ARQUITECTURA DE SOFTWARE

Unidad III – Diseño de Patrones

Actividad 1 - Saber

Alumno:

Alan Manuel Mendoza Arredondo –1221100341

GIDS4071

Contenido

Introducción

En el panorama actual del desarrollo de software, la construcción de aplicaciones complejas y robustas demanda enfoques arquitectónicos y de diseño que promuevan la modularidad, la reutilización y la fácil mantenibilidad del código. En este contexto, los patrones de diseño emergen como herramientas fundamentales que ofrecen soluciones probadas a desafíos comunes.

En el campo del desarrollo de software, los "patrones emergentes" se refieren a soluciones recurrentes a problemas comunes que evolucionan y se consolidan con el tiempo. Estos patrones representan enfoques efectivos para diseñar y estructurar sistemas de software, proporcionando a los desarrolladores soluciones probadas y buenas prácticas para enfrentar desafíos específicos.

En este documento, exploraremos varios patrones emergentes, también conocidos como patrones de segunda generación, que se han convertido en piedras angulares para el desarrollo de aplicaciones modernas. Estos patrones no solo proporcionan soluciones a problemas específicos, sino que también abordan la creciente complejidad de las aplicaciones contemporáneas.

Cada patrón será examinado en detalle, resaltando sus características fundamentales, ventajas y desventajas. Además, se presentará una implementación práctica de estos patrones en el contexto de una aplicación Angular con backend PHP y MySQL.

A lo largo del documento, nos sumergiremos en los siguientes patrones emergentes:

Modelo-Vista-Controlador (MVC): Separación clara de responsabilidades en Modelo, Vista y Controlador para facilitar la escalabilidad y el mantenimiento.

Objeto de Acceso a Datos (DAO): Gestión eficiente de la capa de acceso a datos mediante una interfaz uniforme, permitiendo flexibilidad y reutilización del código.

Command Query Responsibility Segregation (CQRS): Separación de operaciones de lectura y escritura para optimizar el rendimiento y escalabilidad de la aplicación.

Diseño Dirigido por Dominio (DDD): Enfoque que busca alinear el desarrollo de software con la lógica del dominio empresarial, facilitando la comprensión y la comunicación.

Modelo-Vista-VistaModelo (MVVM): Arquitectura especialmente útil en el desarrollo de aplicaciones de interfaz de usuario, que separa Modelo, Vista y VistaModelo.

Modelo-Vista-Presentador (MVP): Variante del patrón MVC que mejora la prueba unitaria de la lógica de presentación.

Finalmente, analizaremos la aplicación práctica de dos de estos patrones, MVC y DAO, en una aplicación Angular específica. Este análisis incluirá una descripción detallada de la implementación y su justificación en el contexto del desarrollo de software.

Modelo-Vista-Controlador (MVC)

El patrón Modelo-Vista-Controlador (MVC) es un paradigma arquitectónico ampliamente utilizado en el desarrollo de software. Su diseño se basa en la separación de responsabilidades, dividiendo la aplicación en tres componentes principales:

- **Modelo**: Representa la estructura de datos y la lógica de negocio de la aplicación. Es responsable de gestionar la información y responder a las consultas y comandos de la Vista y el Controlador.
- **Vista**: Presenta la información al usuario de manera visual y se encarga de la interfaz gráfica. La Vista recibe actualizaciones del Modelo y muestra los datos de manera adecuada para la comprensión del usuario.
- **Controlador**: Gestiona las interacciones del usuario y actúa como intermediario entre la Vista y el Modelo. Se encarga de recibir las acciones del usuario, actualizar el Modelo en consecuencia y reflejar los cambios en la Vista.

Características Principales

- **Separación de preocupaciones**: El MVC permite dividir las preocupaciones relacionadas con la presentación, la lógica de negocio y el manejo de eventos en componentes independientes, lo que facilita el mantenimiento y la escalabilidad.
- **Reutilización de componentes**: La separación entre Modelo y Vista permite la reutilización de estas componentes en diferentes partes de la aplicación o incluso en otros proyectos.
- **Facilita el desarrollo colaborativo**: Al tener roles claramente definidos, varios desarrolladores pueden trabajar en paralelo en diferentes aspectos de la aplicación.

Ventajas y Desventajas

- **Ventajas**:
  - Modularidad y escalabilidad.
  - Facilita la prueba unitaria de componentes.
  - Favorece el desarrollo paralelo.
- **Desventajas**
  - Puede resultar complejo para aplicaciones pequeñas.
  - Aumenta la complejidad inicial del diseño.

Aplicaciones Prácticas

- MVC se utiliza extensamente en el desarrollo web, destacando en frameworks como Ruby on Rails y ASP.NET MVC.
- Presente en aplicaciones de escritorio y móviles que buscan una clara separación de la lógica de negocios y la interfaz de usuario.

Diagrama UML de MVC

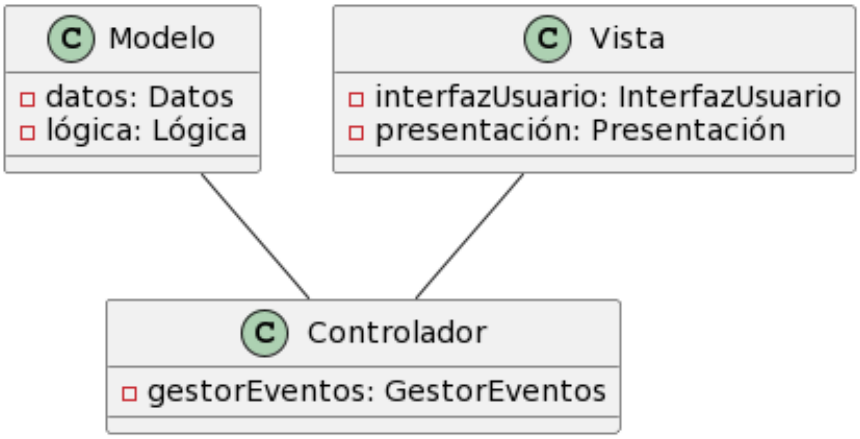

Ilustración 1 Diagrama UML de MVC

Objeto de Acceso a Datos (DAO)

El patrón DAO (Objeto de Acceso a Datos) aborda la gestión de la capa de acceso a datos en una aplicación. Su objetivo es separar la lógica de negocio de las operaciones de acceso a la base de datos, proporcionando una interfaz uniforme para interactuar con diferentes tipos de fuentes de datos.

Características Principales

- **Desacoplamiento del código de acceso a datos**: DAO permite aislar las operaciones de acceso a datos, lo que facilita la modificación de la fuente de datos sin afectar la lógica empresarial.
- **Flexibilidad en la gestión de datos**: Facilita la implementación de operaciones CRUD (Crear, Leer, Actualizar, Eliminar) de manera coherente, independientemente de la fuente de datos subyacente.
- **Reutilización del código de acceso a datos**: Las operaciones de acceso a datos definidas por el DAO pueden reutilizarse en diferentes partes de la aplicación.

Ventajas y Desventajas

- **Ventajas**:
  - Mayor flexibilidad en la gestión de datos.
  - Facilita la reutilización del código de acceso a datos.
- **Desventajas**:
  - Aumenta la complejidad en aplicaciones pequeñas.
  - Requiere un diseño cuidadoso para evitar redundancias.

Aplicaciones Prácticas

- DAO se utiliza comúnmente en aplicaciones empresariales que interactúan con bases de datos relacionales, NoSQL y otros sistemas de almacenamiento.
- Es esencial en sistemas que necesitan gestionar múltiples fuentes de datos o que deben cambiar la fuente de datos sin afectar otras capas de la aplicación.

Diagrama UML de DAO

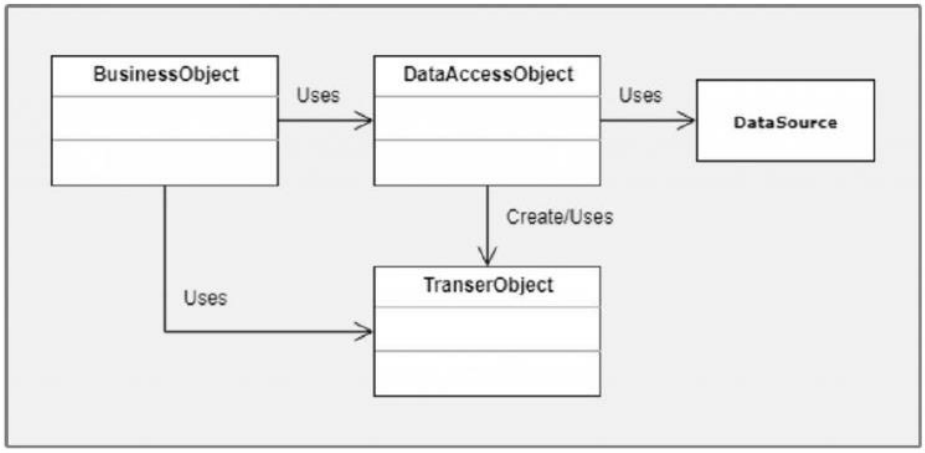

Ilustración 2 Diagrama de DAO

Command Query Responsibility Segregation (CQRS)

El patrón CQRS se centra en la separación de las responsabilidades de lectura y escritura en una aplicación. Divide las operaciones que modifican el estado (comandos) de aquellas que consultan el estado (consultas), utilizando modelos distintos para cada una.

Características Principales

- **Separación de lectura y escritura**: CQRS divide la lógica de la aplicación en dos modelos separados, uno para operaciones de escritura y otro para operaciones de lectura, optimizando el rendimiento y la escalabilidad.
- **Modelos específicos**: Utiliza modelos específicos para las operaciones de lectura y escritura, permitiendo optimizar cada uno para su tarea específica.
- **Escalabilidad**: Permite escalar las operaciones de lectura y escritura de forma independiente, adaptándose mejor a sistemas con requisitos de rendimiento asimétricos.

Ventajas y Desventajas

- **Ventajas**:
  - Mejora el rendimiento al permitir la optimización de modelos para tareas específicas.
  - Facilita la escalabilidad al separar las operaciones de lectura y escritura.
  - Mayor flexibilidad para evolucionar los modelos independientemente.
- **Desventajas**:
  - Introduce complejidad al mantener modelos separados.
  - Puede ser excesivo para aplicaciones simples.

Aplicaciones Prácticas

- CQRS es útil en sistemas donde las operaciones de lectura superan significativamente a las operaciones de escritura.

- Se implementa comúnmente en sistemas de análisis y reporting, así como en aplicaciones con requisitos de rendimiento específicos.

Diagrama UML de CQRS

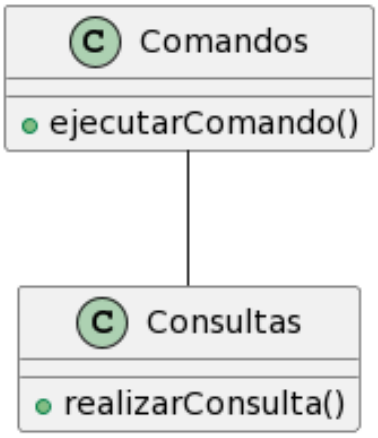

Ilustración 3 Diagrama de CQRS

Diseño Dirigido por Dominio (DDD)

El Diseño Dirigido por Dominio (DDD) es un enfoque que busca alinear el desarrollo de software con el lenguaje y la lógica del dominio de negocio. Propuesto por Eric Evans, DDD se centra en comprender y modelar el dominio empresarial de manera profunda.

Características Principales

- **Ubicuidad del lenguaje**: DDD busca utilizar un lenguaje común entre los desarrolladores y los expertos del dominio, facilitando la comprensión y la comunicación.
- **Modelado del dominio**: Se enfoca en construir un modelo del dominio que refleje con precisión la realidad del negocio.
- **Bounded Contexts** (Contextos Delimitados): Define límites claros entre las distintas partes del sistema para evitar confusiones y conflictos en la interpretación del modelo.

Ventajas y Desventajas

- Ventajas:
  - Mejora la comprensión del negocio y su implementación en software.
  - Facilita la comunicación entre desarrolladores y expertos del dominio.
  - Proporciona un enfoque para gestionar la complejidad del software.
- Desventajas:
  - Puede requerir un cambio cultural en el equipo de desarrollo.
  - La implementación completa de DDD puede ser costosa en términos de tiempo y recursos.

Aplicaciones Prácticas

- DDD se utiliza en proyectos donde la comprensión profunda del dominio es crucial.

- Es beneficioso en sistemas empresariales complejos, sistemas de gestión y en aquellos donde la lógica del negocio es central.

Diagrama UML de DDD

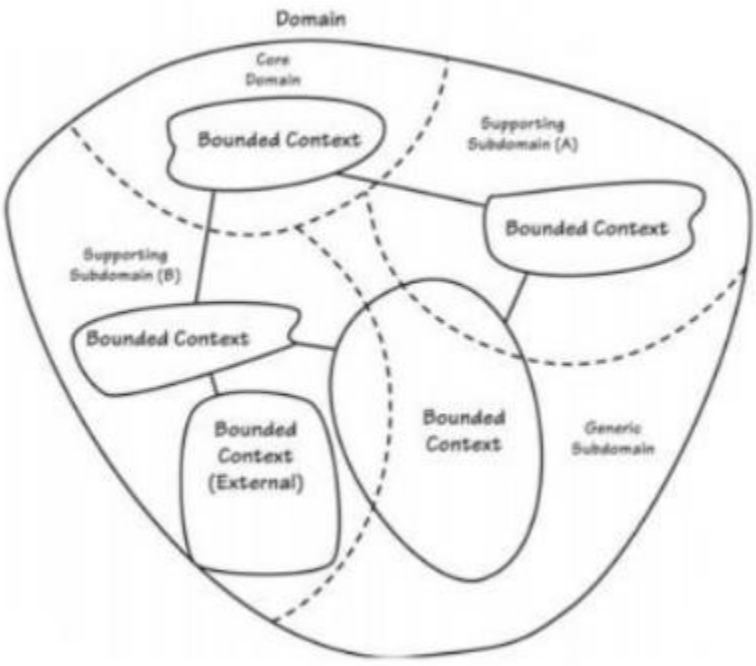

Ilustración 4 Diagrama de DDD

Modelo-Vista-VistaModelo (MVVM)

El patrón Modelo-Vista-VistaModelo (MVVM) es un enfoque arquitectónico que se utiliza comúnmente en el desarrollo de aplicaciones de interfaz de usuario, especialmente en entornos como WPF y Xamarin.

Características Principales

- **Modelo**: Representa la lógica de negocio y los datos de la aplicación.
- **Vista**: Presenta la interfaz de usuario y se encarga de la interacción del usuario.
- **VistaModelo**: Actúa como intermediario entre el Modelo y la Vista, gestionando la lógica de presentación y las interacciones del usuario.

Ventajas y Desventajas

- **Ventajas**:
  - Separación clara de responsabilidades.
  - Facilita el desarrollo paralelo entre diseñadores y desarrolladores.
  - Favorece la reutilización de componentes.
- **Desventajas**:
  - Puede resultar complejo en aplicaciones pequeñas.
  - Requiere un aprendizaje inicial para entender y aplicar correctamente el patrón.

Aplicaciones Prácticas

- MVVM es ampliamente utilizado en el desarrollo de aplicaciones de interfaz de usuario en plataformas como WPF, Xamarin y Angular.
- Beneficioso en entornos donde se requiere una clara separación de la lógica de presentación y la lógica de negocio.

Diagrama UML de MVVM

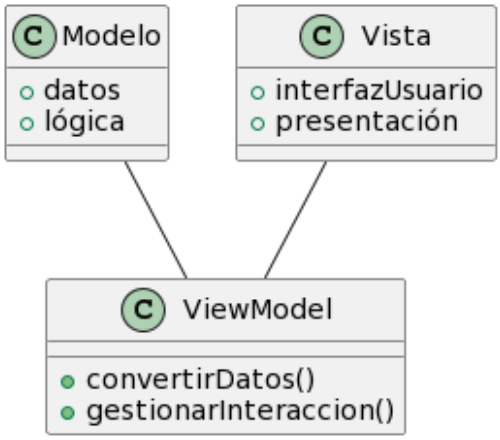

Ilustración 5 Diagrama de MVVM

Modelo-Vista-Presentador (MVP)

El patrón Modelo-Vista-Presentador (MVP) es una variante del patrón MVC que se centra en mejorar la prueba unitaria de la lógica de presentación.

Características Principales

- **Modelo**: Representa la lógica de negocio y los datos de la aplicación.
- **Vista**: Encargada de la presentación y la interfaz de usuario.
- **Presentador**: Actúa como intermediario entre el Modelo y la Vista, gestionando la lógica de presentación y las interacciones del usuario.

Ventajas y Desventajas

- **Ventajas**:
  - Facilita la prueba unitaria al separar la lógica de presentación del código de la interfaz de usuario.
  - Permite el desarrollo paralelo entre diseñadores y desarrolladores.
  - Mejora la mantenibilidad al desacoplar la lógica de presentación de la Vista.
- **Desventajas**:
  - Aumenta la complejidad en comparación con enfoques más simples como MVP.
  - Puede ser excesivo para aplicaciones pequeñas.

Aplicaciones Prácticas

- MVP se utiliza en entornos donde la prueba unitaria de la lógica de presentación es una prioridad.

- Beneficioso en aplicaciones web y móviles que requieren una clara separación de la lógica de presentación y la lógica de negocio.

Diagrama UML de MVP

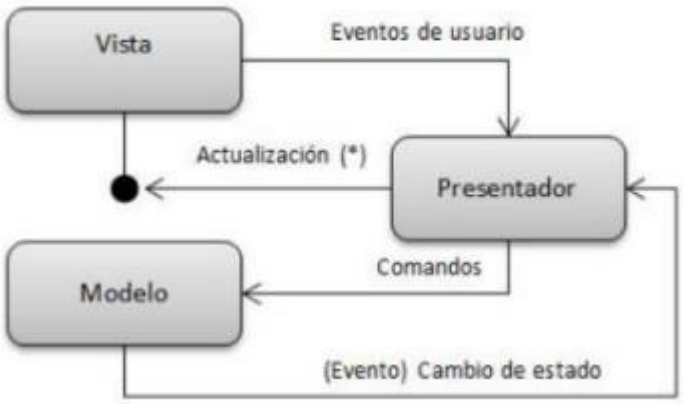

Ilustración 6 Diagrama UML de MVP

Implementación de un patrón emergente

En el desarrollo de la aplicación Angular con backend PHP y MySQL, la elección de los patrones de diseño es crucial para garantizar una estructura modular, mantenible y escalable. Se seleccionaron el patrón Modelo-Vista-Controlador (MVC) y el Objeto de Acceso a Datos (DAO) debido a sus beneficios en la separación de preocupaciones y la gestión eficiente de la capa de datos.

Implementación del patrón MVC:

Modelo (Model)

En la carpeta models, se ha implementado el modelo de datos utilizando TypeScript. Por ejemplo, el archivo models/Proyecto.model.ts define la estructura del modelo de proyecto. Este enfoque facilita la encapsulación de la lógica de datos y la representación coherente de las entidades en la aplicación.

Ilustración 7 Implementación del modelo

Vista (View)

Los componentes Angular, como dashboard.component.html, actúa como vista. Estos componentes están diseñados para presentar información al usuario y gestionar las interacciones. La estructura de la vista se separa claramente del controlador y el modelo, siguiendo los principios del patrón MVC.

Ilustración 8 Implementación de la vista

Controlador (Controller)

El controlador se manifiesta en los componentes de Angular que gestionan la lógica del negocio y coordinan las interacciones entre la vista y el modelo. Por ejemplo, el archivo dashboard.component.ts se encarga de cargar los proyectos desde el servicio y gestionar la lógica relacionada con el dashboard.

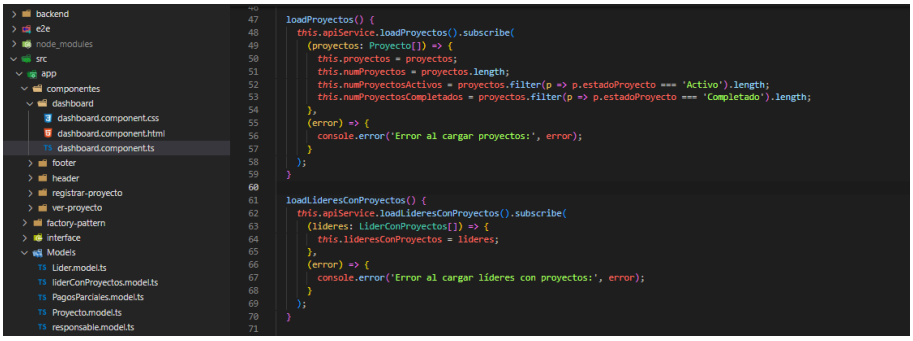

Ilustración 9 Implementación del controlador

Diagrama UML de MVC

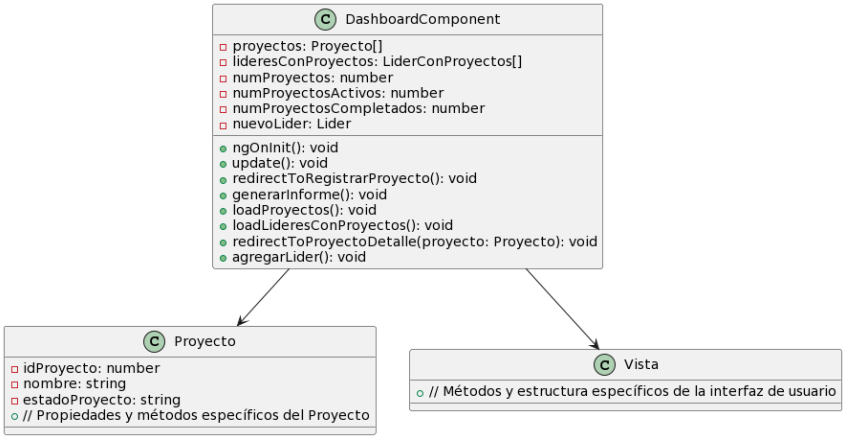

Ilustración 10 Diagrama UML de MVC implementado

Aplicación del Patrón DAO

Acceso a Datos (DAO)

El servicio api.service.ts actúa como una capa DAO que maneja las operaciones de acceso a datos desde el backend PHP y MySQL. Aquí se centralizan las operaciones relacionadas con los proyectos, proporcionando una interfaz para interactuar con la capa de datos.

```
@Injectable({
  providedIn: 'root',
})
export class ApiService {
  private apiUrl = 'http://localhost:8080';

  constructor(private http: HttpClient, private observableService: ObservableService) { }

  // Método para realizar una solicitud GET a una API en el backend.
  public get(endpoint: string): Observable<any> {
    const url = `${this.apiUrl}/${endpoint}`;
    return this.http.get(url);
  }

  // Método para realizar una solicitud POST a una API en el backend.
  public post(endpoint: string, data: any): Observable<any> {
    const url = `${this.apiUrl}/${endpoint}`;
    const headers = new HttpHeaders({ 'Content-Type': 'application/json' });
    return this.http.post(url, data, { headers });
  }

  // Cargar stakeholders
  loadStakeholders(): Observable<Stakeholder[]> {
    const url = `${this.apiUrl}/api/loadStakeholders.php`;
    const headers = new HttpHeaders({ 'Content-Type': 'application/json' });

    return this.http.get<Stakeholder[]>(url, { headers });
  }

  // Cargar responsables
  loadResponsables(): Observable<Responsable[]> {
    const url = `${this.apiUrl}/api/loadResponsables.php`;
    const headers = new HttpHeaders({ 'Content-Type': 'application/json' });

    return this.http.get<Responsable[]>(url, { headers });
  }
  // Registrar proyecto
  registrarProyecto(proyecto: any): Observable<any> {
    const url = `${this.apiUrl}/api/registrarProyecto.php`;
    const headers = new HttpHeaders({ 'Content-Type': 'application/json' });

    return this.http.post(url, proyecto, { headers })
      .pipe(tap(() => this.observableService.notifyProjectUpdate()));
  }
}
```

Ilustración 11 Implementación del patrón DAO

Justificación de Implementación del Objeto de Acceso a Datos (DAO) en api.service.ts

La implementación del Objeto de Acceso a Datos (DAO) en el servicio api.service.ts se justifica por varios principios y beneficios fundamentales en el diseño de software. A continuación, se detallan las razones para evidenciar la aplicación del patrón DAO:

Separación de Responsabilidades:

El patrón DAO permite separar las responsabilidades relacionadas con el acceso a datos de las capas de negocio y presentación. En api.service.ts, cada método se encarga de operaciones específicas de acceso a datos, como

cargar stakeholders, registrar proyectos, o obtener información detallada de un proyecto. Esto facilita el mantenimiento y la comprensión del código al tener una clara separación de responsabilidades.

Reutilización de Código:

Al encapsular las operaciones de acceso a datos en métodos específicos, se facilita la reutilización de código en diferentes partes de la aplicación. Los métodos como loadStakeholders(), loadProyectos(), y otros, pueden ser invocados desde distintos componentes y servicios sin duplicar la lógica de acceso a datos.

Abstracción de la Fuente de Datos:

La implementación del DAO permite abstractizar la fuente de datos subyacente. Los detalles específicos de la comunicación con la API (en este caso, HTTP) están encapsulados en el servicio, permitiendo cambios en la implementación de la API o en la fuente de datos sin afectar la lógica de la aplicación.

Facilita Pruebas Unitarias:

Al tener métodos bien definidos y encapsulados en el servicio, se facilita la escritura de pruebas unitarias para cada operación de acceso a datos. Esto mejora la calidad del código al permitir una cobertura de pruebas más exhaustiva.

Aplicación de Principios CQRS:

El patrón DAO es compatible con los principios de Command Query Responsibility Segregation (CQRS). Las operaciones de lectura (loadStakeholders(), loadProyectos()) están separadas de las operaciones de escritura (registrarProyecto(proyecto), registrarDocumento(folio, documento)), permitiendo una mejor optimización y escalabilidad.

La implementación del DAO en api.service.ts refleja un enfoque modular y bien estructurado para manejar las operaciones de acceso a datos en la aplicación, promoviendo la legibilidad, mantenibilidad y flexibilidad en el desarrollo continuo. Estos principios respaldan la decisión de utilizar el patrón DAO en el diseño del servicio.

Conclusiones

En el dinámico panorama del desarrollo de software, la adopción de patrones emergentes se presenta como un imperativo para enfrentar los desafíos de construir aplicaciones complejas y robustas. Este documento ha explorado varios patrones de segunda generación que han evolucionado como piedras angulares en el desarrollo de aplicaciones modernas, abordando no solo problemas específicos, sino también la creciente complejidad de las aplicaciones contemporáneas.

Cada patrón examinado, desde el Modelo-Vista-Controlador (MVC) hasta el Diseño Dirigido por Dominio (DDD), ofrece soluciones únicas y ventajas particulares. La implementación práctica de dos de estos patrones, MVC y DAO, en una aplicación Angular con backend PHP y MySQL, ha proporcionado un contexto concreto para comprender cómo estos patrones se traducen en estructuras modulares, mantenibles y escalables.

En la aplicación del patrón MVC, se destacó la claridad en la separación de responsabilidades, la reutilización de componentes y la facilidad de desarrollo colaborativo. Por otro lado, la aplicación del patrón DAO en el servicio api.service.ts reveló una clara separación de responsabilidades en las operaciones de acceso a datos, favoreciendo la reutilización del código y abstrayendo la fuente de datos subyacente.

La elección de patrones de diseño no solo impacta la arquitectura del software, sino que también influye en aspectos clave como la mantenibilidad, la escalabilidad y la eficiencia del desarrollo. En resumen, los patrones emergentes no solo ofrecen soluciones a problemas técnicos, sino que también proporcionan un marco conceptual compartido, mejorando la comunicación y la comprensión mutua en equipos de desarrollo.

Al entender y aplicar estos patrones, los desarrolladores están mejor equipados para construir sistemas más robustos, adaptativos y alineados con las necesidades cambiantes del desarrollo de software. La elección y aplicación adecuada de estos patrones emergentes depende del contexto específico de cada proyecto, pero su conocimiento y comprensión son esenciales para abordar los desafíos del desarrollo de software en la era moderna.

En el análisis detallado de cada patrón emergente, se observaron las características clave, ventajas y desventajas, así como aplicaciones prácticas. Estos patrones no solo son herramientas técnicas, sino también conceptos arquitectónicos que influyen en la forma en que los equipos diseñan, construyen y mantienen software.

La implementación práctica de MVC y DAO en una aplicación Angular con backend PHP y MySQL ilustró cómo estos patrones se traducen en una estructura modular y eficiente. El patrón MVC permitió una clara separación de responsabilidades, mientras que el patrón DAO facilitó la gestión eficiente de la capa de datos.

En conclusión, la comprensión y aplicación de patrones emergentes en el desarrollo de software no solo optimizan la estructura técnica de las aplicaciones, sino que también impactan la colaboración entre equipos, la mantenibilidad y la capacidad de adaptación a cambios futuros. Estos patrones, con sus ventajas y desventajas, representan valiosas herramientas en el arsenal de cualquier desarrollador moderno. A medida que el panorama tecnológico evoluciona, la capacidad de aplicar y adaptar estos patrones se convierte en un diferenciador clave en la creación de software exitoso y sostenible.

Bibliografía

- Herrera, J. (2023, diciembre 8). Patrón de diseño MVC. ¿Qué es y cómo puedo utilizarlo? Easy App CODE. https://www.easyappcode.com/patron-de-diseno-mvc-que-es-y-como-puedo-utilizarlo

- (S/f). Studocu.com. Recuperado el 8 de diciembre de 2023, de https://www.studocu.com/es-mx/document/universidad-tecnologica-de-guadalajara/programacion-iii/patrones-de-diseno-emergentes/13116831