Universidad Nacional de Colombia - sede Bogotá
Facultad de Ingeniería
Departamento de Sistemas e Industrial
Curso: Ingeniería de Software 1 (2016701)
Estudiantes: David Ladino, Jorge Cuadrado, Juan Camilo D'Aleman Rodriguez, Smith Yañez.

Patrón Observer - Colombia Raíces

Definición y Propósito

El Patrón Observer es un patrón de diseño de comportamiento que define una dependencia uno-a-muchos entre objetos, donde cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente.

Su propósito principal es:

- Establecer comunicación desacoplada entre componentes.
- Permitir que múltiples objetos reaccionen a cambios de estado.
- Facilitar la extensibilidad sin modificar código existente.
- Implementar un sistema de eventos distribuidos.

Implementación En El Proyecto

El proyecto implementa doble Observer Pattern para diferentes contextos:

1. EventObserver - Eventos Generales del Sistema

Ubicación: main/utils/observer.js

```
class EventObserver {
  constructor() {
    if (EventObserver.instance) {
      return EventObserver.instance;
    }
    this.observers = new Map();
    EventObserver.instance = this;
  }

  subscribe(event, callback) {
    if (!this.observers.has(event)) {
      this.observers.set(event, []);
    }
    this.observers.get(event).push(callback);

    // Retorna función de cleanup
    return () => this.unsubscribe(event, callback);
  }

  notify(event, data) {
    if (this.observers.has(event)) {
      this.observers.get(event).forEach(callback => {
        try {
          callback(data);
        } catch (error) {
          console.error(`Error in observer for event ${event}:`, error);
        }
      });
    }
  }
```

```
  unsubscribe(event, callback) {
    if (this.observers.has(event)) {
      const callbacks = this.observers.get(event);
      const index = callbacks.indexOf(callback);
      if (index > -1) {
        callbacks.splice(index, 1);
      }
    }
  }
}

// Eventos definidos
const APP_EVENTS = {
  USER_LOGIN: "user:login",
  USER_LOGOUT: "user:logout",
  USER_REGISTER: "user:register",
  EXPERIENCE_CREATED: "experience:created",
  EXPERIENCE_UPDATED: "experience:updated",
  RESERVATION_CREATED: "reservation:created",
  DATABASE_CONNECTED: "database:connected",
  APP_READY: "app:ready",
  WINDOW_CLOSED: "window:closed"
};

module.exports = { EventObserver: new EventObserver(), APP_EVENTS };
```

2. AuthObserver - Eventos de Autenticación

Ubicación: main/utils/AuthObserver.js

```
class AuthObserver {
  constructor() {
    this.observers = new Map();
  }

  subscribe(eventType, callback, observerId = null) {
    if (!this.observers.has(eventType)) {
      this.observers.set(eventType, new Map());
    }

    const id = observerId || this.generateObserverId();
    this.observers.get(eventType).set(id, callback);

    return id;
  }

  notify(eventType, data) {
    if (this.observers.has(eventType)) {
      const eventObservers = this.observers.get(eventType);
      eventObservers.forEach((callback, observerId) => {
        try {
          callback(data);
        } catch (error) {
          console.error(`Error en observer ${observerId}:`, error);
        }
      });
    }
  }
```

```
  unsubscribe(eventType, observerId) {
    if (this.observers.has(eventType)) {
      this.observers.get(eventType).delete(observerId);
    }
  }

  generateObserverId() {
    return 'observer_' + Date.now() + '_' + Math.random().toString(36).substr(2, 9);
  }
}

// Eventos de autenticación
const AUTH_EVENTS = {
  USER_LOGIN: "user_login",
  USER_LOGOUT: "user_logout",
  USER_REGISTER: "user_register",
  PASSWORD_CHANGE: "password_change",
  LOGIN_FAILED: "login_failed",
  SESSION_EXPIRED: "session_expired"
};

module.exports = { AuthObserver, AUTH_EVENTS };
```

Uso En El Código Fuente

Implementación en AuthController

Ubicación: main/controllers/AuthController.js

```
const { AuthObserver, AUTH_EVENTS } = require('../utils/AuthObserver');

class AuthController {
  constructor() {
    this.authService = new AuthService();
    this.authObserver = new AuthObserver();
    this.userModel = new UserModel();
  }

  async login(email, password) {
    try {
      const user = await this.userModel.authenticate(email, password);

      // Notificar login exitoso
      this.authObserver.notify(AUTH_EVENTS.USER_LOGIN, {
        userId: user.id,
        email: user.email,
        name: user.name,
        userType: user.user_type,
        timestamp: new Date()
      });

      return { success: true, user, token };
    } catch (error) {
      // Notificar login fallido
      this.authObserver.notify(AUTH_EVENTS.LOGIN_FAILED, {
        email,
        error: error.message,
        timestamp: new Date()
      });
      throw error;
    }
  }
```

```
  async register(userData) {
    try {
      const newUser = await this.userModel.createUser(userData);

      // Notificar registro exitoso
      this.authObserver.notify(AUTH_EVENTS.USER_REGISTER, {
        userId: newUser.id,
        email: newUser.email,
        name: newUser.name,
        userType: newUser.user_type,
        timestamp: new Date()
      });

      return { success: true, user: newUser };
    } catch (error) {
      throw error;
    }
  }
}
```

Configuración de Listeners

Ubicación: main/utils/eventSetup.js

```
const { EventObserver, APP_EVENTS } = require('./observer');
const { AuthObserver, AUTH_EVENTS } = require('./AuthObserver');

function setupEventListeners() {
  // Listeners para eventos generales
  EventObserver.subscribe(APP_EVENTS.USER_LOGIN, (data) => {
    console.log(`Usuario ${data.email} ha iniciado sesión`);
  });

  EventObserver.subscribe(APP_EVENTS.EXPERIENCE_CREATED, (data) => {
    console.log(`Nueva experiencia creada: ${data.title}`);
  });

  EventObserver.subscribe(APP_EVENTS.DATABASE_ERROR, (error) => {
    console.error('Error en la base de datos:', error);
  });

  // Listeners para eventos de autenticación
  const authObserver = new AuthObserver();

  authObserver.subscribe(AUTH_EVENTS.USER_LOGIN, (data) => {
    console.log(`Login exitoso para: ${data.email}`);
  });

  authObserver.subscribe(AUTH_EVENTS.LOGIN_FAILED, (data) => {
    console.log(`Login fallido para: ${data.email} - ${data.error}`);
  });
}

module.exports = { setupEventListeners };
```

Uso en Electron Main Process

Archivo: main/electron.js

```
const { EventObserver, APP_EVENTS } = require('./utils/observer');

app.whenReady().then(async () => {
  try {
    await initializeModels();

    // Notificar que la app está lista
    EventObserver.notify(APP_EVENTS.APP_READY);

    createWindow();
  } catch (error) {
    // Notificar error de base de datos
    EventObserver.notify(APP_EVENTS.DATABASE_ERROR, error);
  }
});

app.on('window-all-closed', () => {
  // Notificar cierre de ventanas
  EventObserver.notify(APP_EVENTS.WINDOW_CLOSED);

  if (process.platform !== 'darwin') {
    app.quit();
  }
});
```

Justificación De Uso

El Patrón Observer fue implementado estratégicamente en Colombia Raíces para crear un sistema de eventos robusto y escalable. Su uso dual (EventObserver para eventos generales y AuthObserver para autenticación) proporciona:

- Comunicación desacoplada entre módulos del sistema de turismo.
- Escalabilidad para agregar nuevas funcionalidades sin modificar código existente.
- Robustez en el manejo de errores y eventos concurrentes.
- Mantenibilidad a través de una arquitectura clara y bien estructurada.

Esta implementación es fundamental para el funcionamiento eficiente del sistema de gestión de turismo rural, permitiendo que componentes como reservas, experiencias, autenticación y notificaciones trabajen de manera coordinada pero independiente.