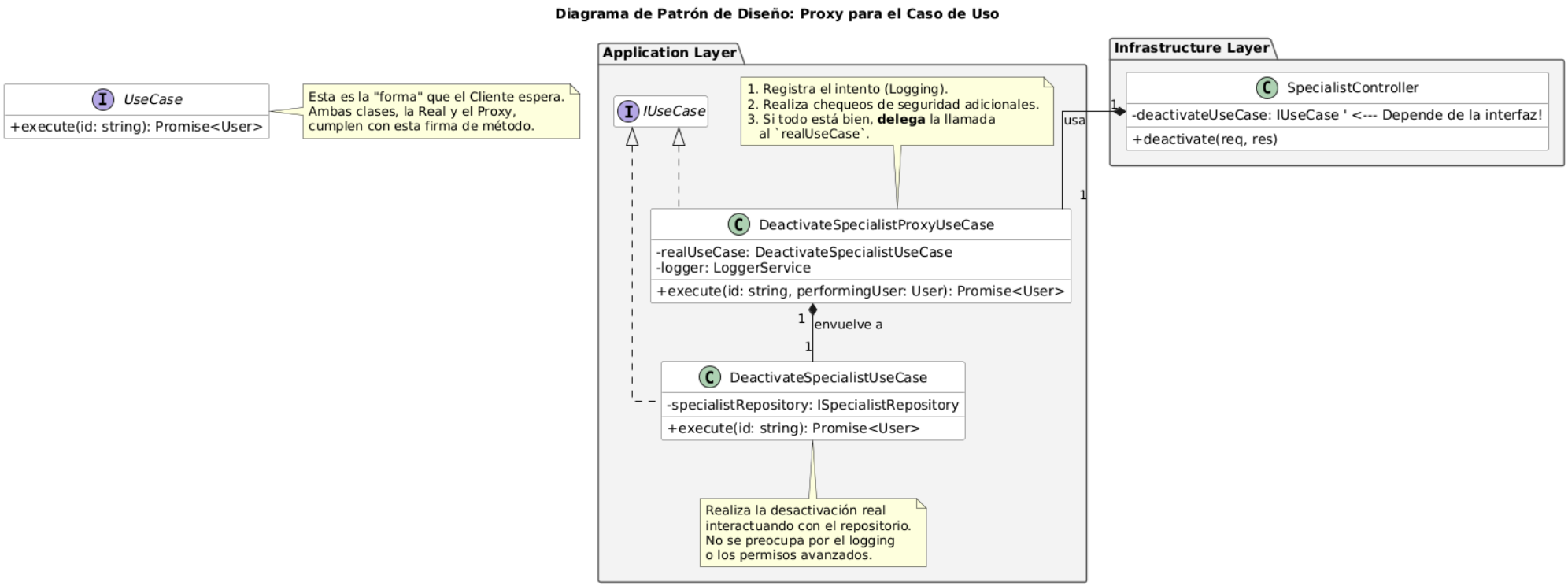
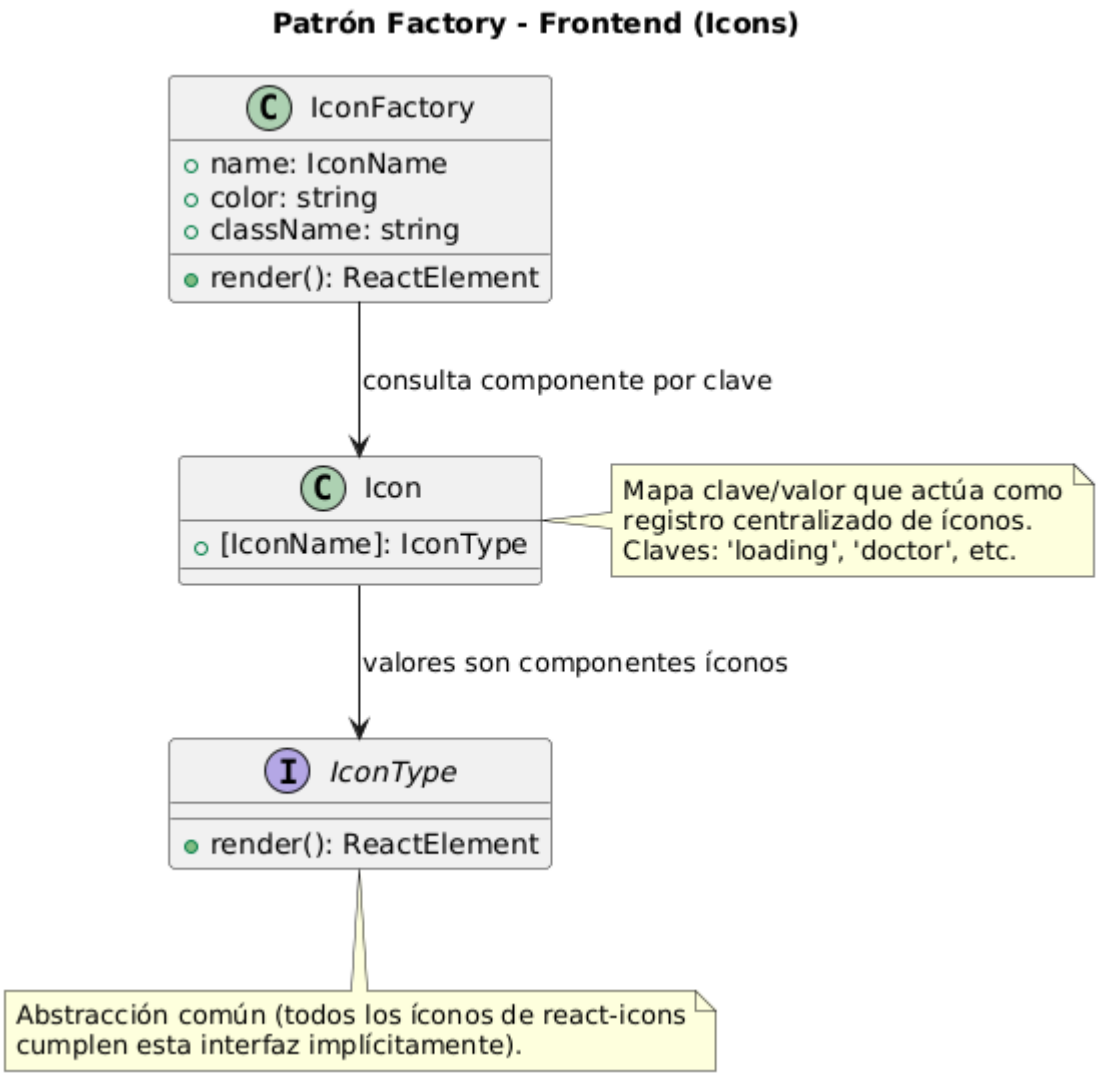


Flujo de Funcionamiento del Proxy

1. El SpecialistController recibe una petición para desactivar a un especialista.
2. En lugar de llamar directamente a DeactivateSpecialistUseCase, llama a nuestro DeactivateSpecialistProxyUseCase.
3. El **Proxy** recibe la llamada y realiza sus tareas adicionales:
    - **Control de Acceso:** Podría verificar de nuevo el rol del usuario que hace la petición (req.user.role).
    - **Logging:** Registra en la consola o en un archivo de log: "Intento de desactivación del especialista con ID [xxx] por el admin [yyy].".
4. Si todas las verificaciones del proxy pasan, **delega la llamada** al objeto DeactivateSpecialistUseCase real, que contiene la lógica de negocio principal.
5. El resultado de la operación real se devuelve al proxy, y este, a su vez, lo devuelve al controlador.

Este patrón se compone de tres elementos clave:

1. **El Sujeto Real (DeactivateSpecialistUseCase):** Es la clase que contiene la lógica de negocio fundamental para desactivar un especialista. Su única responsabilidad es interactuar con el repositorio para cambiar el estado del usuario.
2. **El Proxy (DeactivateSpecialistProxyUseCase):** Es una clase que actúa como un intermediario o "envoltorio" del Sujeto Real. Implementa la misma interfaz (es decir, tiene el mismo método execute), por lo que desde la perspectiva del cliente, son intercambiables. Antes de delegar la llamada al objeto real, el Proxy realiza tareas adicionales:
    - **Logging:** Registra cada intento de desactivación, creando una pista de auditoría.
    - **Control de Acceso:** Puede realizar verificaciones de permisos adicionales si fuera necesario.
3. **El Cliente (SpecialistController):** Es la clase que inicia la operación. En nuestra arquitectura, el controlador depende de la interfaz del caso de uso. Al momento de la inyección de dependencias, en lugar de pasarle la instancia del DeactivateSpecialistUseCase real, se le pasa la instancia del DeactivateSpecialistProxyUseCase. El controlador no es consciente de este cambio y simplemente llama al método execute, interactuando con el Proxy como si fuera el objeto original.



El Patrón Factory se utiliza para encapsular la creación de objetos, permitiendo instanciarlos de forma dinámica sin acoplar el código a clases concretas. Este patrón se usa para generar componentes de íconos según un identificador textual (name).

Flujo del patrón:

1. El componente IconFactory recibe un name (por ejemplo: "doctor").
2. Este name se usa como clave para buscar en el objeto ICONS, que actúa como registro centralizado de componentes de íconos.
3. El componente correspondiente (FaUserDoctor, por ejemplo) se obtiene dinámicamente del mapa.
4. Se devuelve e instancia el componente con las props necesarias (color, className).
5. Si no existe el nombre solicitado, se retorna null de forma segura.

Elementos clave del patrón:

Elemento	Rol dentro del patrón
IconFactory	**Fábrica**: punto central de creación de íconos. Se encarga de decidir qué componente devolver según la entrada.
ICONS	**Registro de clases**: almacén de componentes, mapea nombres (IconName) a componentes de react-icons.
IconName	**Clave del producto**: entrada controlada (tipada) para buscar el componente correcto.
ReactIcon (implícito)	**Producto común**: todos los componentes del mapa comparten una estructura común (son componentes de React que aceptan props).
color, className	**Parámetros de construcción**: permiten personalizar la instancia renderizada (estilo, animación, etc.).