**FORMATO PARA EL DESARROLLO DE COMPONENTE FORMATIVO**

| PROGRAMA DE FORMACIÓN | Análisis y desarrollo de Software |
| --- | --- |

| COMPETENCIA | 220501095 – Diseñar la solución de Software de acuerdo con los procedimientos y los requisitos técnicos. | RESULTADOS DE APRENDIZAJE | 220501095-01 - Elaborar los artefactos de diseño del software siguiendo las prácticas de la metodología seleccionada. |
| --- | --- | --- | --- |

| NÚMERO DEL COMPONENTE FORMATIVO | 18 |
| --- | --- |
| NOMBRE DEL COMPONENTE FORMATIVO | 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 *softwares*. |
| PALABRAS CLAVE | Arquitectura de sistema, diagrama, GOF, patrones, vistas estáticas |

| ÁREA OCUPACIONAL | 2 - CIENCIAS NATURALES, APLICADAS Y RELACIONADAS |
| --- | --- |
| IDIOMA | Español |

1. **TABLA DE CONTENIDO:**

**Introducción**

1. Conceptos sobre patrones de diseño
2. Patrones comportamentales
3. Patrones creacionales
4. Patrones estructurales
5. Vistas estáticas
6. Diagrama de despliegue
7. Diagrama de componentes

Síntesis

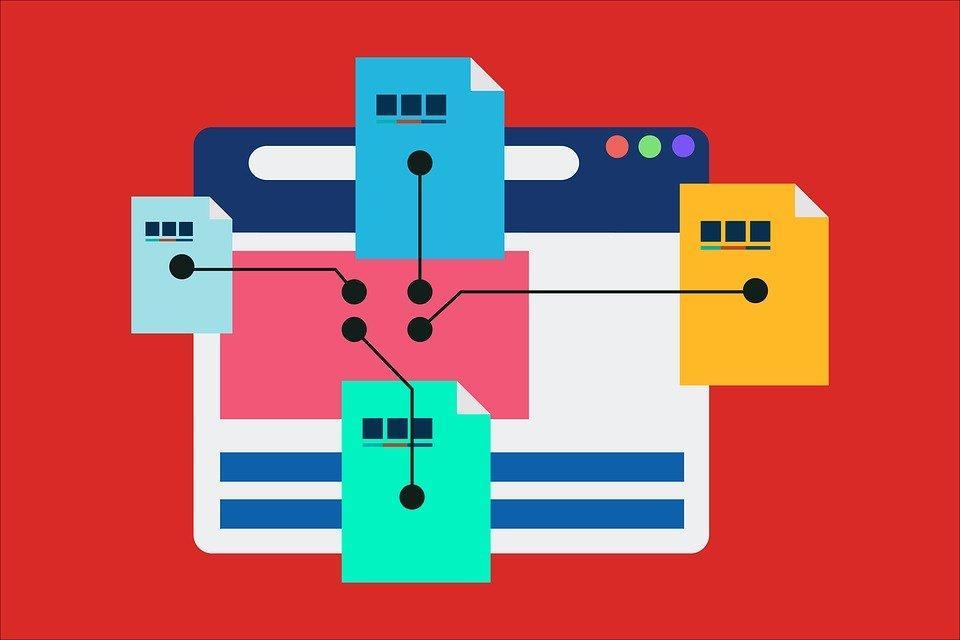
1. **DESARROLLO DE CONTENIDO:**

**Introducción**

Le damos la bienvenida al componente formativo “Diseño de patrones de *software*”. Para comenzar el recorrido por el mismo, visite el recurso didáctico que se muestra a continuación:



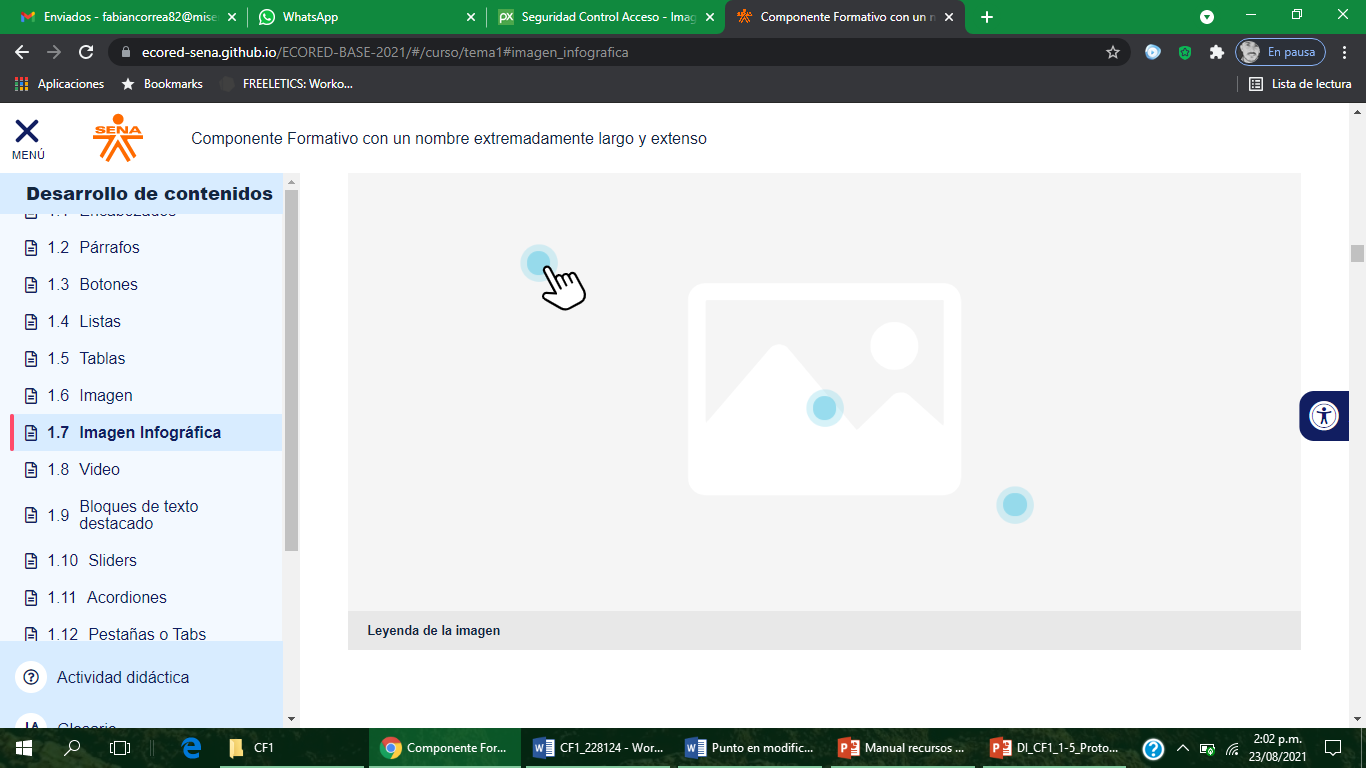
1. **Conceptos sobre patrones de diseño**



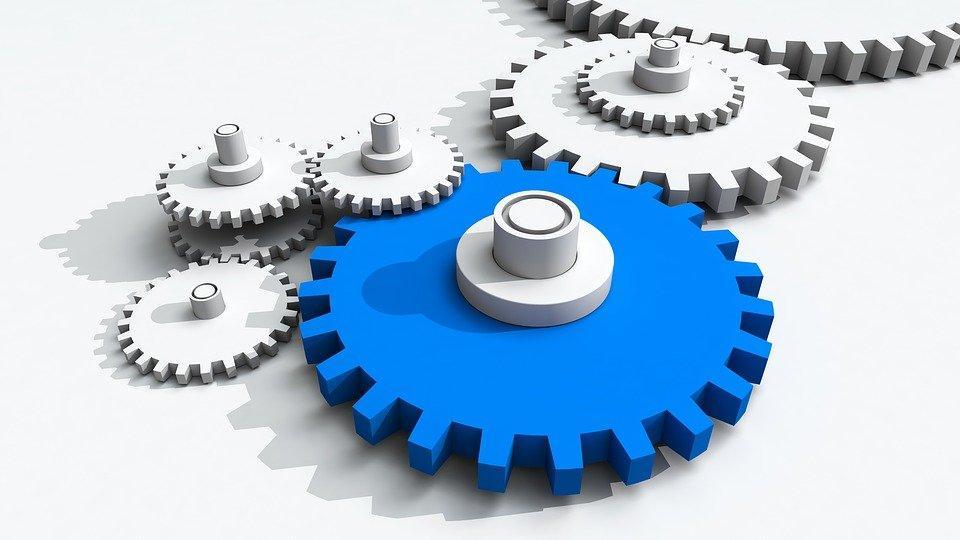
El uso de patrones de diseño en el ámbito de la industria de *software*, implica el conocimiento y la aplicación de una serie de criterios que aportan calidad al proceso mismo y efectividad del producto logrado. En ese mismo sentido, se favorece la estandarización del código fuente de determinada aplicación,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



1. **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.

Explore el siguiente recurso y conozca los aspectos más relevantes relacionados con patrones comportamentales:



1. **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 lógica.

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



1. **Patrones estructurales**



Los patrones estructurales proveen una orientación relacionada a 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.

**Fachada**

El patrón fachado se utiliza cuando el sistema está compuesto por varios subsistemas y se hace complejo gestionar los mensajes que debe realizar el cliente en cada uno de estos subsistemas. 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 el cual los clientes no pueden acceder a los subsistemas sólo 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. Landa. (2018)

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.

En la figura 1 se pueden ver cada uno de los componentes del patrón.

**Figura 1**

*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. En este ejemplo el cliente quiere realizar una compra, pero este proceso involucra acciones por parte de tres diferentes subsistemas: subsistema de compras que verifica la viabilidad de la tarjeta para el pago, el subsistema de inventario que verifica si hay productos en stock y el 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.

| 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();**  **}**  **}** |

Para obtener una copia completa del proyecto, para su verificación, usted puede consultar el repositorio público disponible en: <https://gitlab.com/jonathanga/patron-fachada-en-java/-/tree/master>.

***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 lo que busca es evitar asumir todas las responsabilidades en una sola instancia y delegar las actividades en otras instancias que son especializadas en resolver dicha tarea.

A continuación, se muestra un ejemplo de implementación de este patrón de diseño donde se hace uso del patrón *delegate* para incorporar en una clase concreta, funcionalidades que están definidas en otras clases haciendo uso de Interfaces.

| En primer lugar se crean las interfaces que definen el comportamiento a ser reutilizado |
| --- |
| **public interface IDisenadora {**  **void disenar();**  **}**  **public interface ICodificadora {**  **void codificar();**  **}** |
| Se crean clases concretas que implementen este funcionamiento definido en las interfaces |
| **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");**  **}**  **}** |
| Ahora se crea la clase que va a reutilizar las funcionalidades definidas en las interfaces |
| **public 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();**  **}**  **}** |
| En este caso la clase principal instancia la clase indicando por parámetro las clases concretas a ser utilizadas para la extensión de los métodos que serán 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 usada por la clase Empleado para su reutilización solo indicando en el parámetro del constructor la respectiva instancia que implementa la nueva lógica.** |

Para obtener una copia completa del proyecto, para su verificación, usted puede consultar el repositorio público disponible en: <https://gitlab.com/jonathanga/patron-delegate/-/tree/master>.

1. **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, Generalización y de dependencia. Entre los diagramas de UML que se utilizan para representar la vista estática del sistema encontramos (ITCA, 2021):

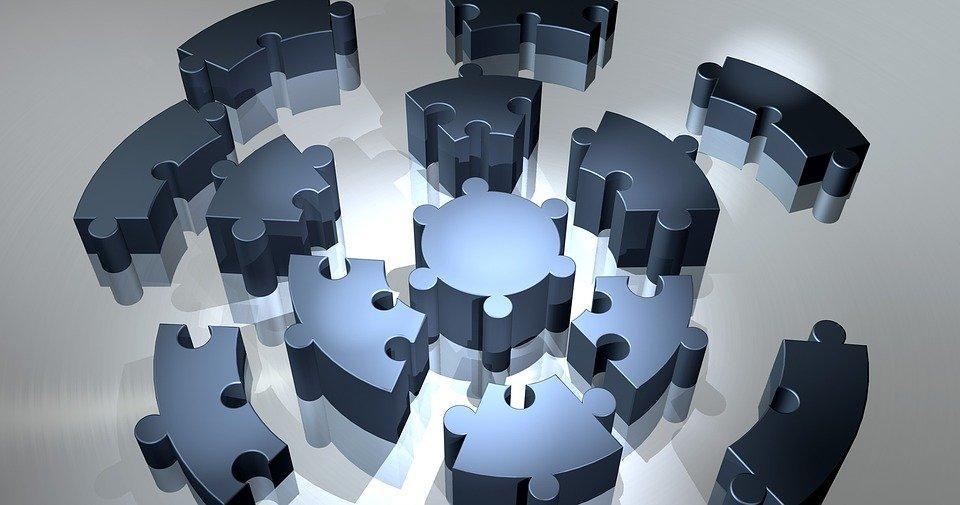
* Diagrama de clases.
* Diagrama de objetos.
* Diagramas de componentes.

1. **Diagrama de despliegue**



Los diagramas de despliegue hacen parte de los tipos de diagrama propuestos por UML y su objetivo es la representación de la arquitectura del sistema en términos de hardware y software físico y los medios por los cuales se conectan. Este tipo de diagrama es muy útil 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:



1. **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 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.

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



**Síntesis**

Hasta este punto, el componente formativo ha favorecido la profundización en los aspectos y generalidades más importantes relativos al diseño de patrones de software.

Para lograr una síntesis general y tener más y mejores herramientas conceptuales y de aplicación, explore el recurso que se sugiere a continuación:



1. **ACTIVIDADES DIDÁCTICAS (OPCIONALES SI SON SUGERIDAS)**

| DESCRIPCIÓN DE ACTIVIDAD DIDÁCTICA | |
| --- | --- |
| Nombre de la actividad | Repaso de características representativas en patrones de diseño. |
| Objetivo de la actividad | Afianzar las características más importantes de algunos patrones de diseño de software. |
| Tipo de actividad sugerida | Arrastrar y soltar el nombre del patrón de diseño con la característica que lo identifica. |
| Archivo de la actividad  (Anexo donde se describe la actividad propuesta) | Anexo documento en Word llamado Actividad didáctica 1 |

1. **MATERIAL COMPLEMENTARIO**

| **Tema** | **Referencia APA del Material** | **Tipo de material**  **(Video, capítulo de libro, artículo, otro)** | **Enlace del recurso o**  **archivo del documento o material** |
| --- | --- | --- | --- |
| Conceptos sobre patrones de diseño | Leiva, A. (2020). *Patrones de diseño software: Repaso completo en 10 minutos* [Video]. YouTube. <https://www.youtube.com/watch?v=6BHOeDL8vls&feature=youtu.be> | Video | <https://youtu.be/6BHOeDL8vls> |
| Diagrama de despliegue | Nicosiored. (2018). *Diagrama de Despliegue - 22 - Tutorial UML en español* [Video]. YouTube. <https://www.youtube.com/watch?v=NSB0ATJUavA&feature=youtu.be> | Video | <https://youtu.be/NSB0ATJUavA> |
| Diagrama de componentes | Nicosiored. (2018). *Diagrama de Componentes I - 20- Tutorial UML en español* [Video]. YouTube. <https://www.youtube.com/watch?v=oOycG_n1ARs&feature=youtu.be> | Video | <https://youtu.be/oOycG_n1ARs> |

1. **GLOSARIO**

| TÉRMINO | SIGNIFICADO |
| --- | --- |
| 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. |
| UML | Lenguaje unificado de modelado. Relaciona un conjunto de diagramas estandarizados para la representación de sistemas de información desde diferentes tipos de vista. |

1. **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>

1. **CONTROL DEL DOCUMENTO**

|  | **Nombre** | **Cargo** | **Dependencia** | **Fecha** |
| --- | --- | --- | --- | --- |
| **Autor (es)** | Jonathan Guerrero Astaiza | Experto Temático | Regional Cauca - Centro de teleinformática y producción industrial. | Septiembre 2021 |
| Fabián Leonardo Correa Díaz | Diseñador Instruccional | Regional Tolima - Centro agropecuario La Granja. | Octubre de 2021 |
| Ana Catalina Córdoba Sus | Revisora metodológica y pedagógica | Regional Distrito Capital – Centro para la Industria de la Comunicación Gráfica. | Octubre 2021 |
| Rafael Neftalí Lizcano Reyes | Asesor pedagógico | Regional Santander - Centro Industrial del Diseño y la Manufactura. | Octubre 2021 |
| Sandra Patricia Hoyos Sepúlveda | Revisión y corrección de estilo | Centro para la Industria de la Comunicación Gráfica - Distrito capital | Octubre 2021 |

1. **CONTROL DE CAMBIOS**

|  | Nombre | Cargo | Dependencia | Fecha | Razón del cambio |
| --- | --- | --- | --- | --- | --- |
| Autor (es) |  |  |  |  |  |