Identificación de los patrones de diseño utilizados por el Framework de seguridad

El equipo seleccionó **Spring Security** como framework para la implementación de la seguridad de la aplicación, mediante autenticación y autorización de usuarios. Este framework está basado en una arquitectura modular y extensible que utiliza múltiples **patrones de diseño** para lograr una estructura flexible, mantenible y desacoplada.

# 1. Patrón Strategy

**Descripción:**  
Permite definir una familia de algoritmos (en este caso, distintos métodos de autenticación o codificación de contraseñas) y seleccionarlos en tiempo de ejecución sin modificar el código cliente.

**Aplicación en Spring Security:**  
Spring Security utiliza **Strategy** en componentes como:

* PasswordEncoder → distintas estrategias de cifrado (por ejemplo, BCryptPasswordEncoder, Pbkdf2PasswordEncoder, etc.).
* AuthenticationProvider → diferentes estrategias de autenticación (por base de datos, en memoria, LDAP, JWT, OAuth2).

**Ejemplo en el código del proyecto:**

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder(); // Estrategia concreta de encriptación

}

En este caso, la clase BCryptPasswordEncoder es una **implementación concreta** de la estrategia de cifrado definida por la interfaz PasswordEncoder.

# 2. Patrón Chain of Responsibility

**Descripción:**  
Permite que una solicitud sea procesada por una cadena de objetos, donde cada uno puede manejar o pasar la solicitud al siguiente.

**Aplicación en Spring Security:**  
Se utiliza en la **cadena de filtros (Security Filter Chain)**.  
Cada filtro maneja una parte del proceso de seguridad (autenticación, autorización, login, logout, CSRF, etc.), y si no puede procesar la petición, la pasa al siguiente filtro de la cadena.

**Ejemplo en el proyecto:**

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(...)

.formLogin(...)

.logout(...)

.csrf(...);

return http.build();

}

Cada configuración agregada (authorizeHttpRequests, formLogin, logout, csrf) se traduce internamente en un **filtro especializado** que forma parte de la **cadena de responsabilidad** del framework.

# 3. Patrón Decorator

**Descripción:**  
Permite añadir dinámicamente nuevas responsabilidades a un objeto sin modificar su estructura original.

**Aplicación en Spring Security:**  
Se aplica al envolver objetos de autenticación (Authentication) y usuarios (UserDetails) para agregar información adicional como roles, permisos o tokens.  
Ejemplo: al crear un UserDetails que extiende de otro usuario base para añadir roles o banderas de cuenta activa.

UserDetails user = User.builder()

.username("admin")

.password(passwordEncoder().encode("1234"))

.roles("ADMIN")

.build();

Spring Security **decora** al usuario base con roles, estados de cuenta, fechas de expiración, etc., sin alterar la clase original.

# 4. Patrón Proxy

**Descripción:**  
Permite controlar el acceso a un objeto sustituyéndolo por otro que actúa como intermediario.

**Aplicación en Spring Security:**  
Los proxies dinámicos de Spring AOP son usados para interceptar métodos marcados con anotaciones de seguridad como @PreAuthorize, @Secured o @RolesAllowed.  
El proxy verifica los permisos del usuario antes de permitir la ejecución real del método.

**Ejemplo:**

@PreAuthorize("hasRole('ADMIN')")

public void eliminarUsuario(String id) { ... }

Aquí, Spring crea un **proxy** del servicio que evalúa la expresión de seguridad antes de ejecutar el método.

# 5. Patrón Factory Method

**Descripción:**  
Centraliza la creación de objetos delegando la lógica de instanciación a una clase concreta.

**Aplicación en Spring Security:**  
El framework utiliza fábricas internas para construir los diferentes componentes de seguridad, por ejemplo, los beans de AuthenticationManager, UserDetailsService, SecurityFilterChain, o PasswordEncoder.  
El método http.build() del ejemplo es un **Factory Method** que devuelve una instancia configurada de la cadena de filtros.

# 6. Patrón Observer (Event Listener)

**Descripción:**  
Define una relación de “suscriptor” entre objetos, donde un cambio en uno notifica a los demás.

**Aplicación en Spring Security:**  
Se utiliza para **eventos de seguridad**, como inicio de sesión exitoso o fallido (InteractiveAuthenticationSuccessEvent, AuthenticationFailureBadCredentialsEvent).  
Esto permite ejecutar lógica personalizada (por ejemplo, auditorías o logs) cuando ocurre un evento de seguridad.

**🧠 Conclusión**

Spring Security combina varios patrones de diseño para lograr una arquitectura sólida y extensible:

| **Categoría** | **Patrón** | **Función en el Framework** |
| --- | --- | --- |
| **Comportamiento** | Strategy | Selección flexible de algoritmos de autenticación o cifrado |
| **Comportamiento** | Chain of Responsibility | Procesamiento secuencial de filtros de seguridad |
| **Estructural** | Decorator | Añadir roles, permisos y datos sin modificar clases base |
| **Estructural** | Proxy | Interceptar métodos y validar seguridad mediante anotaciones |
| **Creacional** | Factory Method | Construcción flexible de componentes (SecurityFilterChain, PasswordEncoder) |
| **Comportamiento** | Observer | Gestión de eventos de seguridad (login/logout) |

En conjunto, estos patrones permiten que Spring Security sea **altamente configurable**, **desacoplado** y **extensible**, manteniendo un fuerte control sobre la seguridad sin necesidad de reescribir la infraestructura base.