

**Universidad Tecnológica de Panamá**

**Facultad de Ingeniería en Sistemas Computacionales**

**Ingeniería en Sistemas de Información**

**Ingeniería de Software Aplicada**

**Investigación**

**Patrones de Diseño**

**Facilitadora**: Ing. Jeanette Riley

**Estudiante:**

**Johel Batista 8-914-587**

**Enlace de la Charla:** [**https://youtu.be/5I0DdMEeO9A**](https://youtu.be/5I0DdMEeO9A)

**I Semestre Académico**

Índice de Contenidos

[Introducción o Comentarios Iniciales 3](#_Toc136806690)

[¿Qué son los Patrones de Diseño? 4](#_Toc136806691)

[Claves de los Patrones de Diseño: 5](#_Toc136806692)

[Historia de Patrones de Diseño 5](#_Toc136806693)

[Claves de la Historia de los Patrones de Diseño: 6](#_Toc136806694)

[Tipos de Patrones de Diseño 7](#_Toc136806695)

[Patrones de Diseño Creacionales 8](#_Toc136806696)

[Patrones de Diseño Estructurales 10](#_Toc136806697)

[Patrones de Diseño de Comportamiento 11](#_Toc136806698)

[Patrón de Diseño “Prototype” 13](#_Toc136806699)

[Caso de Estudio: Aplicación de Dibujo Vectorial 13](#_Toc136806700)

[Implementación Básica | Aplicación del Dibujo Vectorial (Genérico) 14](#_Toc136806701)

[Diagrama de Clases | Patrón de Diseño Prototype: Objetos Geométricos 17](#_Toc136806702)

[Conclusión o Comentarios Finales 20](#_Toc136806703)

[Referencias Bibliográficas 21](#_Toc136806704)

# Introducción o Comentarios Iniciales

La ingeniería de software, como cualquier otra disciplina ingenieril, tiene su base en la resolución de problemas mediante la aplicación de principios, prácticas y patrones probados y efectivos. Entre estos se encuentran los patrones de diseño, que se han convertido en una herramienta crucial en el arsenal de todo desarrollador de software.

Un patrón de diseño es una solución reutilizable a un problema común que se presenta en un contexto específico durante el diseño de software. (“¿Cuál es la diferencia entre un algoritmo y un patrón de diseño?”) Los patrones no son piezas de código que se pueden copiar y pegar en un programa, sino más bien guías o plantillas que pueden ser usadas para resolver problemas de diseño en diversas circunstancias. Estas soluciones están basadas en experiencias y prácticas consagradas de la ingeniería de software, proporcionando un lenguaje común entre los desarrolladores.

Existen diversos tipos de patrones de diseño, clasificados generalmente en tres categorías: creacionales, estructurales y de comportamiento. Los patrones creacionales tratan con la creación de objetos, los estructurales se ocupan de la composición de clases y objetos, y los patrones de comportamiento se ocupan de la comunicación entre objetos.

Los patrones de diseño ofrecen una serie de ventajas. Proporcionan soluciones probadas a problemas de diseño de software, hacen que el código sea más flexible, facilitan la reutilización de código y hacen que el software sea más fácil de mantener. Sin embargo, su aplicación debe ser cuidadosa, ya que no todos los patrones son aplicables a todas las situaciones y su uso incorrecto puede llevar a más problemas que soluciones.

El estudio y aplicación de los patrones de diseño son esenciales para cualquier desarrollador de software, no solo para mejorar la calidad del software que se produce, sino también para mejorar la eficiencia y efectividad de su trabajo.

# ¿Qué son los Patrones de Diseño?

En el corazón de la ingeniería de software yace la capacidad de resolver problemas de manera sistemática y eficiente. Para esto, los profesionales del software a menudo recurren a métodos probados que ayudan a proporcionar soluciones optimizadas a problemas recurrentes en el diseño de software. A estas soluciones se les conoce como patrones de diseño.

Los patrones de diseño son esencialmente plantillas o guías de alto nivel que describen cómo abordar problemas comunes y recurrentes en la ingeniería de software. Estos patrones no representan una solución final o un código específico que se pueda reutilizar directamente, sino que sirven como un esquema o marco de referencia para solucionar problemas particulares. Estos patrones incorporan las mejores prácticas acumuladas a lo largo de la historia del desarrollo de software y presentan soluciones elegantes, eficientes y reutilizables para problemas comunes.

Existen diversos tipos de patrones de diseño, pero generalmente se agrupan en tres categorías principales: patrones creacionales, patrones estructurales y patrones de comportamiento. Los patrones creacionales tratan de la creación y la inicialización de objetos o clases, los patrones estructurales abordan cómo se organizan y relacionan las entidades y los patrones de comportamiento se enfocan en la comunicación e interacción entre objetos y clases.

Los patrones de diseño son una parte integral del lenguaje y la comunicación en la ingeniería de software. Ofrecen un vocabulario común que permite a los desarrolladores discutir soluciones y alternativas de manera clara y precisa, y también ayuda a documentar sistemas de software. Al utilizar patrones de diseño, los ingenieros de software pueden describir las soluciones de manera más abstracta y menos dependiente del lenguaje de programación específico.

Además de mejorar la comunicación, los patrones de diseño también promueven la reutilización del código y la modularidad, facilitando la creación de software de alta calidad de una manera más rápida y eficiente. Esto resulta en sistemas de software más robustos, mantenibles y escalables, lo que finalmente puede llevar a una reducción de costos y tiempo en el ciclo de vida del desarrollo de software.

Sin embargo, es crucial entender que los patrones de diseño no son una panacea y no deben aplicarse indiscriminadamente. Deben ser empleados en función del problema a resolver, y no todos los patrones son apropiados para todas las situaciones. En este sentido, el buen juicio, la experiencia y la comprensión del problema son fundamentales para la correcta utilización de los patrones de diseño.

## ****Claves de los Patrones de Diseño:****

1. **Reutilización y Eficiencia**: Los patrones de diseño representan soluciones optimizadas y reutilizables a problemas comunes en el diseño de software, lo que mejora la eficiencia en el desarrollo y el mantenimiento del software.
2. **Comunicación y Documentación**: Los patrones de diseño proporcionan un lenguaje común que facilita la comunicación entre los desarrolladores y la documentación de los sistemas de software.
3. **Aplicación Cuidadosa**: Los patrones de diseño deben aplicarse con buen juicio y comprensión del problema a resolver, no todos los patrones son apropiados para todas las situaciones.

# Historia de Patrones de Diseño

La historia de los patrones de diseño en la ingeniería de software está íntimamente ligada a la evolución de la propia disciplina y refleja el esfuerzo constante de los profesionales del software por resolver problemas de diseño de manera más eficiente y efectiva.

La idea de los patrones de diseño no es exclusiva de la ingeniería de software. De hecho, tiene sus raíces en la arquitectura, específicamente en el trabajo del arquitecto Christopher Alexander, quien en la década de 1970 introdujo el concepto de patrones para solucionar problemas comunes en el diseño arquitectónico. Alexander describió los patrones como soluciones a problemas que se presentan una y otra vez en el entorno construido.

Esta idea fue adoptada y adaptada a la ingeniería de software en la década de 1980, pero fue en la década de 1990 cuando los patrones de diseño comenzaron a cobrar mayor importancia en la disciplina. El libro "Design Patterns: Elements of Reusable Object-Oriented Software" (1994), escrito por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (colectivamente conocidos como la "Gang of Four" o GoF), es considerado un hito en la historia de los patrones de diseño. En este libro, los autores introdujeron 23 patrones de diseño clasificados en tres categorías: creacionales, estructurales y de comportamiento.

El trabajo de la "Gang of Four" ha tenido un impacto significativo en la ingeniería de software y ha influenciado a generaciones de programadores en su enfoque del diseño de software. Los patrones de diseño que describen han sido ampliamente adoptados y siguen siendo un recurso valioso para los ingenieros de software.

Desde la publicación del libro de la "Gang of Four", se han introducido y documentado muchos otros patrones de diseño, adaptados a diferentes contextos y paradigmas de programación. Los patrones de diseño han influido en la creación de muchos lenguajes de programación y frameworks, y han moldeado la manera en que se enseña y se practica la ingeniería de software.

A día de hoy, los patrones de diseño siguen siendo una parte esencial de la ingeniería de software, proporcionando soluciones probadas a problemas comunes y facilitando la comunicación entre desarrolladores. Sin duda, seguirán evolucionando junto con la disciplina, a medida que los ingenieros de software continúen explorando nuevas formas de resolver problemas de diseño.

## ****Claves de la Historia de los Patrones de Diseño:****

1. **Influencia Arquitectónica**: El concepto de patrones de diseño se originó en la arquitectura y fue adaptado a la ingeniería de software, evidenciando la interconexión entre diversas disciplinas.
2. **Contribución de la "Gang of Four"**: El libro "Design Patterns: Elements of Reusable Object-Oriented Software", escrito por la "Gang of Four", es un hito importante en la historia de los patrones de diseño, estableciendo 23 patrones esenciales que aún son ampliamente utilizados hoy en día.
3. **Evolución Continua**: Los patrones de diseño han evolucionado y se han expandido a lo largo de la historia de la ingeniería de software, y seguirán evolucionando para adaptarse a nuevos contextos y paradigmas de programación.

# Tipos de Patrones de Diseño

Los patrones de diseño en ingeniería de software se clasifican generalmente en tres categorías principales: patrones creacionales, patrones estructurales y patrones de comportamiento. Cada tipo de patrón de diseño aborda un conjunto diferente de problemas de diseño y tiene su propia forma de hacer frente a los desafíos inherentes al desarrollo de software.

**1. Patrones Creacionales:** Los patrones de diseño creacionales se ocupan de los mecanismos de creación de objetos. Proporcionan maneras de crear objetos mientras ocultan la lógica de creación, en lugar de instanciar objetos directamente. Esto permite a los programadores hacer el sistema independiente de cómo sus objetos son creados, compuestos y representados. Algunos ejemplos notables de patrones creacionales incluyen el Singleton, Factory Method, Abstract Factory y Builder.

**2. Patrones Estructurales:** Los patrones de diseño estructurales se centran en cómo se organizan y relacionan los objetos y las clases. Se preocupan por la forma en que las clases, los objetos y los componentes se componen para formar estructuras más grandes. Estos patrones proporcionan diferentes maneras de asegurar que las partes de un sistema se ajusten de manera eficiente y efectiva, y se comuniquen entre sí correctamente. Algunos ejemplos de patrones estructurales incluyen el Adapter, Decorator, Proxy y Composite.

**3. Patrones de Comportamiento:** Los patrones de diseño de comportamiento se ocupan de la interacción y la distribución de responsabilidades entre objetos y clases. Proporcionan soluciones para la interacción entre objetos y cómo se dividen las responsabilidades. Estos patrones ayudan a asegurar que los componentes de un sistema interactúen de manera eficiente y coherente, al tiempo que promueven la flexibilidad y la reutilización del código. Ejemplos de patrones de comportamiento incluyen el Observer, Strategy, Command y State.

La elección del patrón de diseño a utilizar depende de la naturaleza del problema que se quiere resolver. Cada patrón tiene sus propias ventajas y desventajas, y su uso efectivo depende de un entendimiento profundo de los desafíos de diseño y las implicaciones de la implementación del patrón. Por tanto, es fundamental para los ingenieros de software conocer y entender los diferentes tipos de patrones de diseño, y saber cómo y cuándo aplicar cada uno.

**Claves de los Tipos de Patrones de Diseño:**

1. **Patrones Creacionales**: Se centran en los mecanismos de creación de objetos, proporcionando maneras de crear objetos mientras ocultan la lógica de creación, permitiendo que el sistema sea independiente de cómo sus objetos son creados y representados.
2. **Patrones Estructurales**: Se preocupan por la forma en que las clases, los objetos y los componentes se componen para formar estructuras más grandes, proporcionando diferentes maneras de asegurar que las partes de un sistema se ajusten de manera eficiente y efectiva.
3. **Patrones de Comportamiento**: Se ocupan de la interacción y distribución de responsabilidades entre objetos y clases, proporcionando soluciones para la interacción entre objetos y cómo se dividen las responsabilidades, ayudando a asegurar que los componentes de un sistema interactúen de manera eficiente y coherente.

## Patrones de Diseño Creacionales

Los patrones de diseño creacionales se enfocan en los mecanismos de creación de objetos, buscando crear objetos de manera adecuada a la situación. Los principales patrones de diseño creacionales son: Singleton, Factory Method, Abstract Factory, Prototype y Builder.

1. **Singleton:** El patrón Singleton asegura que una clase tenga solo una instancia, mientras proporciona un punto de acceso global a esta. Este patrón se usa frecuentemente en situaciones donde se requiere un control riguroso sobre las variables globales. Por ejemplo, puede ser útil en el manejo de conexiones a bases de datos, donde solo se requiere una única conexión compartida por todo el sistema.
2. **Factory Method:** El patrón Factory Method proporciona una interfaz para crear objetos en una superclase, pero permite a las subclases alterar el tipo de objetos que se crearán. Este patrón se utiliza cuando una clase no puede anticipar el tipo de objetos que necesita crear y desea delegar esta responsabilidad a sus subclases.
3. **Abstract Factory:** El patrón Abstract Factory ofrece una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas. Este patrón se utiliza cuando un sistema debe ser independiente de cómo se crean, componen y representan sus productos, y cuando se desea que un conjunto de productos relacionados se use en conjunto.
4. **Prototype:** El patrón Prototype se usa para crear objetos duplicando un prototipo en lugar de llamar a un constructor. Este patrón es útil cuando la creación de un objeto es costosa o compleja, pero se necesita una copia del mismo.
5. **Builder:** El patrón Builder se utiliza para construir objetos complejos paso a paso. Permite separar la construcción de un objeto complejo de su representación, de manera que el mismo proceso de construcción pueda crear diferentes representaciones. Se utiliza comúnmente cuando se necesita crear diferentes tipos de objetos complejos que tienen un algoritmo de creación común.

## Patrones de Diseño Estructurales

Los patrones de diseño estructurales se centran en cómo se componen los objetos y las clases para formar estructuras más grandes. Los patrones estructurales principales son: Adapter, Bridge, Composite, Decorator, Facade, Flyweight y Proxy.

1. **Adapter:** El patrón Adapter permite que una interfaz existente sea utilizada como otra interfaz. Es especialmente útil para hacer que las clases existentes trabajen con otras sin modificar sus interfaces, es decir, permite la interoperabilidad entre interfaces incompatibles.
2. **Bridge:** El patrón Bridge separa una abstracción de su implementación, de modo que ambas puedan variar independientemente. Esto es útil cuando se quiere evitar un enlace permanente entre una abstracción y su implementación.
3. **Composite:** El patrón Composite permite tratar a los objetos individuales y las composiciones de objetos de la misma manera. Este patrón se utiliza en casos en los que se necesita representar una jerarquía de objetos.
4. **Decorator:** El patrón Decorator permite agregar responsabilidades a los objetos dinámicamente. Los decoradores proporcionan una alternativa flexible a la herencia para extender la funcionalidad. Este patrón es útil cuando la extensión por herencia es impracticable.
5. **Facade:** El patrón Facade proporciona una interfaz unificada a un conjunto de interfaces en un subsistema. Facade define una interfaz de nivel más alto que hace que el subsistema sea más fácil de usar. Este patrón se usa para proporcionar una interfaz simplificada a un sistema complejo.
6. **Flyweight:** El patrón Flyweight se utiliza para minimizar el uso de memoria o el costo computacional al compartir la mayor cantidad de datos posible con otros objetos similares. Es útil cuando se necesita una gran cantidad de objetos que tienen una parte de su estado interno en común.
7. **Proxy:** El patrón Proxy proporciona un sustituto o marcador de posición para otro objeto para controlar el acceso a este. Este patrón se utiliza cuando se necesita una referencia versátil o sofisticada a un objeto, por ejemplo, si es necesario agregar seguridad o cualquier otro servicio adicional.

## Patrones de Diseño de Comportamiento

Los patrones de diseño de comportamiento se centran en la comunicación y la asignación de responsabilidades entre objetos y clases. Los patrones de comportamiento principales son: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method y Visitor.

1. **Chain of Responsibility:** El patrón Chain of Responsibility permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si procesar la solicitud o pasarla al siguiente manejador en la cadena. Este patrón se utiliza cuando más de un objeto puede manejar una solicitud y el manejador no se conoce a priori, se debe determinar automáticamente.
2. **Command:** El patrón Command convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Las solicitudes se encapsulan como objetos, permitiendo parametrizar a los clientes con colas, solicitudes y operaciones. Se utiliza cuando se necesita desacoplar el objeto que envía una solicitud del objeto que la recibe.
3. **Interpreter:** El patrón Interpreter representa una gramática dada como una clase de lenguaje y usa una jerarquía de expresiones para interpretar las sentencias de ese lenguaje. Se utiliza cuando hay un lenguaje para interpretar y se puede representar como árboles de sintaxis abstracta.
4. **Iterator:** El patrón Iterator proporciona una forma de acceder a los elementos de un objeto agregado de forma secuencial sin exponer su representación subyacente. Este patrón es útil cuando se quiere proporcionar una forma de recorrer una estructura de datos sin exponer su implementación.
5. **Mediator:** El patrón Mediator define un objeto que encapsula cómo un conjunto de objetos interactúa. Este patrón promueve un acoplamiento débil al evitar que los objetos se refieran entre sí explícitamente, lo que permite variar sus interacciones de manera independiente. Se utiliza cuando se quiere reducir las interdependencias complejas entre un conjunto de objetos relacionados.
6. **Memento:** El patrón Memento proporciona una forma de guardar y restaurar el estado anterior de un objeto sin revelar los detalles de su implementación. Se utiliza cuando se necesita guardar instantáneas del estado de un objeto para poder restaurarlo a ese estado más tarde.
7. **Observer:** El patrón Observer define una dependencia uno a muchos entre objetos, de manera que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente. Este patrón es útil cuando el cambio de estado en un objeto debe afectar a otros, pero no se sabe cuántos objetos deben ser notificados.
8. **State:** El patrón State permite a un objeto alterar su comportamiento cuando su estado interno cambia. Este patrón se utiliza cuando el comportamiento de un objeto depende de su estado y debe cambiar en tiempo de ejecución según ese estado.
9. **Strategy:** El patrón Strategy define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. La estrategia permite que el algoritmo varíe independientemente de los clientes que lo usan. Este patrón se utiliza cuando hay varias variantes de un algoritmo y se debe seleccionar uno de ellos en tiempo de ejecución.
10. **Template Method:** El patrón Template Method define el esqueleto de un algoritmo en una operación, aplazando algunos pasos a las subclases. Este patrón se utiliza cuando hay un algoritmo con pasos que permanecen iguales en todas las situaciones, pero algunos pasos pueden variar según la situación.
11. **Visitor:** El patrón Visitor representa una operación a realizar en los elementos de una estructura de objetos. Visitor te permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera. Este patrón se utiliza cuando se necesitan realizar diversas operaciones sobre un conjunto de objetos de diferentes clases y se quiere evitar "contaminar" sus interfaces con estas operaciones

# Patrón de Diseño “Prototype”

Todos los patrones de diseño Prototype tienen una interfaz común que hace posible copiar objetos incluso cuando sus clases concretas son desconocidas. Los objetos de Prototype pueden producir copias completas ya que los objetos de la misma clase pueden tener acceso a los campos privados de cada uno.

## ****Caso de Estudio: Aplicación de Dibujo Vectorial****

Imagina que estamos creando una aplicación de dibujo vectorial. En esta aplicación, los usuarios pueden crear formas geométricas y luego ajustar, copiar o mover estas formas en su lienzo de trabajo.

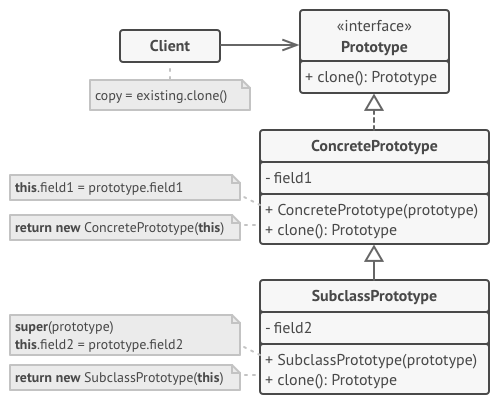
Cada forma en el lienzo tiene muchas propiedades como color, grosor de línea, y ubicación. También, cada forma puede tener una o más subformas. Por ejemplo, una forma de casa podría estar compuesta por un rectángulo (la base de la casa), un triángulo (el techo), y dos círculos (las ventanas).

Ahora, supón que un usuario quiere crear una serie de casas que sean similares pero no idénticas. Podrían querer cambiar el color de las casas, mover las ventanas o hacer que el techo sea más empinado. Sería muy ineficiente tener que construir cada casa desde cero.

Aquí es donde el patrón de diseño Prototype puede ser útil. En lugar de construir cada casa desde cero, el usuario puede crear una "casa prototipo". Esta casa prototipo tiene todas las subformas correctas en las posiciones correctas. Luego, cada vez que el usuario quiera una nueva casa, pueden simplemente clonar el prototipo y hacer las modificaciones necesarias.

Este caso de estudio hipotético muestra cómo el patrón de diseño Prototype puede ser utilizado para hacer que la creación de objetos similares pero no idénticos sea más eficiente y consistente. En esta aplicación de dibujo vectorial, el patrón Prototype ahorra tiempo y esfuerzo a los usuarios y permite un alto grado de personalización y creatividad.

## Implementación Básica | Aplicación del Dibujo Vectorial (Genérico)



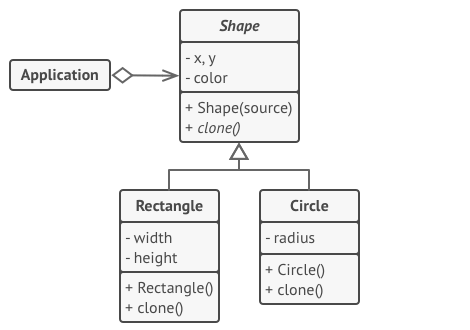
*Imagen #1: Diagrama de Clases | Caso de Estudio*

En un Diagrama de Clases de Diseño (DCD) para el patrón Prototype aplicado en una aplicación de dibujo vectorial, tenemos varias entidades clave que interactúan entre sí para facilitar la funcionalidad de clonación. Ahondaremos en estas entidades, sus roles, y cómo interactúan entre sí.

1. **Interfaz Prototype:** Este es un componente crucial en el patrón de diseño Prototype. La interfaz Prototipo establece los estándares para los objetos clonables, al declarar métodos de clonación que deben ser implementados por cualquier clase que quiera adoptar el patrón Prototype. En muchos casos, esta interfaz contiene un solo método esencial: clonar. Este método garantiza que cualquier objeto que implemente la interfaz Prototipo puede producir una copia de sí mismo.
2. **Clase Prototipo Concreto:** Esta clase es la que implementa la interfaz Prototipo y, por lo tanto, hereda el método de clonación declarado en la interfaz. La implementación del método de clonación implica no solo copiar la estructura del objeto original, sino también manejar algunos aspectos más delicados de la clonación. Por ejemplo, puede ser necesario deshacer dependencias recursivas, clonar objetos vinculados, copiar valores de campos y mucho más. En otras palabras, la clase Prototipo Concreto se encarga de las peculiaridades de la clonación para asegurar que el objeto clonado sea una réplica fiel del objeto original.
3. **Interacción del cliente con el sistema:** En el contexto de la aplicación de dibujo vectorial, el "cliente" puede referirse a la parte del código (o incluso al usuario) que solicita la creación de un nuevo objeto mediante la clonación de un objeto existente. El cliente interactúa con la interfaz de Prototipo, solicitando una copia de un objeto. No necesita saber los detalles de cómo se realiza la clonación; simplemente tiene que interactuar con la interfaz del prototipo, lo que abstrae los detalles subyacentes.
4. **Interacción entre ConcretePrototype (Clase Base) y SubClase ConcretePrototype (Clase Derivada):**
   1. **Herencia de Atributos:** La SubClase ConcretePrototype hereda todos los atributos públicos y protegidos de la Clase ConcretePrototype. Los atributos privados de la Clase ConcretePrototype, sin embargo, no son accesibles directamente desde la SubClase ConcretePrototype.
   2. **Herencia de Métodos:** De manera similar, la SubClase ConcretePrototype hereda todos los métodos públicos y protegidos de la Clase ConcretePrototype. Esto significa que puede invocar estos métodos como si fueran propios. Los métodos privados de la Clase ConcretePrototype no son accesibles desde la SubClase ConcretePrototype.
   3. **Sobrescribir Métodos:** La SubClase ConcretePrototype puede sobrescribir los métodos heredados de la Clase ConcretePrototype. Esto significa que puede proporcionar su propia implementación del método clonar (u otro método) que podría estar adaptada a sus propios requisitos específicos.
   4. **Extensiones:** La SubClase ConcretePrototype puede extender la funcionalidad de la Clase ConcretePrototype al agregar nuevos atributos y métodos que no están presentes en la Clase ConcretePrototype.
   5. **Polimorfismo:** Si un método en la Clase ConcretePrototype es declarado como virtual, la SubClase ConcretePrototype puede proporcionar una implementación diferente para ese método. Esto significa que el método a invocar se decide en tiempo de ejecución, dependiendo del tipo del objeto.
   6. En resumen, la interacción entre la Clase ConcretePrototype y su SubClase ConcretePrototype gira en torno a la herencia, la sobrescritura de métodos, las extensiones y el polimorfismo, lo que proporciona una forma eficaz y flexible de reutilizar y extender el código.

En resumen, el Diagrama de Clases de Diseño para el patrón Prototype en una aplicación de dibujo vectorial consiste en la Interfaz de Prototipo y la Clase de Prototipo Concreto que interactúan para facilitar la clonación de objetos. El cliente interactúa con estas entidades para obtener una copia exacta de un objeto existente, sin tener que preocuparse por los detalles internos de cómo se realiza la clonación.

## Diagrama de Clases | Patrón de Diseño Prototype: Objetos Geométricos



*Imagen #2: Patrón de Diseño Prototype: Objetos Geométricos*

A continuación, se presenta la jerarquía de clases, fundamentada en el Patrón de Diseño Prototype, implementada mediante el Lenguaje de Programación Java.

Esta jerarquía evidencia la estructura sistemática y rigurosa de las clases, así como las relaciones y la interacción entre ellas, todas estas características intrínsecas al mencionado patrón de diseño. La implementación en Java proporciona una demostración práctica y tangible de este patrón, lo que permite una comprensión más profunda y aplicada del mismo.

abstract class Shape {

protected int X;

protected int Y;

protected String color;

public Shape() {}

public Shape(Shape target) {

if (target != null) {

this.X = target.X;

this.Y = target.Y;

this.color = target.color;

}

}

public abstract Shape clone();

}

class Rectangle extends Shape {

private int width;

private int height;

public Rectangle() {}

public Rectangle(Rectangle target) {

super(target);

if (target != null) {

this.width = target.width;

this.height = target.height;

}

}

@Override

public Shape clone() {

return new Rectangle(this);

}

}

class Circle extends Shape {

private int radius;

public Circle() {}

public Circle(Circle target) {

super(target);

if (target != null) {

this.radius = target.radius;

}

}

@Override

public Shape clone() {

return new Circle(this);

}

}

public class Application {

private ArrayList<Shape> shapes = new ArrayList<>();

public Application() {

Circle circle = new Circle();

circle.X = 10;

circle.Y = 10;

circle.radius = 20;

shapes.add(circle);

Circle anotherCircle = (Circle) circle.clone();

shapes.add(anotherCircle);

Rectangle rectangle = new Rectangle();

rectangle.width = 10;

rectangle.height = 20;

shapes.add(rectangle);

}

public void businessLogic() {

ArrayList<Shape> shapesCopy = new ArrayList<>();

for (Shape shape : shapes) {

shapesCopy.add(shape.clone());

}

}

}

# Conclusión o Comentarios Finales

Para concluir, este trabajo ha proporcionado una visión exhaustiva y detallada de los Patrones de Diseño en la Ingeniería de Software, enfocándose en particular en el Patrón de Diseño Creacional Prototype. Este patrón juega un papel crucial en el desarrollo de software, permitiendo la creación eficiente de nuevos objetos mediante la clonación de instancias existentes.

El uso de este patrón puede resultar en ahorros significativos de recursos y tiempo, ya que evita la necesidad de crear nuevas instancias desde cero, en lugar de ello, se clonan a partir de prototipos ya existentes. Esto es especialmente útil cuando la instanciación de un objeto es un proceso costoso en términos de tiempo y recursos.

Además, el Patrón de Diseño Prototype fomenta la reutilización y la flexibilidad en el software, ya que permite generar copias de objetos sin hacer suposiciones sobre su tipo. Este patrón, al reducir la dependencia de las clases concretas, aumenta la modularidad del código, facilitando su mantenimiento y su capacidad de adaptarse a los cambios.

La aplicación práctica del Patrón de Diseño Prototype se evidenció en la implementación en el lenguaje de programación Java. Este caso práctico resaltó las ventajas del patrón y cómo puede ser utilizado para mejorar la eficiencia y la estructura del código en un proyecto de desarrollo de software.

En definitiva, los Patrones de Diseño, y en particular el Patrón de Diseño Prototype, representan una herramienta vital en la caja de herramientas de un ingeniero de software. Su uso puede mejorar enormemente la eficiencia, la calidad y la mantenibilidad del código, llevando al desarrollo de software a nuevos niveles de excelencia.

# Referencias Bibliográficas

1. Tutorialspoint. (s.f.). Design Pattern Tutorial. Recuperado de <https://www.tutorialspoint.com/design_pattern/index.htm>
2. SourceMaking. (s.f.). Design Patterns. Recuperado de <https://sourcemaking.com/design_patterns>
3. DZone. (s.f.). Refcardz: Design Patterns. Recuperado de <https://dzone.com/refcardz/design-patterns>
4. Refactoring Guru. (s.f.). Design Patterns. Recuperado de <https://refactoring.guru/design-patterns>
5. Gang of Four (GoF) Design Patterns. (s.f.). Recuperado de <https://www.gofpatterns.com/>
6. Software Design Patterns by Example. (s.f.). Recuperado de <https://www.softwaretestinghelp.com/design-patterns-tutorial-1/>
7. Design Patterns in Java. (s.f.). Recuperado de <https://www.javatpoint.com/design-patterns-in-java>
8. Journal of Object Technology (JOT). (s.f.). Recuperado de <https://www.jot.fm/>
9. Software Engineering Institute (SEI) - Design Patterns. (1994). Recuperado de <https://www.sei.cmu.edu/reports/94tr007.pdf>
10. IBM Developer - Design Patterns. (s.f.). Recuperado de <https://developer.ibm.com/tutorials/j-patterns/>