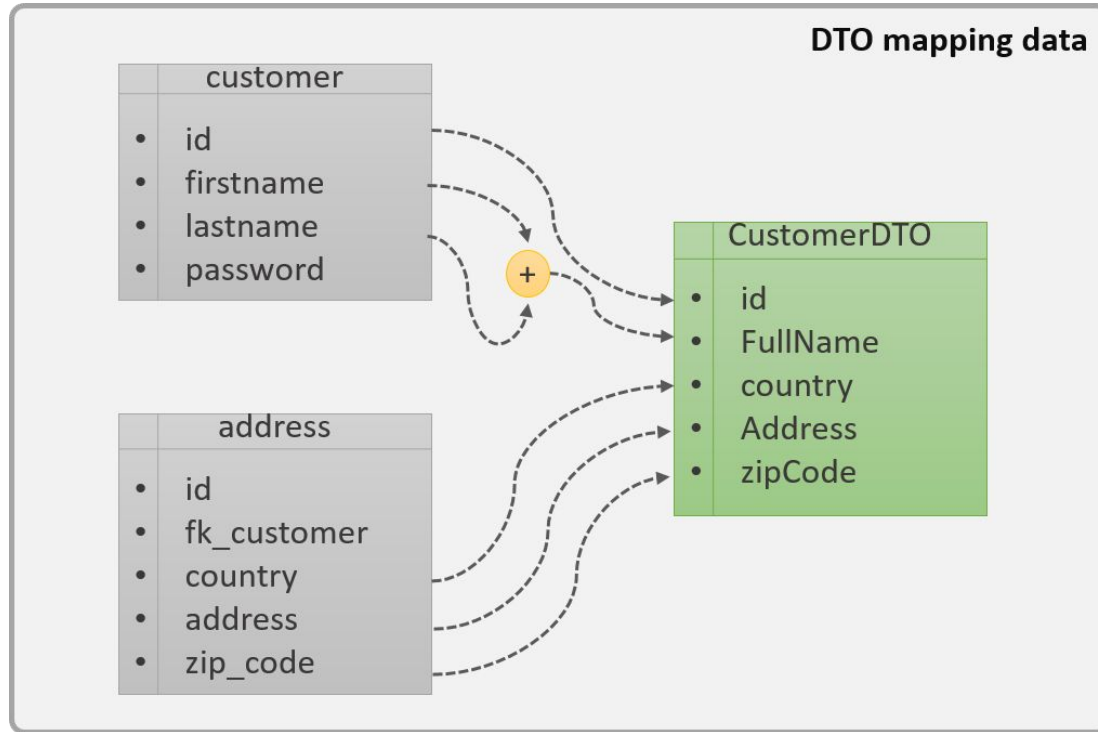
# Patrón de diseño DTO

---

- Concepto
  - Objeto de transferencia de datos
  - Define un objeto que transporta datos entre procesos
- Motivación
  - La comunicación entre procesos se realiza generalmente mediante interfaces remotas
  - Cada llamada es una operación costosa
  - Permite agregar datos de varias entidades
  - Protege atributos de las entidades



# Diagrama representativo





# Implementación en Java - Ejemplo

```
public class UserMapper {  
  
    // Convierte una entidad User en un DTO  
    public static UserDTO toDTO(User user) {  
        UserDTO dto = new UserDTO();  
        dto.setName(user.getName());  
        dto.setEmail(user.getEmail());  
        return dto;  
    }  
  
    // Convierte un DTO en una entidad User  
    public static User toEntity(UserDTO dto) {  
        User user = new User();  
        user.setName(dto.getName());  
        user.setEmail(dto.getEmail());  
        return user;  
    }  
}
```

```
@Service  
public class UserService {  
  
    private final UserRepository userRepository;  
  
    public UserService(UserRepository userRepository) {  
        this.userRepository = userRepository;  
    }  
  
    public UserDTO getUserById(Long id) {  
        User user = userRepository.findById(id)  
            .orElseThrow(() -> new RuntimeException("User not found"));  
        return UserMapper.toDTO(user);  
    }  
  
    public UserDTO createUser(UserDTO userDTO) {  
        User user = UserMapper.toEntity(userDTO);  
        user = userRepository.save(user);  
        return UserMapper.toDTO(user);  
    }  
}
```



# MapStruct

---

- MapStruct es un procesador de anotaciones que genera implementaciones de interfaces de mapeo entre clases Java.
- Al definir una interfaz con métodos de conversión y anotarla con @Mapper, MapStruct genera el código necesario para transformar entre DTOs y entidades.

```
<dependency>
  <groupId>org.mapstruct</groupId>
  <artifactId>mapstruct</artifactId>
  <version>1.6.2</version>
</dependency>
<dependency>
  <groupId>org.mapstruct</groupId>
  <artifactId>mapstruct-processor</artifactId>
  <version>1.6.2</version>
  <scope>provided</scope>
</dependency>
```



# Ejemplo

```
import org.mapstruct.Mapper;  
import org.mapstruct.factory.Mappers;
```

```
@Mapper(componentModel = "spring")  
public interface UserMapper {
```

```
    UserMapper INSTANCE = Mappers.getMapper(UserMapper.class);
```

```
    UserDTO toDTO(User user);  
    User toEntity(UserDTO dto);
```

```
}
```

```
@Service  
public class UserService {
```

```
    private final UserRepository userRepository;  
    private final UserMapper userMapper;
```

```
    public UserService(UserRepository userRepository, UserMapper userMapper) {  
        this.userRepository = userRepository;  
        this.userMapper = userMapper;  
    }
```

```
    public UserDTO getUserById(Long id) {  
        User user = userRepository.findById(id)  
            .orElseThrow(() -> new RuntimeException("User not found"));  
        return userMapper.toDTO(user);  
    }
```

```
    public UserDTO createUser(UserDTO userDTO) {  
        User user = userMapper.toEntity(userDTO);  
        user = userRepository.save(user);  
        return userMapper.toDTO(user);  
    }  
}
```



# Referencias

---

- [Why to use Service Layer in Spring MVC,](#)
- [Inyección de dependencias](#)
- [Data Transfer Object \(DTO\)](#)
- [Use Your DTOs in Java Like a Pro](#)
- [Patterns of Enterprise Application Architecture](#)