Diseño de patrones de “software”

Breve descripción:

Con el estudio de este componente, el aprendiz estará en capacidad de conceptualizar, interpretar y aplicar diseños de patrones de “software”. Así mismo, se afianzará en lo referente a patrones comportamentales, creacionales, estructurales, vistas estáticas, diagramas de despliegue y de componentes, todo ello en el marco general de análisis y desarrollo de “software”.

Abril 2024

Grupo de Ejecución de la Formación Virtual

Tabla de contenido

Introducción

Los patrones de diseño surgen, inicialmente, en el área de la arquitectura y la ingeniería civil, refiriéndose a casos particulares a los que se enfrentaron en algún momento y de los cuales se documentó una solución particular. Posteriormente, este concepto se empezó a asociar al diseño orientado a objetos. En lugar de referirse a la forma de colocar paredes, puertas y ventanas, se hizo referencia a la forma en que se construyen clases, objetos e interfaces, y la forma en cómo estos deben interactuar.

Los patrones de diseño del "software" representan un conjunto de formas estandarizadas para resolver un problema particular, los cuales surgen a partir de la experiencia desarrollada en la industria.

Un patrón de diseño debe estar estructurado por cuatro componentes:

- El nombre.
- Problema o contexto de aplicación.
- Solución propuesta.
- Ventajas y desventajas.

En este componente formativo, se abordarán algunos de los patrones de diseño más representativos en la industria del "software", los cuales le permitirán implementar, a futuro, soluciones más robustas acordes con los altos estándares de calidad ampliamente conocidos en la industria. Este estudio profundizará en cómo estos patrones facilitan la reutilización de soluciones probadas y eficientes, mejorando así la mantenibilidad y la escalabilidad del "software".

1. Conceptos sobre patrones de diseño

El uso de patrones de diseño en el ámbito de la industria de "software" implica el conocimiento y la aplicación de una serie de criterios que aportan calidad al proceso mismo y efectividad del producto logrado. En ese mismo sentido, se favorece la estandarización del código fuente de determinada aplicación, lo cual facilita su monitoreo y mantenimiento.

Los patrones de diseño tienen las siguientes características:

- Representan soluciones concretas que, aunque se representan de forma genérica, son aplicados para resolver problemas reales.
- Son especificaciones técnicas basadas en los principios de la programación orientada a objetos y dependiendo del lenguaje de programación la forma de implementarlos puede variar.
- Son frecuentemente utilizados ya que se construyen a partir de la experiencia acumulada en la industria del desarrollo de "software".
- Favorecen la implementación de las características de la programación orientada a objetos como la encapsulación, las jerarquías y el polimorfismo.
- No necesariamente el uso de un patrón implica el uso de palabras claves, reservadas o de librerías especializadas.
- Los patrones generalmente hacen referencia al uso de interfaces, clases y objetos que deben ser ajustados de acuerdo con la solución concreta a desarrollar.

Para comprender mejor la organización y aplicación de los patrones de diseño en el desarrollo de “software”, es importante reconocer su clasificación. Estos suelen dividirse en tres grandes grupos, según su finalidad:

- Patrones comportamentales
- Patrones creacionales
- Patrones estructurales

2. Patrones comportamentales

Los patrones de diseño comportamentales se centran en definir la manera en que los objetos interactúan entre ellos, por medio de mensajes. Son tres los tipos de patrones comportamentales: estrategia, comando e iterador.

Estrategia

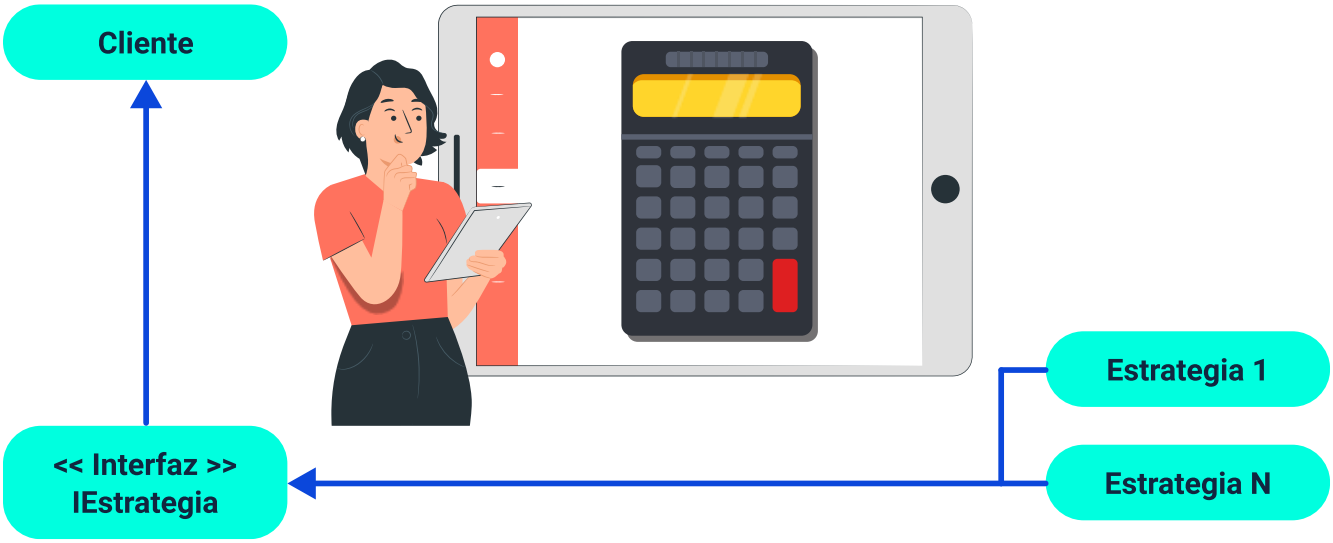
El patrón estrategia permite encapsular un conjunto de algoritmos de manera que puedan ser seleccionados dinámicamente para su ejecución, durante el tiempo de ejecución y de acuerdo con las acciones del cliente. Este patrón es una de las formas en las que se ven reflejadas fácilmente, las características de la programación orientada a objetos, particularmente en lo referente a encapsulamiento y polimorfismo. Landa. (2018).

Este patrón de diseño es útil cuando una misma funcionalidad puede ser provista usando diferentes mecanismos, algoritmos o estrategias, que serán seleccionadas dependiendo de las acciones realizadas por el cliente, en el momento que está ejecutando el programa.

Ejemplo

Suponga que quiere implementar una calculadora, la cual provee un conjunto de operaciones (suma, resta, multiplicación y división) que serán usadas por el cliente según su deseo. Cada una de estas operaciones representan una estrategia diferente y será el cliente quien invocará la ejecución de cada una de ellas dependiendo de su deseo por medio de una interfaz que, usando las propiedades del polimorfismo, se transformará para poder responder a cada solicitud.

Figura 1. Ejemplo patrón comportamental



Comando

El patrón comando permite aislar los objetos que realizan una petición de los objetos concretos encargados de recibir y realizar dicha acción, esto permite entre otras cosas, que las peticiones puedan ser enviadas a varios receptores y si se maneja el estado de las solicitudes, controla acciones de tipo Undo y Redo. El patrón comando necesita la implementación de varios elementos:

- **Comando**

  Clase que sirve de puente entre el cliente y los receptores.

- **“Invoker”**

  Elemento usado por los clientes y que le solicita al comando llevar a cabo una acción.

- **Cliente**

  Invoca la ejecución de las acciones desde el “Invoker”.

- **Icomando**

Interfaz donde se especifican las operaciones a ejecutar.

- **Receptor**

  Clase que realiza la acción.

- **Ejecutar**

  Operación que necesita ser llevada a cabo.

Ejemplo

Una persona (Cliente) quiere hacer uso del televisor y hace todas las solicitudes de servicios por medio del control remoto (“Invoker”). El control remoto se comunica con una interfaz encargada de responder solicitudes de cada comando, que el usuario puede hacer como, por ejemplo, prender el televisor, apagar el televisor, subir el volumen, entre otros. Cada comando realiza una acción particular sobre el televisor (Receptor).

Figura 2. Ejemplo patrón comportamental Comando

Iterador

Este patrón de diseño está orientado al trabajo con colecciones y facilita el acceso a todos los elementos de la colección sin tener la necesidad de conocer su estructura.

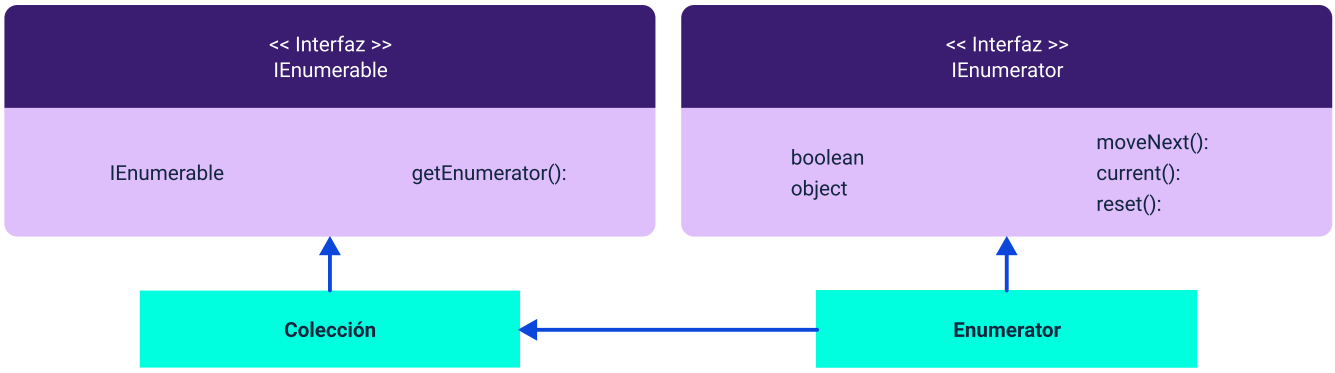
En este patrón se reconocen los enumeradores y los iteradores. Los enumeradores se establecen para conocer el siguiente elemento de la estructura y los iteradores corresponden a un mecanismo para recorrer la estructura, de acuerdo con su secuencia, de inicio a fin. El enumerado se encarga de implementar un conjunto de métodos estándar para poder establecer la secuencia con la que se debe recorrer la estructura.

Los métodos más comunes:

- "MoveNext"() que indica si existe o no un próximo elemento por recorrer.
- Método "Current"() que devuelve el valor actual de la colección según la posición actual en la secuencia.
- Método "Reset"() que permite iniciar nuevamente la secuencia a su punto de partida.

El iterador necesita del enumerador para poder hacer el proceso de recorrido. Este tipo de operaciones son tan comunes en los sistemas actuales, que los lenguajes de programación ya poseen una implementación propia del patrón iterador.

Figura 3. Ejemplo para el patrón comportamental iterador


<< Interfaz >>
IEnumerable
IEnumerable
getEnumerator():
<< Interfaz >>
IEnumerator
boolean
object
moveNext():
current():
reset():
Colección
Enumerator

3. Patrones creacionales

Una de las labores comunes en el proceso de construcción de “software” es precisamente distribuir responsabilidades en un conjunto de módulos o clases, siguiendo los principios definidos en los paradigmas de programación, como por ejemplo la programación orientada a objetos. Sin embargo, al momento de dar solución a un requerimiento particular se requerirá instanciar objetos de diferentes tipos, los cuales bajo sus responsabilidades implementan algún tipo de lógica.

En el recurso que le presentamos a continuación, profundice en los aspectos más importantes de los patrones creacionales:

Video 1. Patrones creacionales

Enlace de reproducción del video

Síntesis del video: Patrones creacionales
Los patrones de diseño creacionales se encargan de definir diferentes tipos de estrategia que pueden ser usadas al momento de requerir la instancia de un objeto particular. “Singleton”, este patrón de diseño creacional se encarga de definir la forma en que podemos garantizar que exista una única instancia de una clase particular en el contexto de la aplicación, esto es útil para casos en los que por cuestiones de manejo de memoria o de la lógica del negocio se requiere que sea el mismo objeto quien responda todos los mensajes independientemente del contexto actual de la aplicación, este patrón se ve reflejado en el cuerpo de la misma clase que se requiere sea instancia una sola vez, un ejemplo puede ser el manejo de conexiones a bases de datos, en algunos casos no sería conveniente instanciar nuevos objetos de conexión cada vez que se genere un evento, uno de los mecanismos usados para implementar el patrón “singleton” es la definición de constructores privados por lo cual es imposible crear instancias de forma directa, la única forma de instanciar un objeto de la clase es por medio del método “getInstance” el cuál verificará si existe o no una instancia creada de esta clase para determinar si usará el constructor primera invocación o si simplemente devuelve la instancia ya creada para el resto de las veces. Fabrica abstracta, este factor de diseño permite la instanciación de una familia de objetos relacionados o dependientes sin necesidad de entrar en detalles particulares de su implementación, de forma que tanto el cliente como los elementos creados por la fábrica son totalmente independientes unos de otros. Este patrón define una interfaz de tipo fábrica la cual se refina por medio de la creación de fábricas concretas, las cuales pueden producir diferentes tipos de objetos y en

diferentes combinaciones según las necesidades del cliente, los elementos importantes de este patrón son: interfaz de la fábrica abstracta, la cual relaciona las operaciones de creación de los productos abstractos; fábricas concretas, las cuales implementan las operaciones de creación definidas en el interfaz de fábrica abstracta; interfaz de producto, el cual relaciona los comportamientos de los productos o elementos; productos concretos, los cuales implementan los comportamientos definidos en la interfaz de producto.

4. Patrones estructurales

Los patrones estructurales proveen una orientación relacionada con la forma de definir los componentes de los objetos. Dentro de los patrones estructurales se encuentran:

- **El patrón Fachada**

  Utilizado cuando el sistema está compuesto por varios subsistemas.

- **El patrón "Delegate"**

  Usado cuando se quiere reutilizar y extender funcionalidades de una clase, sin hacer uso de la herencia.

4.1. Fachada

El patrón fachada se emplea cuando un sistema está conformado por múltiples subsistemas, lo que complica la gestión de las interacciones que el cliente debe realizar con cada uno de ellos. Este patrón permite generar al cliente una vista de alto nivel que simplifica el control y el envío de mensajes a los subsistemas, ocultando los detalles relacionados con la gestión de las clases e instancias.

Existen diferentes variaciones del patrón fachada:

- **Opaco**

  Una de las variaciones más utilizadas y en la cual los clientes no pueden acceder a los subsistemas, solo se puede hacer mediante el objeto fachada.

- **Transparente**

  El cliente tiene la posibilidad de acceder a los subsistemas por medio de la fachada, pero también puede hacerlo de forma directa.

- **Estática**

  En esta variación la fachada se implementa como una clase estática, por lo cual, no se hace necesaria la instancia de un objeto concreto de la fachada.

En este patrón se reconocen tres partes fundamentales. Landa, (2018):

- **Fachada**

  Clase que provee las operaciones de alto nivel que serán usadas por el cliente.

- **Subsistemas**

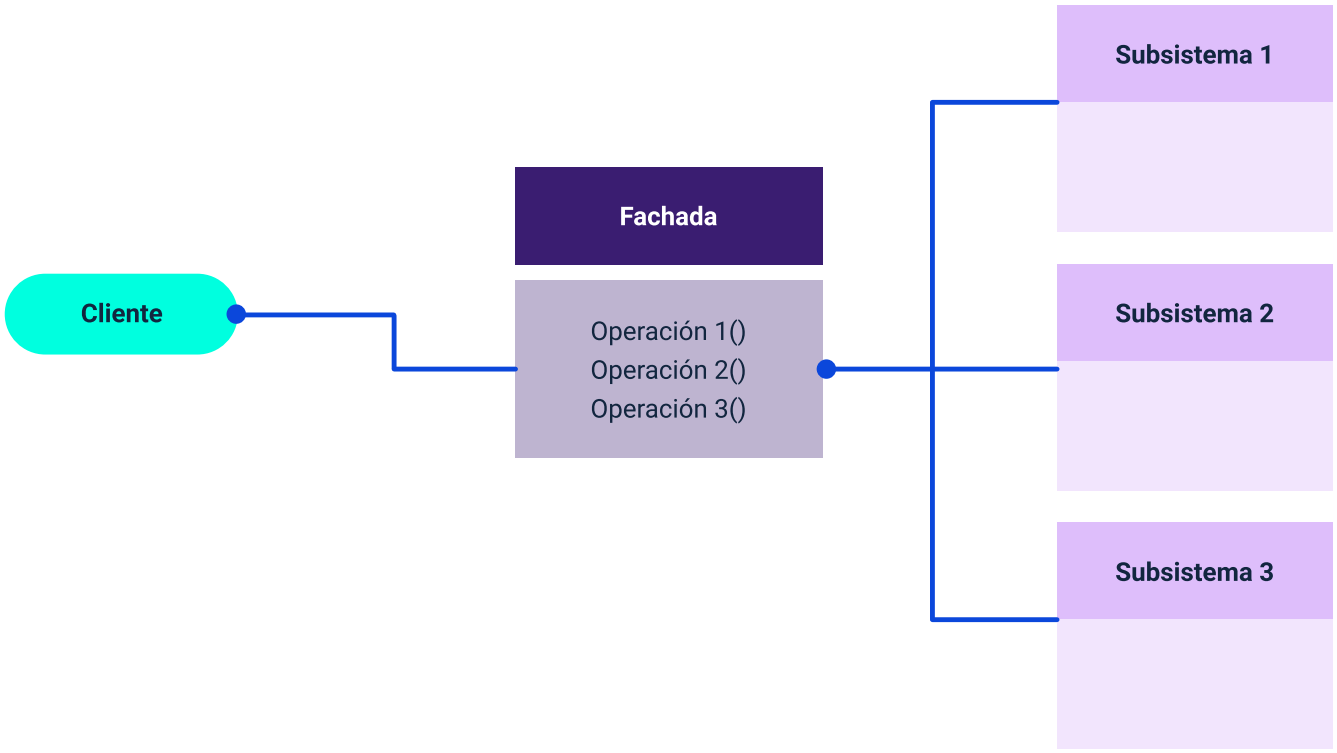
  Clases que proveen las funcionalidades que son expuestas por la fachada.

- **Cliente**

  Hace uso de las operaciones de alto nivel por medio de la fachada.

Se representa de la siguiente manera:

Figura 4. Diagrama patrón fachada



A continuación, se muestra una implementación en código Java de cada uno de los elementos requeridos para la implementación del patrón fachada en un ejemplo sencillo.

Ejemplo: el cliente quiere realizar una compra, pero este proceso involucra acciones por parte de tres diferentes subsistemas:

- **Subsistema de compras**: verifica la viabilidad de la tarjeta para el pago.
- **Subsistema de inventario**: verifica si hay productos en "`stock`".
- **Subsistema de envíos**.

Para disminuir la complejidad, de la cual no debería ocuparse el cliente, se creará una fachada que encapsula las acciones de todos los subsistemas involucrados y proveer una interfaz simple con la que el cliente podrá interactuar más fácilmente. Como se presenta a continuación:

Clase que representa el subsistema de compras

```
public class GestorCompra {

 private Scanner in = new Scanner(System.in);

 public boolean comprar(){

    int numero;

    System.out.println("Ingrese el número de tarjeta para realizar el pago");

    número = in.nextInt();

    if(numero == 4567){

       System.out.println("Procesando la compra");

       System.out.println("--------------------");

       System.out.println("Pago aceptado");

       return true;

    }else{

       System.out.println("Pago rechazado");

       return false;

    }

  }

}
```

Clase que representa el subsistema de inventario

```
public class GestorInventario {

    private int stock;

    public GestorInventario() {

        this.stock = 2;

    }

    public boolean retirarStock(){

        if(stock > 0){

            System.out.println("Producto listo para envío");

            stock--;

            return true;

        }else{

            System.out.println("Producto no disponible, no hay existencias, entrega reprogramada!");

            return false;

        }

    }

}
```

Clase que representa el subsistema de envíos

```
public class GestorEnvio {
  public void enviarPedido(){
    System.out.println("Envio autorizado y en camino!");
  }
}
```

Clase que representa la fachada de todos los subsistemas

```
public class Fachada {
  //Instancias de los subsistemas que gestiona
  private GestorCompra compra = new GestorCompra();
  private GestorInventario inventario = new GestorInventario();
  private GestorEnvio envio = new GestorEnvio();
  //Metodo de alto nivel que se ofrece al cliente
  public void compra(){
    if(compra.comprar() && inventario.retirarStock()){
      envio.enviarPedido();
    }
  }
}
```

Clase que representa al cliente

```
public class Cliente {

  public static void main(String[] args) {

    Fachada fachada = new Fachada();

    // Se hace uso de las operaciones de alto nivel

    fachada.compra();

    fachada.compra();

    fachada.compra();

  }

}
```

4.2. "Delegate"

El patrón "delegate" se usa cuando se quiere reutilizar y extender funcionalidades de una clase sin hacer uso de la herencia. Este patrón permite de cierta forma implementar algo similar a la herencia múltiple que no es admitido por algunos lenguajes de programación, pero adicionalmente permite tener un control más detallado sobre este proceso, ya que se puede ocultar parte de los elementos heredados o, incluso, compartir elementos que no son posibles de heredar bajo el mecanismo de herencia tradicional.

Este patrón busca evitar la concentración de todas las responsabilidades en una única instancia, optando en cambio por delegar las tareas en otras instancias especializadas en resolver dichas funciones.

El siguiente ejemplo se muestra la implementación del patrón de diseño en cuestión, donde se aplica el patrón "Delegate" para integrar en una clase concreta funcionalidades definidas en otras clases a través del uso de interfaces.

Creación de interfaces para comportamientos reutilizables

```
public interface IDisenadora {
    void disenar();
}
public interface ICodificadora {
    void codificar();
}
```

Implementación de interfaces en clases concretas

```
public class ClaseCodificadora implements ICodificadora{
  @Override
  public void codificar() {
     System.out.println("Codificación por la clase codificadora");
  }
}
public class ClaseDisenadora implements IDisenadora{
  @Override
  public void disenar() {
```

```
        System.out.println("Diseñado por clase Diseñadora");
    }
}
```

Generación de clase reutilizadora de funcionalidades

```
ublic class Empleado implements ICodificadora,IDisenadora{
    ICodificadora codificador;
    IDisenadora disenador;
    public Empleado(ICodificadora codificador, IDisenadora disenador) {
        this.codificador = codificador;
        this.disenador = disenador;
    }
    @Override
    public void codificar() {
        codificador.codificar();
    }
    @Override
    public void disenar() {
        disenador.disenar();
    }
```

```
}
```

Instanciación de la clase principal con clases concretasrserán reutilizados

```
public class PatronDelegate {

    public static void main(String[] args) {

        Empleado objEmpleado = new Empleado(new ClaseCodificadora(),new
ClaseDisenadora());

        objEmpleado.codificar();

        objEmpleado.disenar();

    }

}
```

Cualquier nueva clase concreta que implemente las interfaces originales podrá ser utilizada por la clase Empleado para su reutilización, simplemente pasando la instancia correspondiente que implementa la nueva lógica al parámetro del constructor.

5. Vistas estáticas

La vista estática está encargada de modelar los conceptos significativos del dominio de la aplicación desde sus propiedades internas y las relaciones existentes. Se denomina vista estática porque no modela el comportamiento del sistema ni muestra las variaciones que se puedan presentar por efecto del tiempo.

Los elementos fundamentales de la vista estática son las clases que describen los conceptos del dominio del problema y las relaciones que pueden ser de tipo:

- Asociación
- Dependencia
- Generalización

Entre los diagramas de Lenguaje Unificado de Modelado (UML) que se utilizan para representar la vista estática del sistema encontramos (ITCA, 2021):

- Diagrama de clases.
- Diagrama de objetos.
- Diagramas de componentes.

6. Diagrama de despliegue

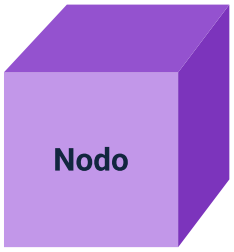
Los diagramas de despliegue, que forman parte de los tipos de diagramas propuestos por UML, tienen como objetivo representar la arquitectura física del sistema, tanto en "hardware" como en "software", así como las conexiones entre ellos. Estos diagramas son de gran utilidad para el proceso de despliegue del sistema.

Los diagramas de despliegue utilizan un conjunto de elementos gráficos que tienen una representación y significado estandarizado. A continuación, se detalla cada uno de ellos:

- **Nodos**

  Representa un elemento que puede ser "hardware" o "software" y se representa por un cubo.

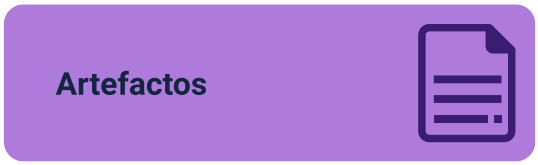
Figura 5. Nodos



- **Artefactos**

  Representan elementos concretos generados en el proceso de desarrollo, por ejemplo bibliotecas, archivos, entre otros.

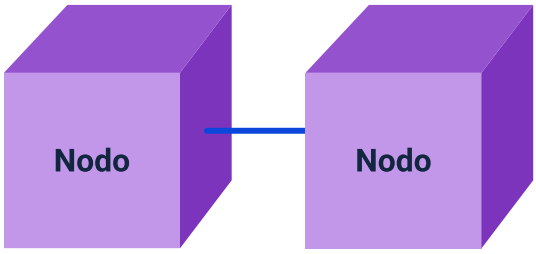
Figura 6. Artefactos



- **Asociación de comunicación**

  Representa el camino de comunicación entre nodos, ilustrado mediante una línea continua que une los nodos correspondientes.

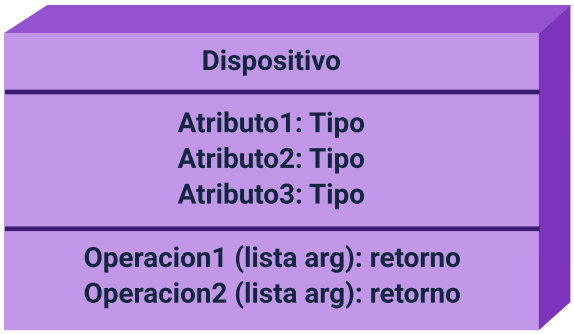
Figura 7. Asociación de comunicación



- **Dispositivos**

  Es un tipo especial de nodo que representa un recurso computacional del sistema, por ejemplo un servidor.

Figura 8. Dispositivos



- **Especificaciones de despliegue**

Representan configuraciones que se deben tener en cuenta para desplegar un artefacto en un nodo.

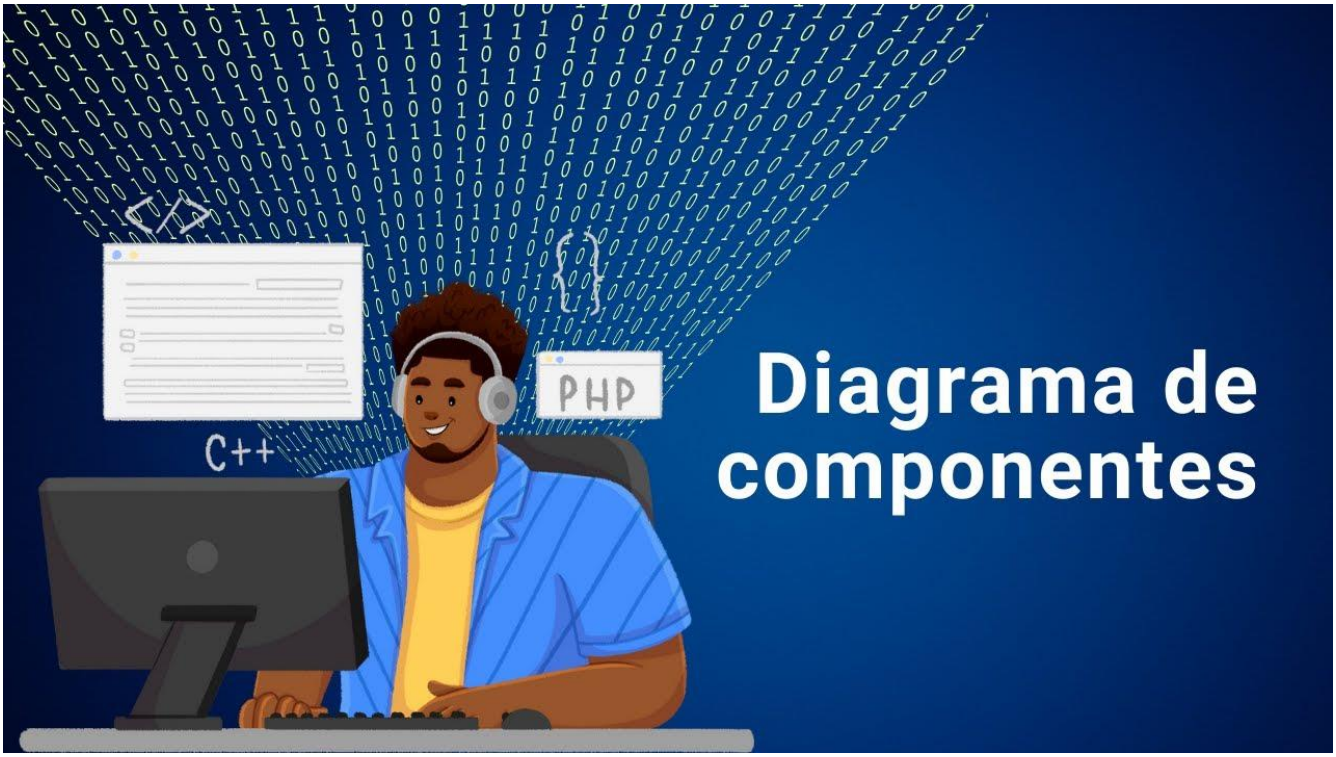
Figura 9.

7. Diagrama de componentes

El diagrama de componentes, uno de los diagramas sugeridos por UML, muestra una perspectiva estática del sistema de información y pertenece a los diagramas estructurales. Representa cómo se organiza y se relaciona el "software" en términos de componentes lógicos o físicos, como bibliotecas, módulos, paquetes y capas. Ofrece una visión general de los componentes del sistema, resaltando la organización y dependencia entre ellos, lo cual es fundamental para comprender cómo se integran y cooperan para realizar las funcionalidades del sistema.

Le invitamos a revisar los aspectos más característicos e importantes de los diagramas de componentes, explorando el recurso que se muestra a continuación:

Video 2. Diagrama de componentes

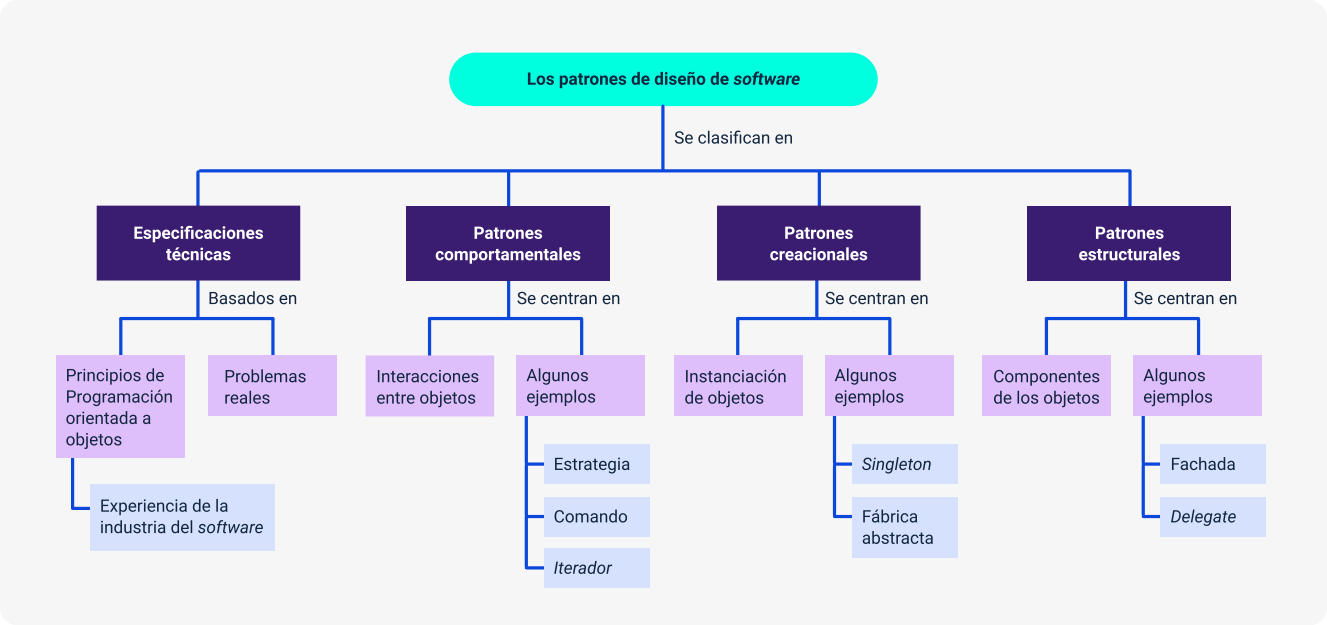


Enlace de reproducción del video

Síntesis del video: Diagrama de componentes
El diagrama de componentes es uno de los diagramas propuestos en UML, que representa una vista estática del sistema de información y hace parte de los diagramas estructurales, este tipo de diagrama proporciona una vista de alto nivel de los componentes dentro del sistema y generalmente se construye posterior a la construcción del diagrama de clases. Un componente puede ser "software", como por ejemplo las bases de datos o una interfaz de usuario, también puede ser "hardware" como un dispositivo o incluso una unidad como la nómina, el inventario, proveedores, etc. Los diagramas de componentes son muy útiles para arquitecturas orientadas a servicios, permiten mostrar la estructura general del código por lo que puede ser usado para mostrar las funciones del sistema que se construye a cualquier parte interesada. Los elementos que conforman un diagrama de componentes son: Componente: es una abstracción de un nivel más alto que las clases y representa unidades lógicas del sistema. Interfaz: las interfaces siempre se asocian a los componentes y representan el lugar que debe ser usado por otros componentes para poder establecer comunicación con el. Relación de dependencia: representa una relación más general entre dos componentes para indicar que un componente requiere de otro, para poder ejecutar su trabajo. Paquetes: también es posible utilizar los paquetes para agrupar lógicamente un conjunto de componentes dentro de un subsistema.

Síntesis

A continuación, se presenta una síntesis de la temática estudiada en el componente formativo:



Material complementario

Tema	Referencia	Tipo de material	Enlace del recurso
Conceptos sobre patrones de diseño	Leiva, A. (2020). Patrones de diseño “software”: Repaso completo en 10 minutos.	Video	https://www.youtube.com/watch?v=6BHOeDL8vls&feature=youtu.be
Diagrama de despliegue	Nicosiored. (2018). Diagrama de Despliegue - 22 - Tutorial UML en español.	Video	https://www.youtube.com/watch?v=NSB0ATJUavA&feature=youtu.be
Diagrama de componentes	Nicosiored. (2018). Diagrama de Componentes I - 20- Tutorial UML en español.	Video	https://www.youtube.com/watch?v=oOycG_n1ARs&feature=youtu.be

Glosario

Lenguaje Unificado de Modelado (UML): relaciona un conjunto de diagramas estandarizados para la representación de sistemas de información desde diferentes tipos de vista.

Patrones GOF: los patrones GOF, se trata de los patrones de diseño y se denominan de esa manera por el libro "Desing Patterns elements of reusable software", de cuatro autores que descubrieron la manera fundamental de adentrarse en el mundo de la programación. GOF es la sigla de "The Gang of four", que significa La pandilla de los cuatro.

"Software": se trata del compendio de rutinas o de programas que favorecen a los dispositivos el cumplimiento y ejecución de determinadas tareas, funciones o aplicaciones.

Referencias bibliográficas

Creately (2021, Enero 15). La Guía Fácil de los Diagramas de Despliegue UML. [Web log post]. Blogspot. https://creately.com/blog/es/diagramas/tutorial-de-diagrama-de-despliegue/

DiagramasUML (2013, Diciembre 8). ¿Qué es UML? ¿Qué diagramas componen UML? [Web log post]. Blogspot. https://diagramasuml.com/componentes/

EcuRed (2021). Patrones Gof. ECURED. https://www.ecured.cu/Patrones_Gof

Escuela especializada en ingeniería (2021). Diagramas UML estáticos. ITCA-FEPADE. https://virtual.itca.edu.sv/Mediadores/ads/222_diagramas_uml_estticos.html

Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Booch, G. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.

Landa, N. (2018). Patrones de Diseño de Software. [Video]. YouTube. https://www.youtube.com/playlist?list=PLM-p96nOrGcbqbL_A29b0z3KUXdq2_fpn

Créditos

Nombre	Cargo	Centro de Formación y Regional
Milady Tatiana Villamil Castellanos	Líder del Ecosistema	Dirección General
Olga Constanza Bermudez Jaimes	Responsable de Línea de Producción	Centro de Servicios de Salud - Regional Antioquia
Jonathan Guerrero Astaiza	Experto temático	Centro de Teleinformática y Producción Industrial - Regional Cauca
Paola Alexandra Moya	Evaluadora Instruccional	Centro de Servicios de Salud - Regional Antioquia
Juan Daniel Polanco Muñoz	Diseñador de Contenidos Digitales	Centro de Servicios de Salud - Regional Antioquia
Jhon Jairo Urueta Álvarez	Desarrollador Fullstack	Centro de Servicios de Salud - Regional Antioquia
Edgar Mauricio Cortés García	Actividad Didáctica	Centro de Servicios de Salud - Regional Antioquia
Daniela Muñoz Bedoya	Animador y Productor Multimedia	Centro de Servicios de Salud - Regional Antioquia
Luis Gabriel Urueta Álvarez	Validador de Recursos Educativos Digitales	Centro de Servicios de Salud - Regional Antioquia
Margarita Marcela Medrano Gómez	Evaluador para Contenidos Inclusivos y Accesibles	Centro de Servicios de Salud - Regional Antioquia
Daniel Ricardo Mutis Gómez	Evaluador para contenidos inclusivos y accesibles	Centro de Servicios de Salud - Regional Antioquia