



**UNCUYO**  
UNIVERSIDAD  
NACIONAL DE CUYO



**FACULTAD  
DE INGENIERÍA**

## Ingeniería en Software 2

### Trabajo Práctico Integrador N° 1 Proyecto “Gimnasio Sport”

Grupo 3:

Alvarez Lucía

Olivares Agustin

Padilla Gonzalo

## Índice

<b>Introducción.....</b>	<b>2</b>
<b>Especificación del sistema.....</b>	<b>3</b>
<b>Diagrama de Clases de Diseño.....</b>	<b>4</b>
<b>Estado Actual del Sistema.....</b>	<b>5</b>
<b>Ubicación en el Proceso RUP.....</b>	<b>5</b>
<b>Requisitos del Sistema.....</b>	<b>5</b>
<b>Caso de uso crítico.....</b>	<b>5</b>
<b>Requisitos Funcionales.....</b>	<b>6</b>
<b>Diagrama de Caso de Uso.....</b>	<b>7</b>
<b>Escenarios de Caso de Uso.....</b>	<b>8</b>
Alta Rutinas.....	9
Baja Rutina.....	10
Modificar Rutina.....	11
<b>Diagrama de Secuencia de Diseño.....</b>	<b>11</b>
Alta Rutina.....	12
Modificar Rutina.....	12
Baja Rutina.....	13
<b>Prototipado.....</b>	<b>13</b>
<b>Diagrama de Paquete de Software.....</b>	<b>15</b>
<b>Diagrama Entidad Relación.....</b>	<b>17</b>
<b>Requisitos No Funcionales.....</b>	<b>17</b>
<b>Patrones de diseño.....</b>	<b>19</b>
Inyección de Dependencias.....	19
MVC.....	19
Capas.....	20
DTO.....	21
GRASP.....	21
1. Controller.....	21
2. Creator.....	22
3. Information Expert (Experto en la información).....	22
4. Low Coupling (Bajo acoplamiento).....	22
5. High Cohesion (Alta cohesión).....	22
6. Polymorphism.....	23
7. Pure Fabrication (Fabricación pura).....	23
8. Indirection (Indirección).....	24
9. Protected Variations (Variaciones protegidas).....	24
<b>Patrones GOF.....</b>	<b>24</b>
Clasificación de los patrones GoF.....	25
Ejemplos de uso en el proyecto.....	26
<b>Funcionalidad Propuesta.....</b>	<b>28</b>
Diagrama de Clases de Diseño Funcionalidad Propuesta.....	29

## Introducción

El presente proyecto busca integrar los contenidos trabajados en clase mediante el desarrollo de un sistema de gestión para un gimnasio. El sistema permite el inicio de sesión de usuarios, diferenciando entre socios y empleados. Los socios pueden consultar sus rutinas, verificar y abonar sus cuotas mediante Mercado Pago, mientras que los empleados se dividen en profesores, encargados de crear rutinas personalizadas, y un administrador, responsable de la creación de usuarios y del control de cuotas adeudadas. Además, el sistema incorpora el envío automático de correos electrónicos en fechas relevantes, como cumpleaños o eventos especiales.

Para su implementación se aplicó la metodología RUP (Rational Unified Process), organizada en las fases de Inicio, Elaboración, Construcción y Transición. Actualmente, el proyecto se encuentra en la fase de Construcción, en la cual se materializan los diseños y modelos previos en funcionalidades concretas.

Este documento presenta una descripción conceptual de cada etapa y su aplicación al sistema desarrollado junto con los diagramas realizados además de información sobre patrones implementados.

## Especificación del sistema

El equipo de desarrollo fue contratado por el Gimnasio “Sport” para realizar un Software que permita administrar la gestión de asociados/as que el mismo posee. Entre las funcionalidades que el software debe contemplar se encuentra:

- A. Alta, Consulta, Modificación y Baja de: socios/as y usuarios del sistema con sus respectivos roles.
- B. Gestión de cobro de cuota mensual.
- C. Gestión de Deuda.
- D. Generación y envío de campañas promocionales por correo electrónico.
- E. Saludos por el cumpleaños de asociados/as.

- F. Gestión y seguimiento de rutina entregadas por los profesores de educación física a los socios/as.

El sistema deberá contar con tres tipos de perfiles que habilitarán las funcionalidades según los permisos que tengan cada rol de los usuarios en el sistema.

- A. Usuario Administrador: podrá acceder a todas las funcionalidades del sistema.
- B. Empleado: podrá acceder a todas las funcionalidades exceptuando Alta, Consulta, Modificación de Usuario y Alta del valor de la cuota mensual.
- C. Asociado/a: podrá acceder a la gestión de rutina y pago de cuota mensual para el caso que se realice con Mercado Pago. Además, podrá acceder al informe de deuda donde se detalla el total de la deuda y los meses adeudados.

## Diagrama de Clases de Diseño

En este diagrama de clases se refleja la estructura del modelo, contemplando entidades principales como Empleado, Socio, Usuario, Rutina, Membresía, Pago y Clase, además de las relaciones necesarias para sostener la lógica del sistema

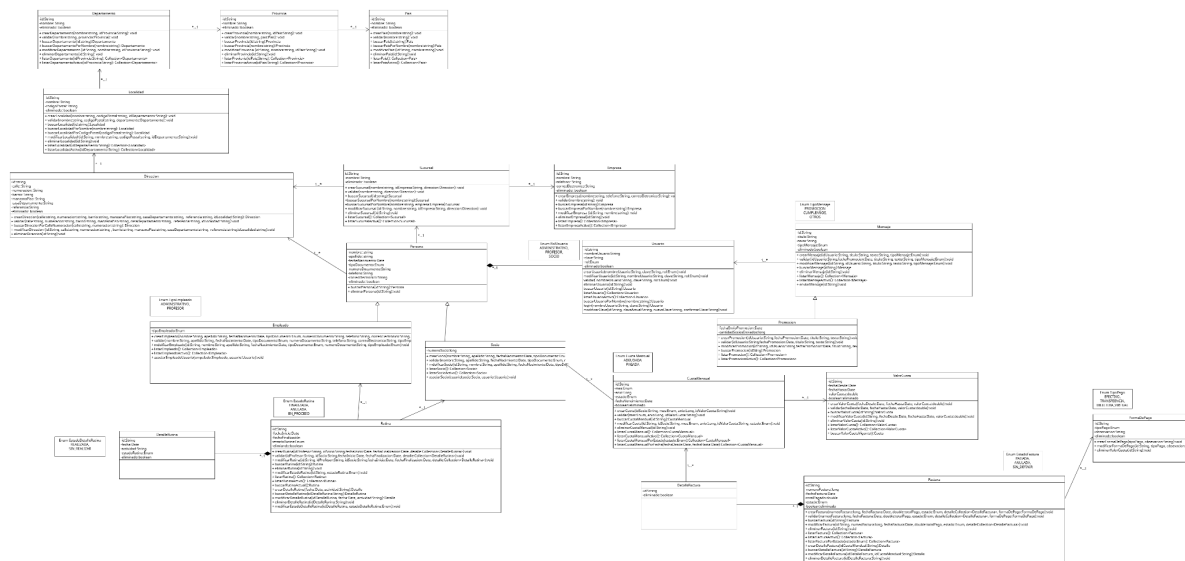


Diagrama Clases de Diseño.png

## Estado Actual del Sistema

El desarrollo se encuentra avanzado, con las funcionalidades principales implementadas. Restan finalizar detalles en la interfaz, tales como:

- Ajustes de responsividad en algunos botones y vistas.
- Corrección de funcionalidades en los botones de modificación y visualización dentro de los listados de entidades.

## Ubicación en el Proceso RUP

Este entregable corresponde principalmente a las fases de:

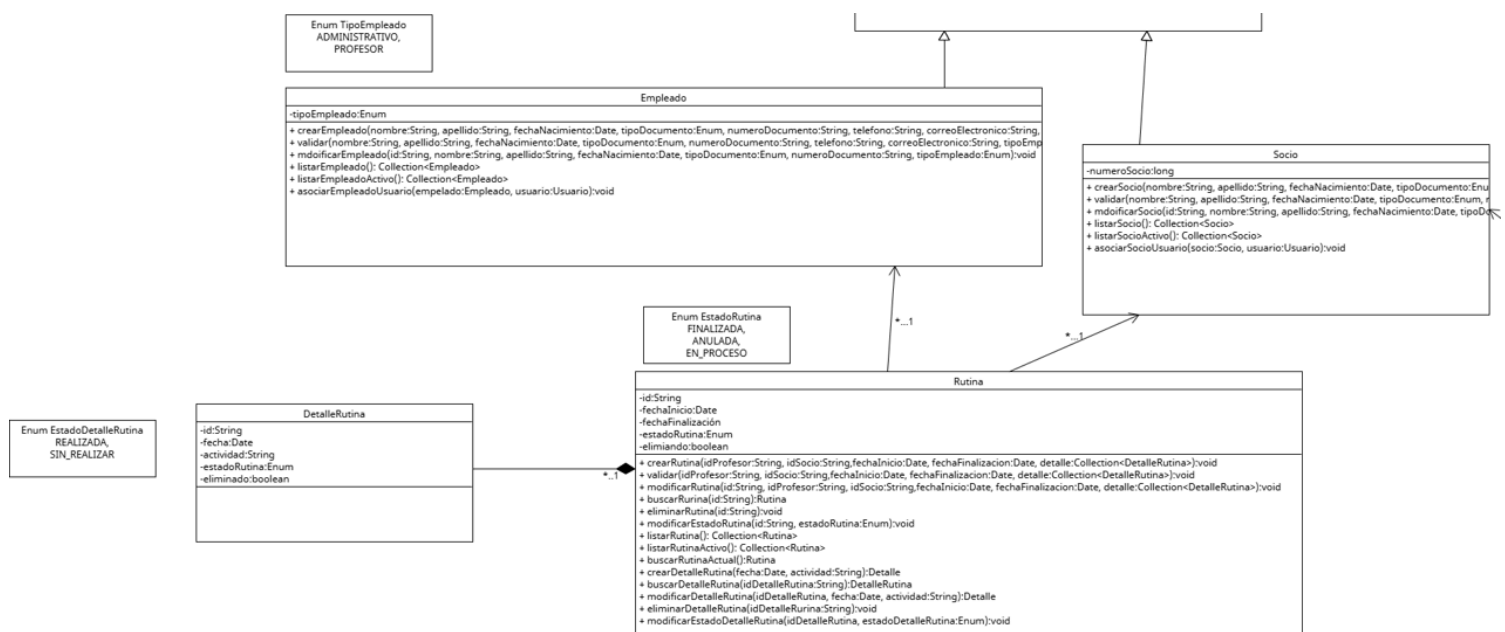
- Elaboración, ya que incluye la definición detallada del alcance, los diagramas de diseño y el modelo de datos.
- Construcción, dado que se han implementado las funcionalidades principales del sistema y se está avanzando en pruebas y ajustes de la interfaz.

## Requisitos del Sistema

El sistema desarrollado para el Gimnasio Sport tiene como objetivo principal brindar soporte a la gestión administrativa y operativa de la institución. A continuación se presentan los requisitos funcionales y no funcionales que guían su diseño e implementación.

## Caso de uso crítico

- ABM Rutina (crítico)



## Requisitos Funcionales

### 1. Gestión de usuarios y roles

- El administrador podrá crear, modificar, eliminar y consultar usuarios del sistema.
- Se deberán distinguir roles con permisos diferenciados: Administrador, Profesor y Socio.

## 2. Gestión de empleados y socios

- Alta, baja, modificación y consulta de empleados y socios del gimnasio.
- Asociación de un empleado al rol de profesor o administrativo.
- Consulta de cuotas adeudadas por los socios.

## 3. Gestión de cuotas, valores y pagos.

- Creación de cuotas mensuales y asignación de valores.
- Emisión de cuotas mensuales y control de cuotas activas, pagadas y adeudadas.
- Registro de pagos realizados por los socios.
- Integración con Mercado Pago para pagos online.
- Generación de comprobantes y actualización automática del estado de la cuota.

## 4. Gestión de rutinas

- Los profesores podrán crear, modificar y asignar rutinas de entrenamiento personalizadas a los socios.
- El socio podrá visualizar sus rutinas desde el sistema.

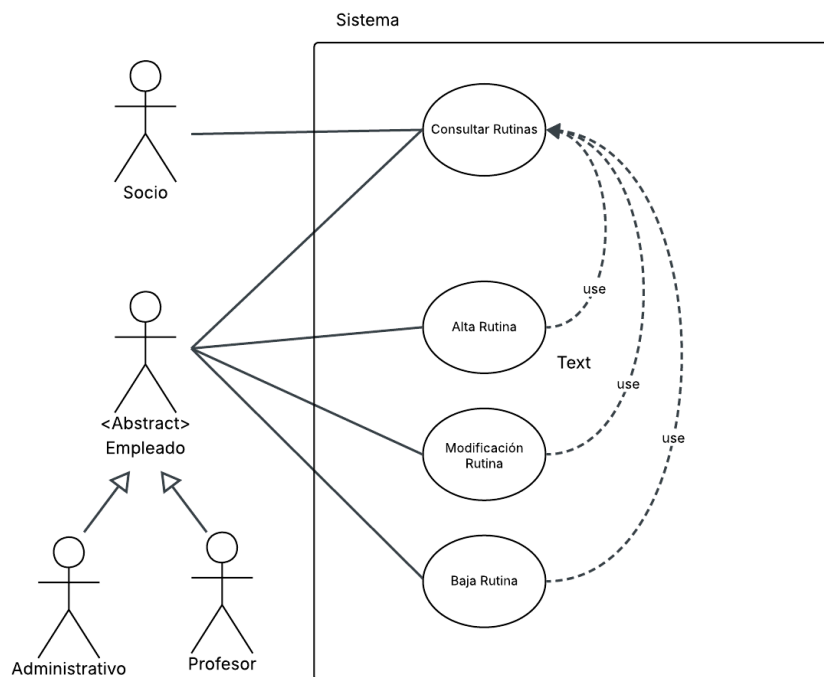
## 5. Comunicación con socios

- Envío automático de correos electrónicos en fechas especiales (cumpleaños, promociones, eventos).
- Notificación de cuotas próximas a vencer o vencidas.

## Diagrama de Caso de Uso

Este diagrama representa las funcionalidades principales del sistema desde la perspectiva del usuario, en este caso respecto a la rutina.

Muestra actores (usuarios u otros sistemas) y los casos de uso (acciones que realizan). Se usa para definir requisitos funcionales y entender qué espera el usuario del sistema.



## Escenarios de Caso de Uso

Los escenarios de caso de uso son una descripción textual y detallada de cómo se ejecuta un caso de uso. Se explica paso a paso la interacción entre el usuario y el sistema en el caso de la rutina desde el punto de vista del profesor. Sirve para aclarar el flujo normal, excepciones y condiciones y se usa como base para diseñar pruebas y diagramas posteriores.



## Alta Rutinas

Caso de uso	Alta Rutina			ID:
Prioridad	Critica	Estimacion:		ID Pantalla prototipo
Precondicion	Existencia de Socio Registrado, Existencia de Profesor Registrado, Coincidencia entre las Sucursales de ambos			
Descripcion	El Profesor da de alta una Rutina junto con los Detalles para el Socio			
Escenario principal	Paso	Accion		
	1	El sistema muestra las rutinas existentes para cada socio a cargo		
	2	El profesor selecciona el socio		
	3	El profesor ingresa el estado de la rutina		
	4	El sistema carga fecha de inicio y de finalizacion por defecto		
	5	El profesor modifica las fechas (opcional)		
	6	El profesor ingresa los detalles de la rutina		
	7	El profesor confirma las configuraciones de la rutina		
	8	El sistema valida las configuraciones		
	9	El sistema crea la rutina		
Postcondicion				
Excepciones	5.1	Si la fecha termina antes de la creacion, Muestra mensaje de error y se vuelve al paso 5		
	6.1	Si la fecha del detalle no coincide con el periodo de la rutina, muestra mensaje de error y vuelve al paso 6		
	6.2	Si los campos de detalle tienen caracteres especiales, muestra mensaje de error y vuelve al paso 6		
Comentarios	Luego de crear la rutina, esta figura tambien para el socio y retorno al paso 1			

EscenarioCasoAlta.png


## Baja Rutina

Caso de uso	Baja Rutina			ID:
Prioridad	Critica	Estimacion:		ID Pantalla prototipo
Precondicion	Existencia previa de al menos una Rutina, Existencia de Profesor Registrado, Existencia de Socio Registrado, Coincidencia entre ambos en la sucursal			
Descripcion	El Profesor da de baja una Rutina junto con los Detalles para el Socio			
Escenario principal	Paso	Accion		
	1	El sistema muestra las rutinas existentes para cada socio a cargo		
	2	El profesor selecciona la rutina a eliminar		
	3	El sistema elimina de forma logica la rutina		
Postcondicion				
Excepciones				
Comentarios	Luego de dar de baja la rutina, los detalles de esta tambien son dados de baja, y se vuelve al paso 1. El socio y el profesor ya no puede ver la rutina			

EscenarioCasoBaja.png

## Modificar Rutina

Caso de uso	Modificacion Rutina		ID:
Prioridad	Critica	Estimacion:	ID Pantalla prototipo
Precondicion	Existencia previa de al menos una Rutina, Existencia de Profesor Registrado, Rutina asignada al profesor		
Descripcion	El Profesor modifica una Rutina existente, la cual pertenece a un Socio		
Escenario principal	Paso	Accion	
	1	El sistema muestra las rutinas existentes para cada socio a cargo	
	2	El profesor selecciona la rutina a modificar	
	3	El sistema devuelve la rutina seleccionada	
	4	El profesor modifica los campos (opcional)	
	5	El profesor agrega detalles a la rutina (opcional)	
	6	El profesor confirma las nuevas configuraciones	
	7	El sistema valida las nuevas configuraciones	
	8	El sistema modifica la rutina existente	
Postcondicion			
Excepciones	4.1	Si la fecha termina antes de la creacion, Muestra mensaje de error y se vuelve al paso 4	
	5.1	Si la fecha del detalle no coincide con el periodo de la rutina, muestra mensaje de error y vuelve al paso 6	
Comentarios	Luego de modificar la rutina, esta figura tambien modificada para el socio y retorno al paso 1		

 EscenarioCasoModificacion.png

## Diagrama de Secuencia de Diseño

En este diagrama se muestra cómo los objetos y clases del sistema se comunican entre sí en el tiempo para ejecutar un caso de uso del ABM rutina. Utiliza mensajes y llamadas a métodos ordenados cronológicamente. Se usa para diseñar la lógica interna y verificar cómo se reparten las responsabilidades.

## Alta Rutina

## Alta Rutina

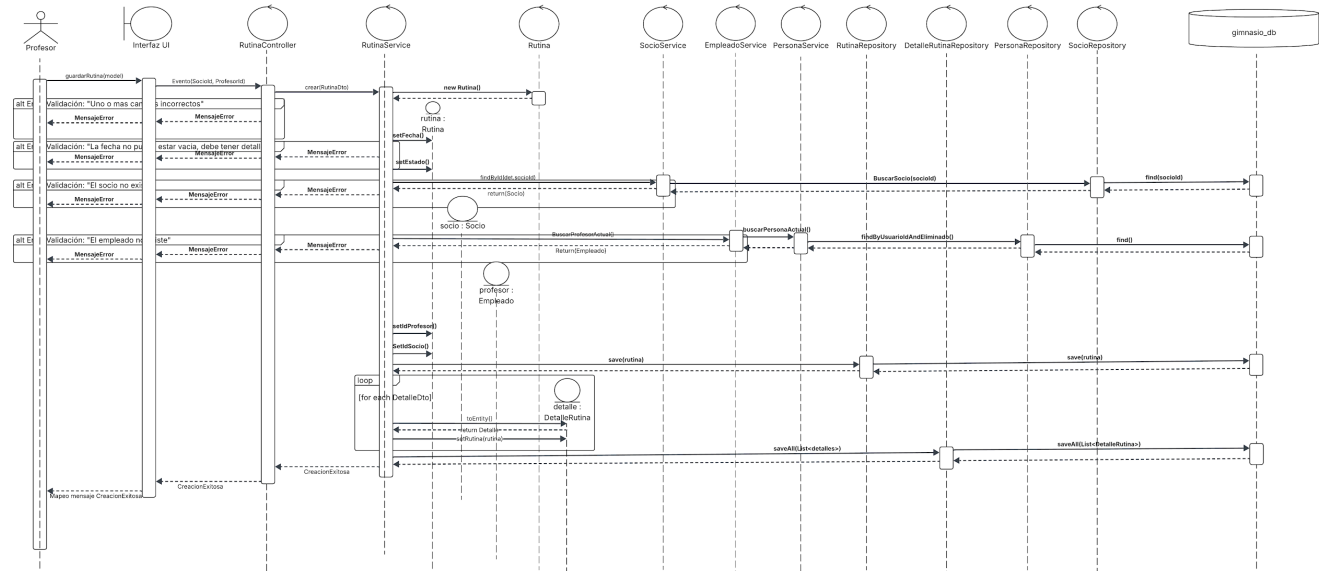


Diagrama Alta.png

## Modificar Rutina

## Modificar Rutina

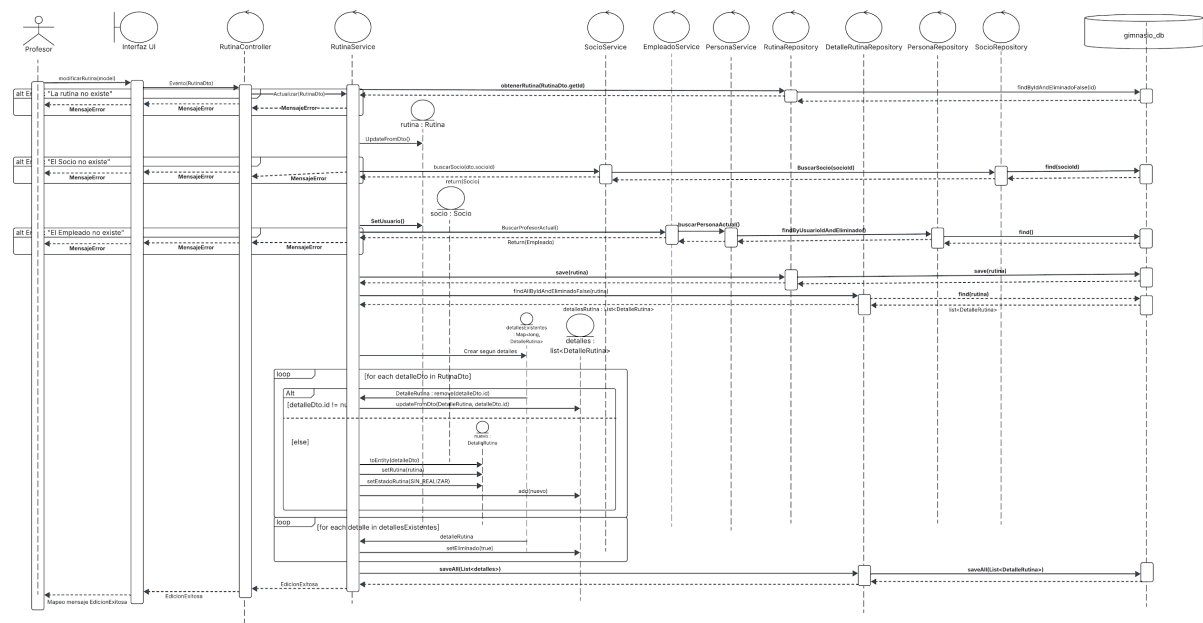


Diagrama Modificacion.png

## Baja Rutina

# Baja Rutina

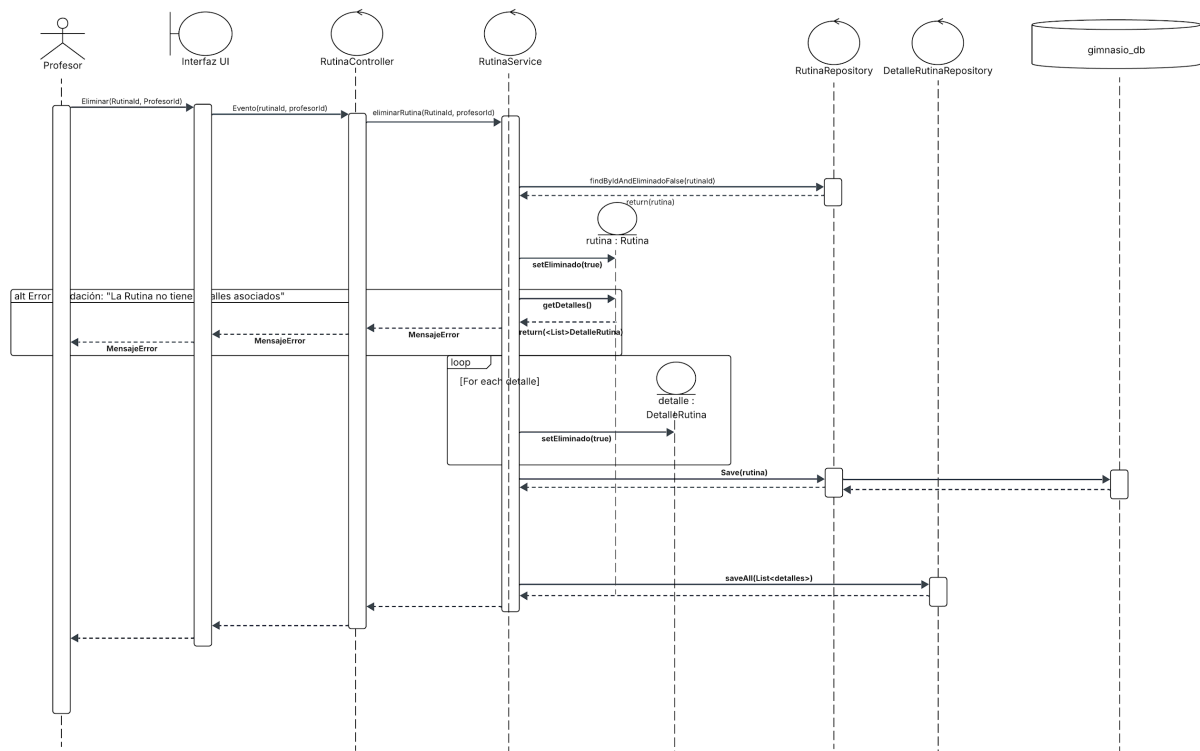


Diagrama Baja.png

## Prototipado

Los prototipos son representaciones gráficas de cómo se verán las pantallas del sistema. Se usa para validar la interfaz con el usuario antes de implementarla. Ayuda a detectar problemas de usabilidad y definir la navegación. En este caso se realizaron desde el punto de vista del profesor para un ABM rutina y desde el punto de vista del socio para ver sus rutinas.

Gimnasio Sport

[←](#)
[→](#)
[↻](#)

Panel de Rutinas

Nueva Rutina

Selecione un Socio

Estado Rutina

Fecha Inicio

Fecha Fin

Actividades:

Agregar Actividad

Socio a

Id	Estado	Inicio	Fin	Detalles	Acciones
1	En Proceso	año_mes_día	año_mes_día	nombre_actividad1 nombre_actividad2	

Socio b

Id	Estado	Inicio	Fin	Detalles	Acciones
2	Finalizada	año_mes_día	año_mes_día	nombre_actividad1 nombre_actividad2	

Selecione un Socio

Estado Rutina

Fecha Inicio

Fecha Fin

Actividades:

Agregar Actividad

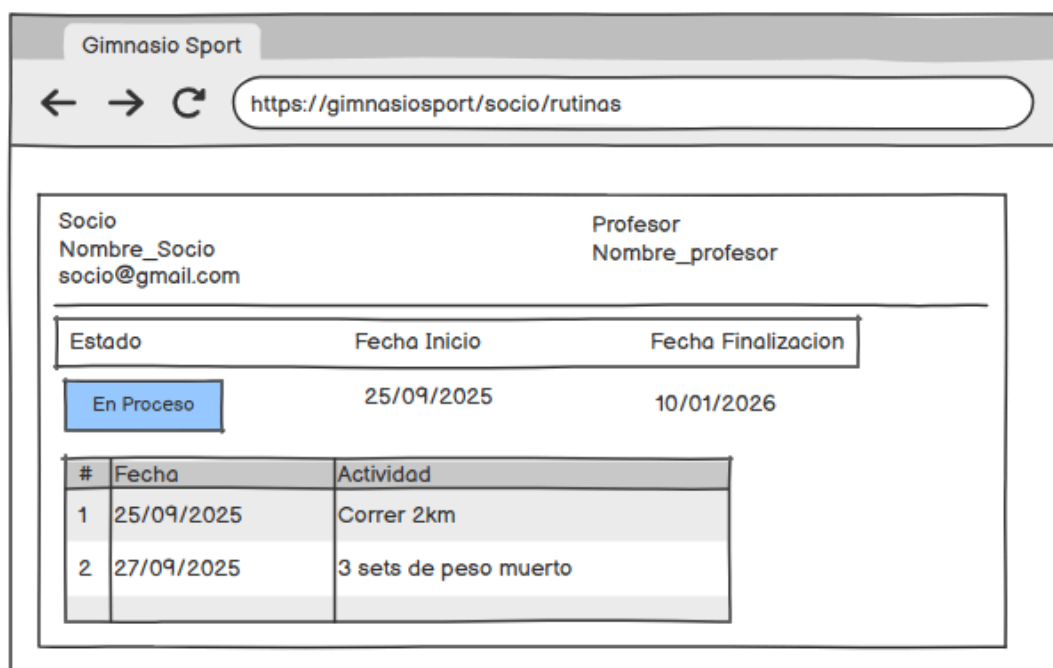
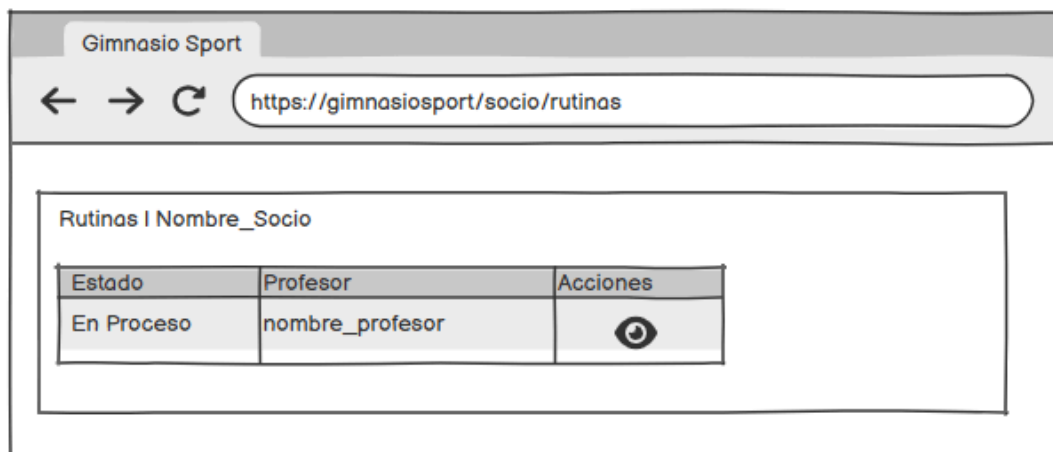
Fecha de la Actividad

/

/

Actividad

Guardar



## Diagrama de Paquete de Software

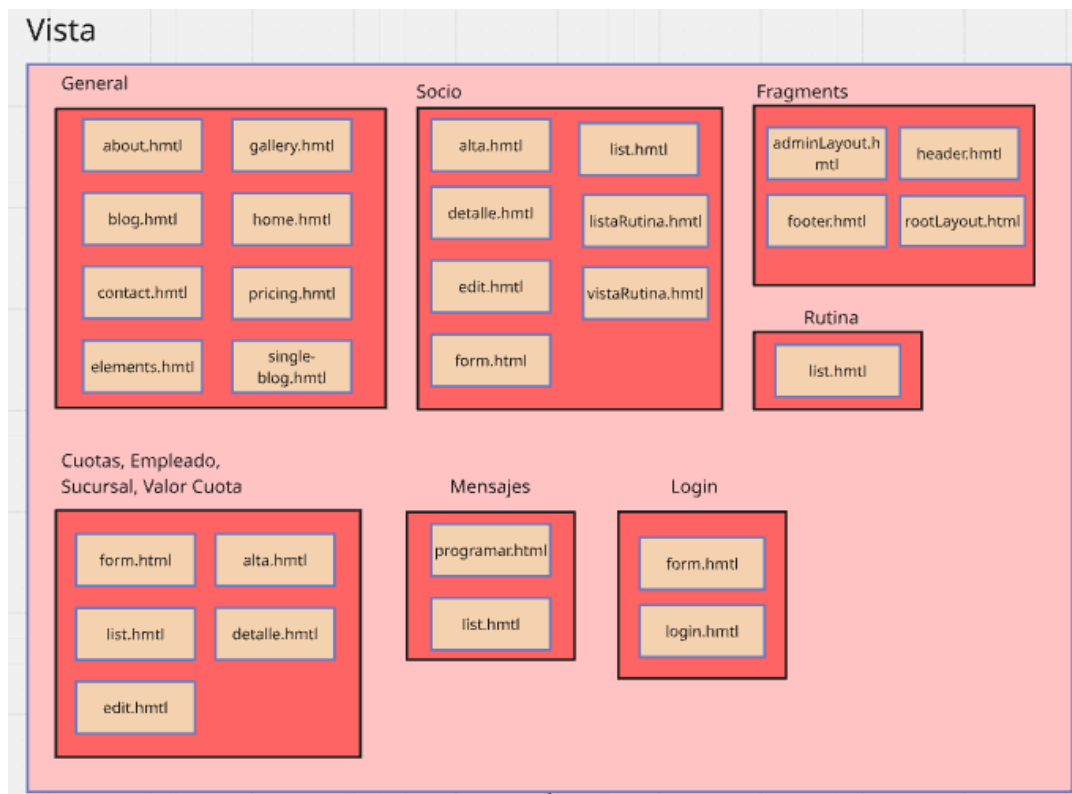
📄 Miro

El diagrama de paquetes organiza el sistema en módulos lógicos o "paquetes", mostrando cómo se relacionan entre sí. Es útil porque da una visión arquitectónica a alto nivel, sin entrar en detalle de clases individuales.

Ayuda a entender la estructura general del proyecto y muestra dependencias entre capas. Además permite explicar cómo está implementado el patrón MVC junto con paquetes de soporte (Auth, Config, Error, Enums, Init, Jobs).

En este proyecto vemos los siguientes paquetes:

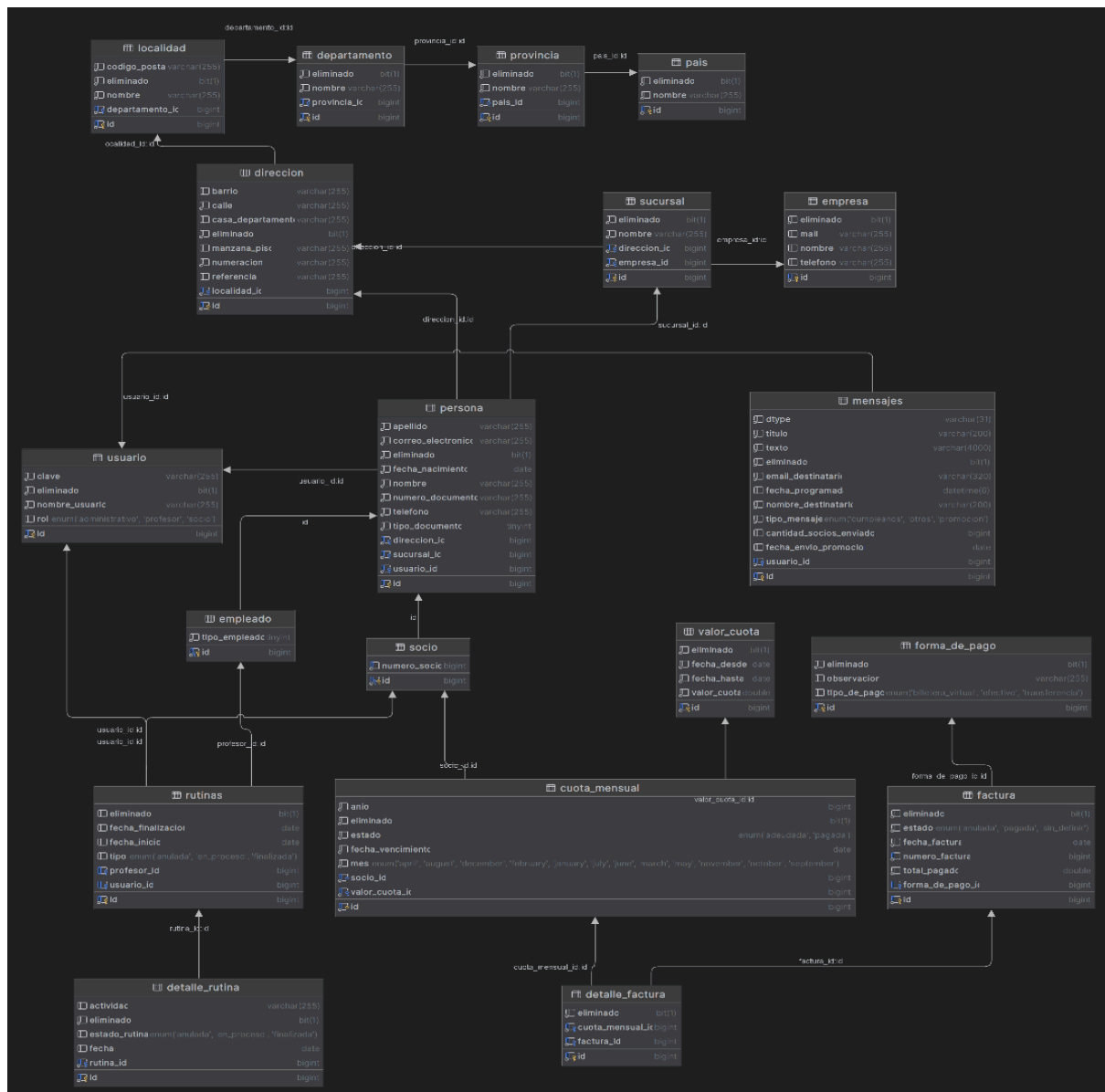
- Paquete **controller** gestiona las solicitudes (ej: EmpleadoController, SocioController).
- Paquete **service** contiene la lógica (ej: PagoService, RutinaService).
- Paquete **repository** maneja acceso a BD.
- Paquete **entity** define las tablas (Socio, Empleado, Rutina, Membresía).
- Paquetes **dto** y **mapper** aíslan la capa de presentación.
- Paquetes de **soporte** (auth, config, error, jobs, etc.) atraviesan la arquitectura.



[https://miro.com/app/board/uXjVJDg9C84=](https://miro.com/app/board/uXjVJDg9C84=/)



## Diagrama Entidad Relación



DER.png

## Requisitos No Funcionales

### 1. Seguridad

- El sistema debe implementar autenticación de usuarios con credenciales únicas.

- Los roles deben restringir el acceso a funcionalidades no autorizadas.
- La información sensible (contraseñas, datos de pago) debe almacenarse encriptada.

## 2. Usabilidad

- La interfaz debe ser intuitiva y clara.
- Los mensajes de error deben ser comprensibles para los usuarios.

## 3. Rendimiento

- Las consultas de listados (socios, empleados, pagos) deben responder en menos de 3 segundos.

## 4. Mantenibilidad

- El código debe estar estructurado en capas (controladores, servicios, repositorios).
- Se deben documentar las principales clases y métodos.

## 5. Escalabilidad

- El sistema debe permitir incorporar nuevas funcionalidades (ejemplo: reservas online) sin alterar la arquitectura existente.

## 6. Compatibilidad

- Debe ser accesible desde navegadores modernos (Chrome, Firefox, Edge).
- Debe integrarse correctamente con la API de Mercado Pago.

# Patrones de diseño

## Inyección de Dependencias

Spring automáticamente inyecta las dependencias que declaremos en nuestras clases, siempre que encuentre una (única) implementación y declaremos un constructor (en este caso con Lombok), que reciba tales dependencias como parámetros. Por ejemplo, la siguiente clase depende de tres interfaces. Spring inyectará automáticamente instancias de las implementaciones disponibles al momento de ejecución.

```
@Service
@RequiredArgsConstructor
public class UsuarioService {
    private final UsuarioRepository usuarioRepository;
    private final PasswordEncoder passwordEncoder;
    private final UsuarioMapper usuarioMapper;

    public Usuario crearUsuario(UsuarioCreateFormDTO formDto) {
        validarDatos(formDto.getClave(), formDto.getConfirmacionClave());

        if (usuarioRepository.existsByNombreUsuarioAndEliminadoFalse(formDto.getNombreUsuario()))
            throw new BusinessException("YaExiste.usuario.nombre");

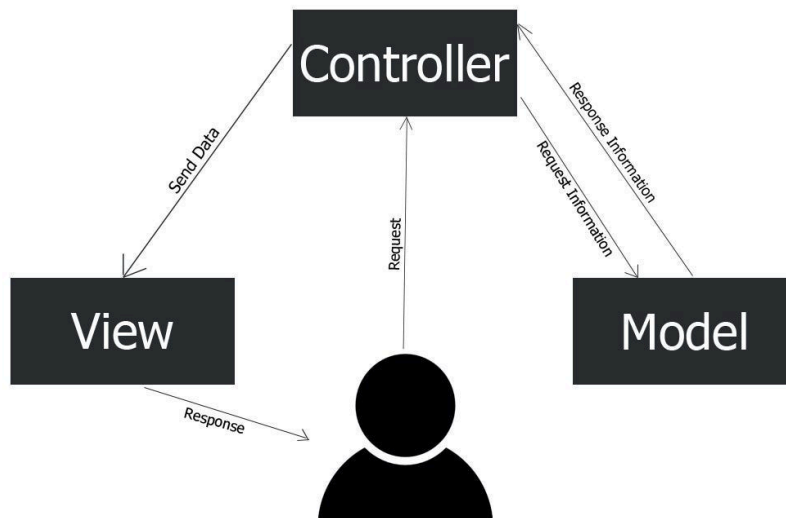
        Usuario usuario = usuarioMapper.toEntity(formDto);
        String hashClave = passwordEncoder.encode(formDto.getClave());
        usuario.setClave(hashClave);

        return usuarioRepository.save(usuario);
    }
}
```

## MVC

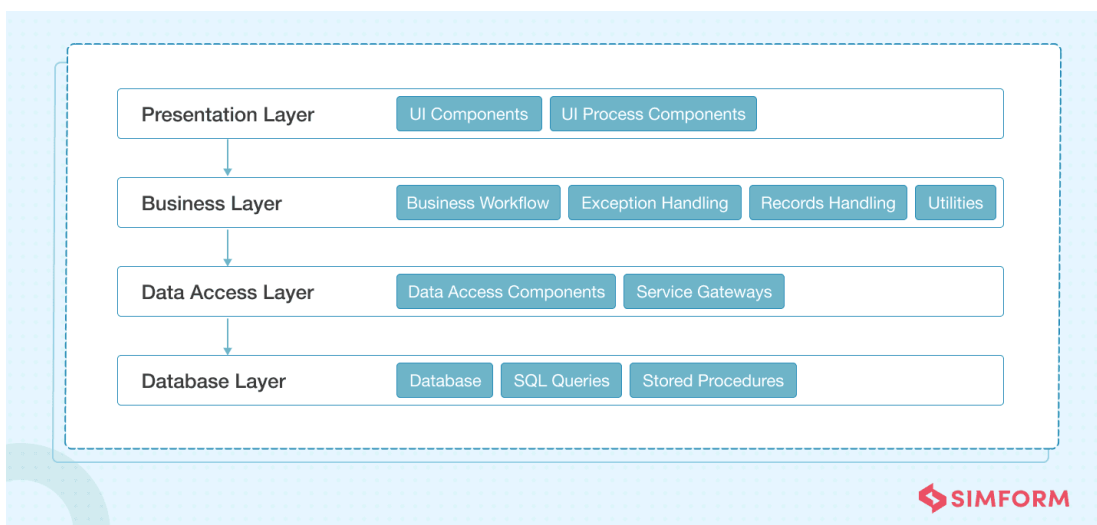
Utilizamos el patrón MVC para organizar el código del proyecto de tal forma de reducir el acoplamiento entre clases, y separar la lógica de negocio (entidades, repositorios, servicios), de los detalles de las distintas vistas, y manejo de rutas. En el mismo, una clase controlador es la que responde a los eventos del usuario. Esta se comunica con el modelo (lógica de negocio) para realizar cambios y obtener respuestas. Luego, inserta los datos en una vista que es devuelta al usuario.

## Model-View-Controller



## Capas

En aplicaciones web se suele usar una arquitectura en capas para separar responsabilidades y hacer el sistema más fácil de mantener. Normalmente hay cuatro capas: presentación, que es lo que ve y usa el usuario (páginas web, botones, formularios); lógica de negocio, donde están las reglas y procesos que definen cómo funciona la aplicación; acceso a datos, que sirve de puente entre la lógica y la base de datos; y datos, que es donde se guarda la información. Cada capa tiene su función y se comunica con la siguiente, evitando que todo quede mezclado.



## DTO

Utilizamos DTOs (Data Transfer Objects) para simplificar el paso de información entre clases de distintas capas, preparar resúmenes de datos para las distintas vistas, y para la validación de formularios.

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class SocioResumenDto {
    private Long id;
    private Long numeroSocio;
    private String nombre;
    private String apellido;
    private String correoElectronico;
    private String nombreSucursal;
}
```

## GRASP

Patrones Generales de Asignación de Responsabilidades en Software. Son un conjunto de principios de diseño orientado a objetos que sirven como guía para decidir qué clase debe tener qué responsabilidad dentro de un sistema. No son patrones concretos como Singleton o Factory, sino lineamientos conceptuales para estructurar el código.

### 1. Controller

- Asignar la responsabilidad de manejar un evento del sistema a un objeto controlador.
  - Los controllers (EmpleadoController, ClienteController, etc.) reciben las solicitudes HTTP (alta, baja, modificación, consultas).
  - Son los controladores GRASP porque median entre la Vista (HTML/Thymeleaf) y el Modelo (servicios/entidades).

- No hacen la lógica de negocio, solo delegan.

## 2. Creator

- Un objeto debería ser responsable de crear instancias de objetos que usa o contiene.
  - Por ejemplo, la clase FacturaService es la encargada de crear, modificar y dar de baja no solo a las entidades de tipo Factura sino también a las de tipo DetalleFactura y FormaDePago
  - Otro ejemplo es la clase Rutina, que está encargada de crear tanto entidades de tipo Rutina como de DetalleRutina.

## 3. Information Expert (Experto en la información)

- Asignar la responsabilidad a la clase que tiene la información necesaria para cumplirla.
  - Por ejemplo, la clase PaisService es la encargada del alta, baja y modificación de las entidades País, ya que es la que tiene la información necesaria para ello.
  - EmpleadoService coordina la lógica de creación, baja y modificación de Empleados porque tiene acceso a los repositorios apropiados.

## 4. Low Coupling (Bajo acoplamiento)

- Minimizar dependencias entre clases.
  - Los Controllers dependen solo de los Servicios, no directamente de los repositorios ni entidades.
  - Los Services dependen de interfaces (JpaRepository), no de implementaciones concretas.
  - Esto permite probar, cambiar la capa de persistencia o incluso reemplazarla sin modificar el resto.

## 5. High Cohesion (Alta cohesión)

- Mantener juntas las responsabilidades relacionadas.

- Los Services agrupan toda la lógica de negocio de una entidad (ej. EmpleadoService se encarga de altas, bajas, modificaciones de empleados).
- Los Repositories se enfocan únicamente en el acceso a datos.
- Los Controllers se manejan las peticiones HTTP.
- Así cada capa tiene cohesión y un propósito claro.

## 6. Polymorphism

- Usar polimorfismo para manejar variaciones en el comportamiento.
  - Por ejemplo, el siguiente método es encargado de poblar los campos de una Persona, la cuál puede ser un Empleado o un Socio.

```
@Transactional
protected void crearPersona(Persona persona, PersonaCreateFormDTO formDto) {
    UsuarioCreateFormDTO usuarioDto = formDto.getUsuario();
    usuarioDto.setNombreUsuario(formDto.getCorreoElectronico());
    Usuario usuario = usuarioService.crearUsuario(usuarioDto);

    Direccion direccion = direccionService.crearDireccion(formDto.getDireccion());

    Sucursal sucursal = sucursalService.buscarSucursal(formDto.getSucursalId());

    personaMapper.updateEntityFromDto(formDto, persona);
    persona.setSucursal(sucursal);
    persona.setUsuario(usuario);
    persona.setDireccion(direccion);
    persona.setId(null);
    persona.setEliminado(false);
}
```

## 7. Pure Fabrication (Fabricación pura)

- Crear clases auxiliares que no representan un concepto del dominio, para mantener bajo acoplamiento y alta cohesión.
  - Los DTOs y Mappers son fabricaciones puras: no existen en el dominio real del gimnasio, pero ayudan a desacoplar el modelo de datos de la capa de presentación.

## 8. Indirection (Indirección)

- Asignar responsabilidades a un intermediario para reducir el acoplamiento.
  - Una clase de servicio es un intermediario entre Controller y Repository.
  - El Mapper es un intermediario entre Entity y DTO.
  - Por ejemplo, el patrón MVC.

## 9. Protected Variations (Variaciones protegidas)

- Proteger el sistema de cambios en partes inestables mediante interfaces o abstracciones.
  - Por ejemplo, el uso de JpaRepository abstrae el acceso a la BD (si se cambia de Postgres a MySQL, el código de negocio no se rompe).
  - Los DTO protegen la Vista de cambios en las entidades internas.
  - Los Services protegen a los controladores de cambios en la lógica interna o repositorios.

## Patrones GOF

Cuando se habla de “patrones GoF” se hace referencia a veintitrés soluciones que se organizaron en tres grandes categorías: creacionales, estructurales y de comportamiento. Cada patrón aborda un problema recurrente en el desarrollo de software orientado a objetos y propone una forma estandarizada de resolverlo, de modo que los diseñadores y programadores puedan reutilizar esa experiencia en lugar de reinventar la rueda.

GoF representa un cuerpo de conocimiento fundamental que introdujo prácticas de diseño reutilizables y claras, que siguen vigentes y sirven de base en el desarrollo moderno de software.



## Clasificación de los patrones GoF

### 1. Patrones creacionales

Resuelve cómo se crean los objetos, buscando que la construcción sea flexible y desacoplada de las clases concretas.

- a. Singleton → asegura que exista una sola instancia global.
- b. Factory Method → delega la creación de objetos a subclases.
- c. Abstract Factory → fabrica familias de objetos relacionados sin especificar clases concretas.
- d. Builder → construye objetos complejos paso a paso.
- e. Prototype → crea nuevos objetos clonando otros ya existentes.

### 2. Patrones estructurales

Se ocupan de cómo se componen y organizan las clases y los objetos para formar estructuras más grandes y fáciles de mantener.

- a. Adapter → adapta una interfaz para que sea compatible con otra.
- b. Bridge → separa una abstracción de su implementación para que evolucionen de forma independiente.
- c. Composite → organiza objetos en jerarquías árbol (parte-todo).
- d. Decorator → añade funcionalidades dinámicamente sin modificar la clase original.
- e. Facade → provee una interfaz simple a un subsistema complejo.
- f. Flyweight → reutiliza objetos para ahorrar memoria en elementos repetidos.
- g. Proxy → actúa como sustituto o intermediario de otro objeto.

### 3. Patrones de comportamiento

Se enfocan en cómo los objetos interactúan entre sí, cómo se reparten las responsabilidades y cómo se comunican de manera eficiente.

- a. Chain of Responsibility → pasa una petición por una cadena de objetos hasta que alguien la atienda.
- b. Command → encapsula una acción en un objeto.
- c. Interpreter → define reglas para interpretar un lenguaje o gramática.

- d. Iterator → permite recorrer colecciones sin exponer su estructura interna.
- e. Mediator → centraliza la comunicación entre objetos para evitar dependencias directas.
- f. Memento → guarda y restaura el estado interno de un objeto.
- g. Observer → notifica automáticamente a varios objetos cuando otro cambia de estado.
- h. State → cambia el comportamiento de un objeto según su estado interno.
- i. Strategy → define distintas estrategias intercambiables para resolver un mismo problema.
- j. Template Method → define el esqueleto de un algoritmo y delega algunos pasos a subclases.
- k. Visitor → permite agregar operaciones a objetos sin modificar sus clases.

## Ejemplos de uso en el proyecto

Algunos de estos patrones son utilizados por el Framework Spring.

### Singleton

Spring genera instancias únicas de las clases que anotamos con la anotación `@Service` para mejorar el rendimiento. Luego pasa esta instancia a las demás clases que la requieran, mediante inyección de dependencias, sin crear una instancia adicional por cada clase o método que la requiera.

```
@Service
@RequiredArgsConstructor
public class PaisService {
```

## Factory Method

En Spring, podemos declarar clases anotadas con `@Configuration`, donde los métodos anotados con `@Bean` serán los utilizados por el Framework para crear las implementaciones de alguna interfaz que sea necesaria. Por ejemplo, hacemos uso de esto en la configuración de Spring Security, donde proveemos métodos factory que proveen implementaciones de las interfaces `SecurityFilterChain`, que especifica la configuración de seguridad de Spring Security, y `PasswordEncoder`, que retorna el tipo de encriptación a utilizar.

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    @Autowired(required = false)
    private DevAutoLoginFilter devAutoLoginFilter;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        http
            .csrf( CsrfConfigurer<HttpSecurity> csrf -> csrf.disable())
            .authorizeHttpRequests( AuthorizationManagerRequestMat... auth -> auth
                .requestMatchers(✓"/css/**", ✓"/fonts/**", ✓"/img/**", ✓"/js/
                .requestMatchers(✓"/", ✓"/about", ✓"/blog", ✓"/contact", ✓"/
                    ✓"gallery", ✓"pricing", ✓"single-blog", ✓"/error", ✓
                .requestMatchers(✓"/países/**").hasRole("ADMINISTRATIVO")
                .anyRequest().authenticated()
            )

        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }
}
```

## Proxy

Spring utiliza este patrón para generar proxies alrededor de los métodos que anotamos con `@Transactional`, entre otras anotaciones. Este proxy consiste de código que intercepta las llamadas a tales métodos, maneja excepciones, logging, etc. Esto reduce la cantidad de código que debe realizar el programador, por ejemplo, para manejar el rollback de transacciones en caso de excepciones.

```
@Transactional
public void modificarPais(PaisDto paisDto) {
    Pais pais = buscarPais(paisDto.getId());

    if (paisRepository.existsByNombreAndIdNotAndEliminadoFalse(paisDto.getNombre(), paisDto.getId()))
        throw new BusinessException("YaExiste.pais.nombre");

    paisMapper.updateEntityFromDto(paisDto, pais);
    paisRepository.save(pais);
}
```

## Funcionalidad Propuesta

En los sistemas de gimnasio es muy común que los socios puedan anotarse a clases específicas que sean dictadas en ciertos horarios por los profesores del establecimiento.

En este caso no está la funcionalidad y resulta interesante para próximas revisiones agregarla.

### 1. Requisitos de la funcionalidad

- Un socio puede reservar una clase disponible.
- Una clase tiene: nombre, día, horario, profesor asignado y cupo máximo.
- Un profesor dicta una o más clases.
- El socio puede anotarse siempre que haya cupos disponibles.
- El sistema registra la reserva y permite consultar las reservas de un socio o las inscripciones a una clase.

### 2. Entidades involucradas

- Socio (id, nombre, email,, etc.)
- Profesor (id, nombre, etc.)
- Clase (id, nombre, día, horario, id\_profesor, cupo\_max)
- Reserva (id, id\_socio, id\_clase, fecha\_reserva)

Relación:

- Un profesor dicta muchas clases.
- Un socio puede hacer muchas reservas.
- Una clase puede tener muchos socios anotados → relación N:M entre Socio y Clase, resuelta con la entidad Reserva.

## Diagrama de Clases de Diseño Funcionalidad Propuesta

