**Actividad: Análisis de Paradigmas de Programación y Principios de Diseño SOLID**

Los paradigmas de programación son diferentes enfoques para estructurar y escribir programas. Cada uno ofrece un conjunto de herramientas conceptuales que influyen en cómo se organiza el código, se gestiona la complejidad y se resuelven problemas. A continuación, veremos los principales paradigmas de programación y sus características. *Para una mejor disposición las ventajas serán indicadas con un círculo blanco ( ○ ) y las desventajas con un círculo negro ( ● )*

**1. Programación Imperativa**

La programación imperativa es uno de los paradigmas más antiguos y tradicionales. En este paradigma, el programador describe cómo el ordenador debe realizar una tarea, usando una secuencia de comandos o instrucciones. Los lenguajes que siguen este paradigma, como C o Python, permiten controlar el estado del programa manipulando variables y flujos de control (bucles, condicionales, etc.).

* + Permite un control detallado del flujo de ejecución del programa.
  + Fácil de entender para quienes conocen la arquitectura del hardware y la forma en que los procesadores ejecutan instrucciones.
  + La manipulación de estados mutables puede hacer que el código sea difícil de mantener y propenso a errores.
  + Se vuelve complicado manejar sistemas grandes, ya que la interdependencia de variables y estados puede crecer rápidamente.

**2. Programación Orientada a Objetos (POO)**

La POO organiza el código en "objetos", que son entidades que contienen tanto datos como funciones que operan sobre esos datos. Estos objetos se definen usando "clases", que actúan como plantillas o modelos. La POO permite el uso de conceptos como herencia, polimorfismo y encapsulación.

* + **Reutilización**: las clases y objetos pueden ser reutilizados en diferentes partes del programa o en diferentes proyectos, facilitando la creación de bibliotecas reutilizables.
  + **Mantenibilidad**: el código se organiza en entidades independientes (objetos), lo que facilita su comprensión y mantenimiento.
  + **Modularidad**: permite dividir un problema grande en pequeñas unidades lógicas (objetos) que pueden interactuar entre sí.
  + El diseño orientado a objetos puede ser excesivo para proyectos pequeños, generando más complejidad de la necesaria.
  + Puede llevar a una sobreingeniería si no se maneja con cuidado, con muchas clases y jerarquías que no agregan valor real.

**3. Programación Funcional**

La programación funcional se basa en la idea de que las funciones son el pilar fundamental de un programa. En este paradigma, las funciones son tratadas como "ciudadanos de primera clase", lo que significa que pueden ser pasadas como argumentos, devueltas como resultados de otras funciones, y asignadas a variables. Una característica clave es la "inmutabilidad", lo que implica que los datos no cambian su valor después de ser creados.

* + **Predecibilidad**: las funciones puras, que no dependen de estados externos ni tienen efectos secundarios, hacen que el comportamiento del código sea más fácil de predecir y probar.
  + **Paralelización**: como no se depende de variables mutables, es más sencillo ejecutar múltiples tareas en paralelo sin que interfieran entre sí.
  + **Complejidad conceptual**: puede ser más difícil de aprender para programadores acostumbrados a paradigmas imperativos.
  + **Uso de memoria**: la inmutabilidad puede aumentar el consumo de memoria, ya que se crean nuevos objetos en lugar de modificar los existentes.

**4. Programación Lógica**

La programación lógica se basa en definir un conjunto de reglas y hechos, y luego hacer consultas sobre ellos. En lugar de especificar cómo resolver un problema, se especifica qué se desea lograr. Un ejemplo de lenguaje que sigue este paradigma es Prolog.

* + Ideal para problemas que requieren deducción lógica, como los problemas de inteligencia artificial y resolución de acertijos.
  + Se necesita escribir menos código, ya que las consultas ejecutan la lógica definida por las reglas.
  + Menos eficiente cuando se aplica a problemas que no encajan naturalmente en un enfoque lógico.
  + No es adecuado para el desarrollo de aplicaciones comerciales o de propósito general.

**Principios SOLID**

En el diseño de software, los principios SOLID son un conjunto de cinco principios que ayudan a los desarrolladores a crear sistemas que sean fáciles de mantener, extender y adaptar a cambios. Estos principios son aplicables especialmente en la programación orientada a objetos.

**S: Single Responsibility Principle (SRP):** Cada clase debe tener una única responsabilidad. Esto significa que una clase debe enfocarse en hacer una sola cosa, lo que facilita su mantenimiento y reduce la posibilidad de errores al modificar el código. Ejemplo: Imagina una clase Usuario, que maneja tanto la validación de los datos del usuario como el registro en la base de datos. Esta clase estaría violando el SRP. Para cumplir con el principio, deberíamos dividir la responsabilidad en dos clases: una clase ValidacionUsuario que maneje la validación, y una clase RegistroUsuario que maneje la conexión y el guardado en la base de datos.

**O: Open/Closed Principle (OCP):** Las clases deben estar abiertas a la extensión, pero cerradas a la modificación. Esto significa que el comportamiento de una clase debe poder ser extendido sin alterar su código original, generalmente a través de la herencia o la composición. Ejemplo: Supongamos que tenemos una clase Notificacion que envía un mensaje de correo electrónico. En lugar de modificar la clase para soportar notificaciones por SMS o WhatsApp, podemos crear nuevas clases que hereden de Notificacion y sobreescriban el método de envío. Esto permite añadir nuevos comportamientos sin cambiar la clase original.

**L: Liskov Substitution Principle (LSP):** Los objetos de una clase derivada deben poder sustituir a los de su clase base sin alterar el correcto funcionamiento del programa. Esto asegura que las clases derivadas mantengan la coherencia de la funcionalidad ofrecida por sus clases base. Ejemplo: Si tenemos una clase base Ave con un método Volar(), y luego creamos una subclase Pinguino que no puede volar, estaríamos violando el LSP, porque un pingüino no puede sustituir a un ave en general si se espera que todas las aves puedan volar. Una solución sería redefinir el diseño para que no todas las aves tengan el método Volar().

**I: Interface Segregation Principle (ISP):** Los clientes no deben verse obligados a depender de interfaces que no utilizan. Es mejor tener varias interfaces pequeñas y específicas para cada cliente en lugar de una interfaz general y masiva. Ejemplo: Imaginemos una interfaz IVehiculo con métodos EncenderMotor(), Conducir(), y Volar(). Si una clase Auto implementa esta interfaz, se vería obligada a tener el método Volar(), aunque no lo necesite. Para cumplir con el ISP, sería mejor dividir la interfaz en IVehiculoTerrestre y IVehiculoAereo, donde cada clase implementaría solo los métodos que necesita.

**D: Dependency Inversion Principle (DIP):** Los módulos de alto nivel no deben depender de módulos de bajo nivel; ambos deben depender de abstracciones. Esto promueve el uso de interfaces para desacoplar las dependencias. Ejemplo: Si una clase ControladorPago depende directamente de una clase concreta ProcesadorPayPal, se estaría violando el DIP. Para cumplir el principio, ControladorPago debería depender de una interfaz IProcesadorPago en lugar de la clase concreta, permitiendo así cambiar el procesador de pagos (por ejemplo, a ProcesadorMercadoPago) sin modificar el código del controlador.

**Uso de Librerías Externas en C#**

Uno de los beneficios de usar lenguajes modernos como C# es la posibilidad de integrar librerías externas mediante herramientas como NuGet. Las librerías externas permiten extender las capacidades de una aplicación sin tener que escribir todo el código desde cero. Estas pueden abarcar desde librerías de acceso a bases de datos hasta herramientas para generar gráficos o manipular archivos.

* + **Ahorro de tiempo**: permite a los desarrolladores centrarse en la lógica de negocio mientras reutilizan código ya probado para tareas comunes.
  + **Mantenimiento**: las librerías suelen ser mantenidas y mejoradas por comunidades activas, lo que asegura correcciones de errores y nuevas funcionalidades.
  + **Dependencia de terceros**: depender de una librería externa puede ser arriesgado si deja de ser mantenida o tiene problemas de compatibilidad con nuevas versiones del lenguaje o framework.
  + **Sobrecarga innecesaria**: algunas librerías pueden traer más funcionalidades de las que realmente se necesitan, lo que aumenta el tamaño del proyecto y su complejidad.

**Preguntas:**

1. Explica en tus propias palabras la principal diferencia entre la programación funcional y la programación orientada a objetos.
2. ¿Cuál es la importancia del principio de responsabilidad única (SRP) en la programación orientada a objetos? Da un ejemplo.
3. Menciona una ventaja y una desventaja del uso de librerías externas en C#. ¿Por qué es importante evaluar qué librerías usar en un proyecto?
4. Explica cómo se puede aplicar el principio de abierto/cerrado (OCP) en un sistema de notificaciones que soporte múltiples canales (email, SMS, WhatsApp). ¿Por qué este principio es importante para mantener el código flexible?
5. ¿Cómo aplicaría el principio de inversión de dependencias (DIP) en la confección de una librería para un proyecto de C#? Explica cómo este principio mejora la flexibilidad del código.