DESARROLLAR LA ARQUITECTURA DE SOFTWARE DE ACUERDO AL PATRÓN DE DISEÑO SELECCIONADO
GA4-220501095-AA2-EV05.
FICHA: 2879694

Presentado por:
OLGA SOFIA GALVIS MIRANDA

TECNOLOGIA EN ANALISIS Y DESARROLLO DEL SOFTWARE
BARRANCABERMEJA
SENA 2025

CONTENIDO

Se entiende por arquitectura de software el conjunto de patrones y abstracciones coherentes que proporcionan un marco definido y claro para interactuar con el código fuente.

Desarrollar la arquitectura de software teniendo en cuenta los siguientes criterios:

- Estudiar el componente formativo “Diseño de patrones de software”.
- Incorporar patrones de diseño propendiendo en mejores prácticas para la codificación y mantenibilidad del software.
- Elaborar la vista de componentes para visualizar el software en fases avanzadas del ciclo de vida.
- Elaborar la vista de despliegue del software para determinar condiciones de la implantación de la solución informática.
- Elegir herramientas necesarias para optimizar los procesos.

INTRODUCCIÓN

Para desarrollar la arquitectura de software teniendo en cuenta los criterios mencionados, a continuación, se presentan las consideraciones y pasos a seguir:

Análisis de requisitos: Comienza por comprender los requisitos del software, las funcionalidades necesarias y las restricciones del sistema. Esto permitirá establecer una base sólida para el diseño arquitectónico.

Selección de patrones de diseño: Identifica los patrones de diseño adecuados que ayuden a mejorar la codificación y mantenibilidad del software. Algunos patrones comunes incluyen el patrón MVC (Modelo-Vista-Controlador) para separar la lógica de presentación de los datos, el patrón de Inyección de Dependencias para facilitar el modularidad y el patrón Observar para manejar eventos y notificaciones.

Diseño de la arquitectura de componentes: Utiliza diagramas de componentes para visualizar el software en fases avanzadas del ciclo de vida. Identifica los componentes principales del sistema y las interacciones entre ellos. Asegúrate de definir interfaces claras y responsabilidades específicas para cada componente.

Diseño de la arquitectura de despliegue: Crea un diagrama de despliegue para determinar las condiciones de implantación de la solución informática. Identifica los nodos físicos (servidores, dispositivos, etc.) y las conexiones entre ellos. Esto te ayudará a comprender cómo se distribuirá el software y cómo se comunicarán los diferentes componentes.

Selección de herramientas: Elige las herramientas necesarias para optimizar los procesos de desarrollo y mantenimiento del software. Esto puede incluir herramientas de control de versiones como Git, entornos de desarrollo integrados (IDE) como Visual Studio o Eclipse, frameworks de desarrollo web como Django o Ruby on Rails, entre otros. Considera las necesidades específicas de tu proyecto y elige herramientas que se ajusten mejor a ellas.

Implementación y pruebas: Con base en la arquitectura diseñada, comienza la implementación del software siguiendo las mejores prácticas y utilizando los patrones de diseño seleccionados. A medida que avanzas en el desarrollo, realiza pruebas continuas para asegurarte de que el software cumpla con los requisitos y funcione correctamente.

Mantenimiento y evolución: Durante todo el ciclo de vida del software, mantén la arquitectura actualizada y evoluciona según sea necesario. Realiza mejoras y actualizaciones para abordar nuevas funcionalidades o cambios en los requisitos. Mantén un enfoque en la mantenibilidad y la escalabilidad del sistema.

Patrones Creacionales

Los patrones creacionales son un conjunto de patrones de diseño que se centran en la creación de objetos de manera flexible y eficiente. Estos patrones proporcionan diferentes mecanismos para la creación de objetos, ocultando los detalles específicos de su implementación y promoviendo la reutilización y la flexibilidad en el código. A continuación, se presentan algunos ejemplos de patrones creacionales Patrón de fábrica (Factory Pattern): Permite la creación de objetos sin especificar explícitamente la clase exacta del objeto que se creará. En lugar de ello, se utiliza

una interfaz común o una clase base para crear los objetos. Esto facilita la adición de nuevas implementaciones de la interfaz sin afectar el código existente.

Patrón de fábrica abstracta (Abstract Factory Pattern): Proporciona una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas. Permite la creación de objetos que están diseñados para trabajar juntos y asegura que los objetos creados sean compatibles entre sí.

Patrón de constructor (Builder Pattern): Se utiliza cuando se necesita construir objetos complejos paso a paso. Permite la creación de diferentes representaciones de un objeto utilizando el mismo proceso de construcción. También permite la creación de objetos inmutables.

Patrón de prototipo (Prototype Pattern): Proporciona una manera de crear nuevos objetos a partir de un objeto existente mediante la clonación. Este patrón evita la creación de subclases para crear objetos y permite la creación de nuevos objetos con un menor costo computacional.

Patrón de objeto Singleton (Singleton Pattern): Limita la creación de una clase a una única instancia y proporciona un punto de acceso global a esta instancia. Es útil cuando se requiere una única instancia de una clase en todo el sistema.

Patrones Estructurales

Los patrones estructurales son un conjunto de patrones de diseño que se centran en la composición de clases y objetos para formar estructuras más grandes y flexibles. Estos patrones ayudan a definir cómo las clases y los objetos se relacionan entre sí, permitiendo cambios en la composición de clases y objetos para formar estructuras más grandes y flexibles. Estos patrones ayudan a definir cómo las clases y los objetos se relacionan entre sí, permitiendo cambios en la estructura sin afectar a los componentes individuales. A continuación, se presentan algunos ejemplos de patrones estructurales:

Patrón de adaptador (Adapter Pattern): Permite que dos interfaces incompatibles trabajen juntas al actuar como un puente entre ellas. Se utiliza cuando se necesita que un objeto interactúe con otro objeto que tiene una interfaz diferente.

Patrón de puente (Bridge Pattern): Desacopla una abstracción de su implementación, lo que permite que ambas varíen independientemente. Se utiliza cuando se desea separar la abstracción de su implementación para que puedan cambiar por separado.

Patrón de decorador (Decorator Pattern): Permite agregar funcionalidad adicional a un objeto dinámicamente. Se envuelve el objeto original con uno o varios objetos decoradores para agregar comportamiento sin modificar la estructura subyacente.

Patrón de fachada (Facade Pattern): Proporciona una interfaz simplificada para un subsistema complejo. Se utiliza para ocultar la complejidad de un conjunto de clases o subsistemas detrás de una interfaz única y fácil de usar.

Patrón de composite (Composite Pattern): Permite tratar a un grupo de objetos de la misma manera que a un objeto individual. Se utiliza para crear estructuras jerárquicas donde los objetos individuales y los grupos de objetos se tratan de manera uniforme.

Patrón de proxy (Proxy Pattern): Proporciona un objeto sustituto que controla el acceso a otro objeto. Se utiliza para controlar el acceso a un objeto, agregar funcionalidad adicional o realizar tareas de optimización.

Patrones De comportamiento

Los patrones de comportamiento son un conjunto de patrones de diseño que se centran en la comunicación y la interacción entre objetos y clases. Estos patrones se utilizan para definir cómo los objetos colaboran entre sí para lograr un comportamiento deseado. A continuación, se presentan algunos ejemplos de patrones de comportamiento:

Patrón de observador (Observer Pattern): Define una relación uno a muchos entre objetos, de modo que cuando un objeto cambia su estado, todos los objetos dependientes son notificados y actualizados automáticamente.

Patrón de estrategia (Strategy Pattern): Permite definir una familia de algoritmos intercambiables, encapsulando cada algoritmo en un objeto separado. Esto permite que los algoritmos varíen independientemente de los clientes que los utilizan.

Patrón de plantilla (Template Pattern): Define el esqueleto de un algoritmo en una clase base, permitiendo que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura general.

Patrón de estado (State Pattern): Permite que un objeto altere su comportamiento cuando su estado interno cambia. Define diferentes clases de estado y permite que el objeto cambie de estado en tiempo de ejecución.

Patrón de mando (Command Pattern): Encapsula una solicitud como un objeto, permitiendo parametrizar a los clientes con diferentes solicitudes y manejar operaciones de forma desacoplada de los objetos que las invocan.

Patrón de iterador (Iterator Pattern): Proporciona una forma de acceder secuencialmente a los elementos de una colección sin exponer su representación subyacente. Esto permite recorrer una colección de objetos sin conocer su estructura interna.

Patrón de visitante (Visitor Pattern): Permite agregar operaciones adicionales a una estructura de objetos sin modificar sus clases. Define operaciones que visitan diferentes elementos de una estructura y permite agregar nuevas operaciones sin cambiar las clases de los elementos.

Vista De Componentes Para Poder Visualizar El Software

La vista de componentes es una forma de visualizar la estructura y organización de un sistema de software en términos de componentes y sus relaciones. Proporciona una representación gráfica de los componentes principales del sistema y cómo se conectan entre sí. Aquí hay algunos pasos para elaborar la vista de componentes:

Identificar componentes principales: Analiza el sistema de software y identifica los componentes principales que lo componen. Un componente puede ser una clase, un módulo, un paquete o incluso un servicio independiente.
Definir relaciones entre componentes: Determina cómo se relacionan entre sí los diferentes componentes. Esto puede incluir relaciones de dependencia, composición,

agregación o cualquier otro tipo de conexión necesaria para que el sistema funcione correctamente.

Crear un diagrama de componentes: Utiliza un lenguaje de modelado visual como UML (Unified Modeling Language) para crear un diagrama de componentes. En el diagrama, representa cada componente como un rectángulo con su nombre y las interfaces que expone.

Conecta los componentes con líneas que representen las relaciones entre ellos.

Agrupar componentes relacionados: Si hay componentes relacionados o que pertenecen a un mismo módulo o subsistema, agrúpalos visualmente en el diagrama de componentes. Esto ayudará a comprender la estructura general del sistema y facilitará la comprensión de las interacciones entre los componentes.

Documentar detalles adicionales: Junto con el diagrama de componentes, documenta detalles adicionales sobre cada componente, como su propósito, responsabilidades, dependencias externas, interfaces de comunicación, etc. Esto ayudará a tener una visión completa de cada componente y su papel en el sistema.

La vista de componentes proporciona una visión general de la estructura modular del sistema y cómo se relacionan los diferentes componentes. Ayuda a los desarrolladores y arquitectos a comprender mejor la organización del software y facilita la comunicación entre los miembros del equipo.

Vista De Despliegue Del Software

La vista de despliegue del software es una representación visual de cómo se implementará y se desplegará el software en un entorno de ejecución. Esta vista muestra los componentes físicos (servidores, dispositivos, redes, etc.) y cómo se relacionan entre sí para soportar la implementación y operación del sistema. Aquí hay algunos pasos para elaborar la vista de despliegue:

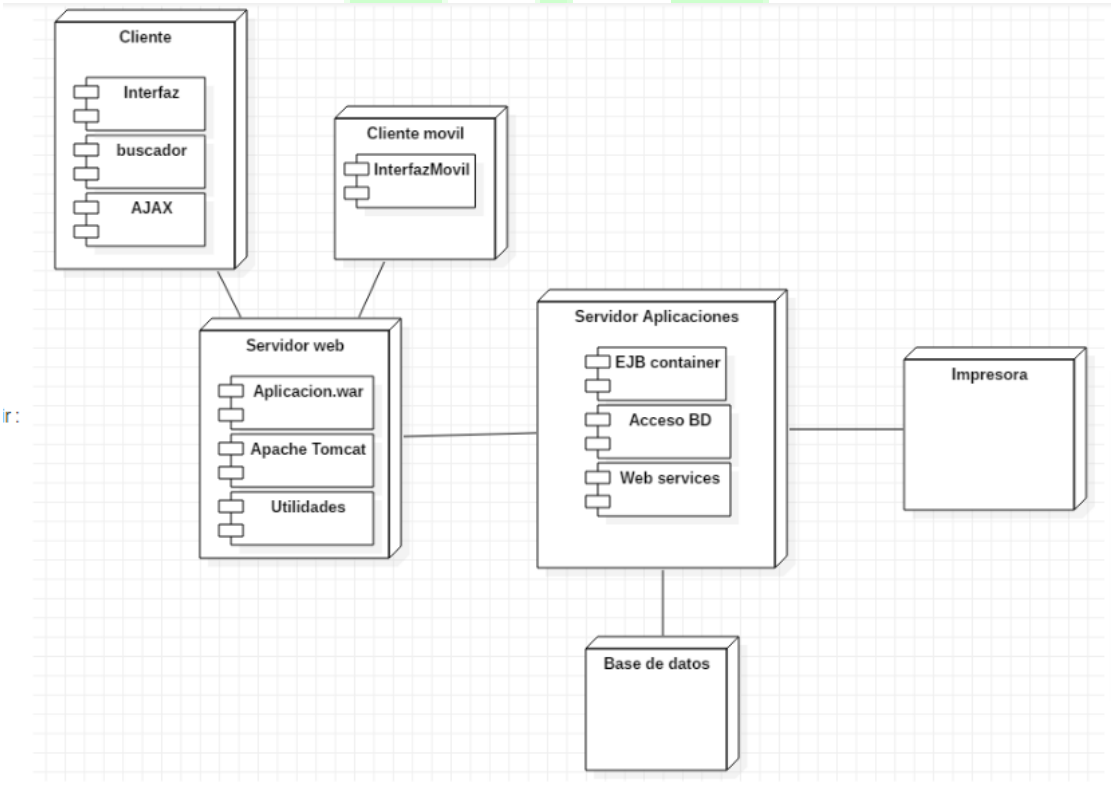
Identificar los nodos físicos: Determina los nodos físicos que formarán parte del entorno de despliegue. Estos nodos pueden ser servidores, estaciones de trabajo, dispositivos de red, bases de datos, sistemas externos, entre otros.

Representar los nodos físicos: En un diagrama de despliegue, representa cada nodo físico como un rectángulo o un símbolo adecuado. Asigna un nombre a cada nodo y, si es necesario, proporciona detalles adicionales, como especificaciones técnicas, sistema operativo, recursos disponibles, etc.

Identificar los componentes de software: Identifica los componentes de software que se implementarán en cada nodo físico. Estos componentes pueden ser aplicaciones, servicios, bases de datos, bibliotecas, etc.

Representar los componentes de software: En el diagrama de despliegue, representa cada componente de software como un rectángulo o un símbolo adecuado, y colócalo dentro del nodo físico correspondiente. Asigna un nombre a cada componente y, si es necesario, proporciona detalles adicionales, como la versión del software o la configuración específica.

Establecer las conexiones entre nodos: Define las conexiones de red o comunicación entre los nodos físicos. Esto puede incluir conexiones cableadas o inalámbricas, conexiones de red local o a través de Internet, entre otros.



Ejemplo de diagrama de despliegue

Documentar detalles adicionales: Junto con el diagrama de despliegue, documenta detalles adicionales sobre cada nodo físico y componente de software. Esto puede incluir detalles de configuración, requerimientos de hardware, dependencias externas, etc.

La vista de despliegue del software proporciona una representación visual clara de cómo se implementará el sistema en el entorno de producción. Ayuda a comprender la infraestructura física necesaria, las interacciones entre los componentes y los requisitos de despliegue del software. Esto es especialmente útil para los equipos de operaciones y administradores de sistemas que son responsables de implementar y mantener el sistema en producción.

CONCLUSIONES

En conclusión, el desarrollo de la arquitectura de software debe considerar lo siguiente:

- La incorporación de patrones de diseño mejora la codificación y mantenibilidad del software.
- La vista de componentes visualiza la estructura del software y sus fases avanzadas.
- La vista de despliegue determina las condiciones de implantación del software.
- La elección de herramientas optimiza los procesos de desarrollo.
- Es fundamental considerar estos aspectos para lograr un diseño eficiente y exitoso del software.

BIBLIOGRAFIAS

https://zajuna.sena.edu.co/Repositorio/Titulada/institution/SENA/Tecnologia/228118/Contenido/OVA/CF18/index.html#/curso/tema7

https://profile.es/blog/patrones-de-diseno-de-software/

https://creately.com/blog/es/diagramas/tutorial-de-diagrama-de-despliegue/