Universidad Nacional de Colombia - sede Bogotá
Facultad de Ingeniería
Departamento de Sistemas e Industrial
Curso: Ingeniería de Software 1 (2016701)

Estudiantes:

Federico Hernández Montano
Juan David Montenegro López
Diego Esteban Ospina Ladino
Juan David Serrano Ruiz

Patrón de Diseño

🧭 1. Introducción

El presente documento tiene como objetivo justificar por qué actualmente no se implementó un patrón de diseño estructurado en el sistema **CACVi-UN**, y cuál sería el patrón más adecuado para implementar en versiones futuras, especialmente cuando el sistema sea migrado a una arquitectura web distribuida y en tiempo real.

📐 2. Estado actual del sistema

CACVi-UN es una aplicación de escritorio desarrollada en **React + Electron**, orientada a mejorar la seguridad dentro del campus de la Universidad Nacional de Colombia sede Bogotá. Permite que miembros de la comunidad universitaria:

- Se registren e inicien sesión (solo con correos @unal.edu.co)
- Reporten incidentes de violencia usando un formulario validado y geolocalización en un mapa (Leaflet)
- Naveguen según su rol (usuario, admin, etc.) con un sistema de permisos dinámico
- Visualicen un mapa y una sección de estadísticas (estructura aún en desarrollo)

El estado de sesión y autenticación se maneja con **React Context**, permitiendo compartir el rol, nombre y correo entre todos los componentes de la app (Login, Register, Report, Map, Statistics, Header).

🤷‍♂️ 3. ¿Por qué actualmente no se usa un patrón de diseño?

El sistema, en su versión actual, **no requiere aún un patrón formal de diseño**, por las siguientes razones:

- La aplicación es **pequeña y monolítica**, con pocas vistas y una sola entrada de datos (el reporte).
- No existen múltiples componentes interdependientes que reaccionen a cambios de estado en tiempo real.
- El flujo es lineal y controlado (inicio de sesión → acción → salida).
- No hay necesidad todavía de desacoplar módulos lógicos complejos o manejar actualizaciones cruzadas entre vistas.

Por lo tanto, forzar un patrón en esta etapa temprana habría sido una **sobrerregulación innecesaria**, que solo complicaría el mantenimiento.

🧩 4. ¿Se usó algún patrón de forma parcial?

Sí. Aunque no se implementó un patrón formal completo, **sí se aplicó de forma parcial el patrón Service Locator** 🔍, mediante el uso del **Contexto de React** (AuthContext) y un hook personalizado (useAuth()).

Esto permite:

- Acceder desde cualquier componente a la información del usuario autenticado sin prop drilling.
- Usar una estructura centralizada de acceso a servicios, típicamente usada en aplicaciones desacopladas.

```
26        const { role } = useAuth();
27        const { correo } = useAuth();
28        const { nombre } = useAuth();
```

Esto refleja una implementación práctica, aunque no explícita, del patrón **Service Locator**.

📷 **Imagen 1: Captura de context.jsx**

```
1   import { createContext, useState, useContext } from "react";
2
3   const AuthContext = createContext();
4
5   export function AuthProvider({ children }) {
6     const [role, setRole] = useState(null);
7     const [correo, setCorreo] = useState(null);
8     const [nombre, setNombre] = useState(null);
9
10
11    return (
12      <AuthContext.Provider
13        value={{ role, setRole, correo, setCorreo, nombre, setNombre }}
14      >
15        {children}
16      </AuthContext.Provider>
17    );
18  }
19
20  // Hook personalizado para usarlo fácilmente
21  export function useAuth() {
22    return useContext(AuthContext);
23  }
```

📷 **Imagen 2: Captura de uso en Header.jsx**

```
15        const permissions = rolePermissions[role]?.canAcces || [];
16
17        return (
18            <header className="report-header go-front">
19                <div className="header-buttons">
20                    { permissions.includes("statitics") && (
21                        <Link to="/statistics">
22                            <button className="header-btn"
23                            disabled={view === "statistics"}
24                            >See the statistics</button>
25                        </Link>
26                    )}
27                    { permissions.includes("report") && (
28                        <Link to="/report">
29                            <button className="header-btn"
30                            disabled={view === "report"}
31                            >Make a report</button>
32                        </Link>
33                    )}
34                    { permissions.includes("map") && (
35                        <Link to = "/map">
36                            <button className="header-btn"
37                            disabled={view === "map"}
38                            >See the map</button>
39                        </Link>
40                    )}
```

🌐 5. Patrón de diseño propuesto para la migración web

✅ **Patrón elegido: Observer (Observador)** 👁️

📌 6. Definición y propósito del patrón Observer

El patrón **Observer** es un patrón de diseño de comportamiento que permite que un objeto ("sujeto") notifique automáticamente a múltiples objetos dependientes ("observadores") sobre cualquier cambio en su estado, sin que exista acoplamiento entre ellos.

Se utiliza cuando un cambio en un objeto requiere que otros objetos se actualicen, y no se desea acoplar directamente esas clases.

🧠 7. ¿Por qué Observer es el más adecuado para CACVi-UN?

Cuando la aplicación se migre a una versión web moderna, el sistema tendrá nuevas necesidades como:

- ✅ Actualización en **tiempo real del mapa** cuando se registre un nuevo reporte
- ✅ Visualización dinámica de estadísticas al acumularse eventos
- ✅ Notificación a administradores en línea sobre nuevas denuncias
- ✅ Integración futura con sistemas de chat, alertas o dashboards

Esto requiere que múltiples componentes se **sincronicen entre sí sin acoplarse directamente**, lo cual es exactamente lo que facilita el patrón **Observer**.

Imagen 3: Captura de Map.jsx

```
<div style={{ height: "400px", width: "100%" }}>
  <MapContainer
    center={[4.638193, -74.084046]} // Centro: UNAL Bogotá
    zoom={17} // Zoom inicial
    minZoom={16} // Zoom mínimo (no puede alejar más)
    maxZoom={18} // Zoom máximo (acercar más)
    maxBounds={[
      [4.6315, -74.0935], // suroeste (más abajo y más a la izquierda)
      [4.6445, -74.069], // noreste (más arriba y más a la derecha)
    ]}
    maxBoundsViscosity={1.0} // Impide salir del área
    style={{ height: "100%", width: "100%" }}
  >
    <TileLayer
      attribution='&copy; <a href="http://osm.org/copyright">OpenStreetMap</a>'
      url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"
    />
    <Marker position={[4.638193, -74.084046]}>
      <Popup>Universidad Nacional de Colombia - Sede Bogotá</Popup>
    </Marker>
  </MapContainer>
</div>
```

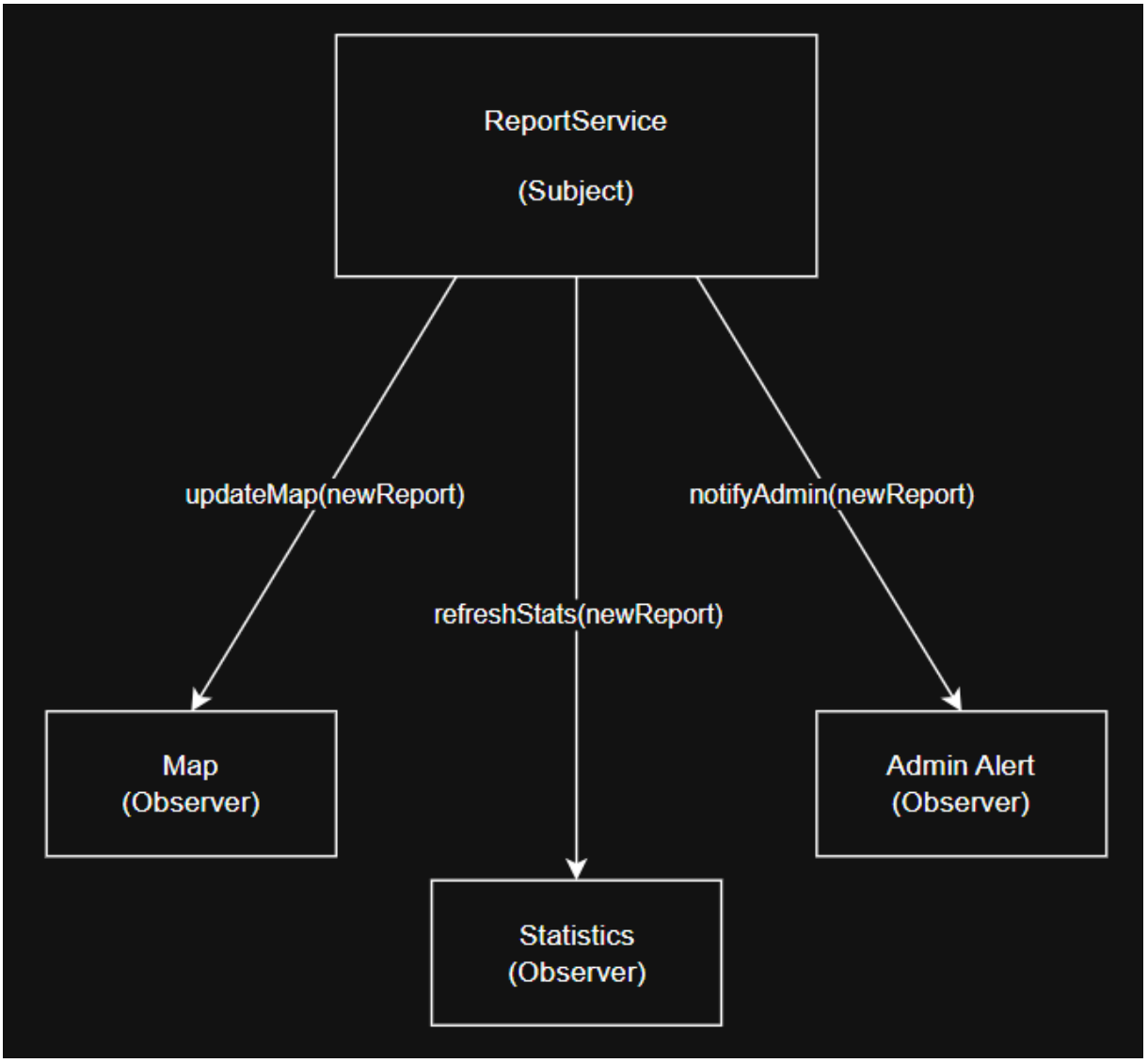
8. ¿Cómo se aplicaría Observer en CACVi-UN?

Ejemplo conceptual:

- Un módulo central actúa como el **sujeto** (ReportService o WebSocketListener)
- Los componentes como Map, Statistics, AdminPanel se registran como **observadores**
- Cuando entra un nuevo reporte, el ReportService notifica a todos los observadores para que se actualicen

```
// Pseudocódigo conceptual
reportService.onNewReport((data) => {
  mapComponent.update(data);
  statisticsComponent.refresh();
  adminAlert.show(data);
});
```

Imagen 4: Diagrama del patrón Observer aplicado al sistema



🔄 9. ¿Por qué no se eligieron otros patrones?

Patrón	¿Por qué no fue elegido?
Singleton	React ya gestiona el estado global con contextos.
Factory	No hay múltiples vistas complejas por tipo de usuario aún.
Strategy	Los roles son fijos y la lógica de acceso es estática.
MVC/MVVM	React ya separa presentación, estado y lógica.
Mediator	No hay interacciones cruzadas complejas entre muchos componentes aún.

🎯 10. Conclusión

El proyecto **CACVi-UN**, en su fase actual, **no requiere implementar un patrón de diseño formal** debido a su simplicidad, bajo acoplamiento y arquitectura monolítica orientada al cliente.

Sin embargo, se ha estructurado correctamente para **facilitar la implementación futura** de un patrón como **Observer**, que será clave en una arquitectura web con vistas sincronizadas.

notificaciones en vivo y componentes independientes.

El uso parcial del patrón **Service Locator** mediante useContext demuestra que el sistema ya tiene bases claras de modularidad y escalabilidad.