Detallaremos cada uno de ellos y señalaremos ejemplos dentro del código.

1. Principio de Responsabilidad Única (SRP - Single Responsibility Principle):

Este principio establece que una clase debe tener una única responsabilidad, y dicha responsabilidad debe estar encapsulada en la clase. En el código proporcionado, podemos observar lo siguiente:

- La clase `Usuario` se encarga de representar a un usuario y proporciona métodos relacionados con los usuarios.

- Las clases derivadas (`Cliente`, `Profesional`, `Administrativo`) extienden la clase `Usuario` y añaden responsabilidades adicionales relacionadas con su rol específico.

- La interfaz `Asesoria` también define un único método `analizarUsuario()` que es implementado por las clases concretas.

En este caso, se cumple el principio de Responsabilidad Única, ya que cada clase y cada interfaz tienen una única responsabilidad bien definida.

2. Principio Abierto/Cerrado (OCP - Open/Closed Principle):

Este principio establece que una entidad debe estar abierta para su extensión, pero cerrada para su modificación. En el código proporcionado, no se aplica explícitamente este principio, ya que no existe una estructura que permita la extensión de las clases sin modificar el código existente.

Para aplicar este principio, podríamos considerar lo siguiente:

- Crear una interfaz o clase abstracta que represente a un usuario genérico y defina el método `analizarUsuario()`.

- Luego, implementar clases concretas como `Cliente`, `Profesional`, `Administrativo`, etc., que extiendan la clase abstracta o implementen la interfaz.

- Esto permitiría agregar nuevos tipos de usuarios sin modificar directamente el código existente.

3. Principio de Sustitución de Liskov (LSP - Liskov Substitution Principle):

Este principio establece que los objetos de una clase derivada deben poder sustituirse por objetos de su clase base sin alterar el correcto funcionamiento del programa. En el código proporcionado, se cumple el principio de Sustitución de Liskov, ya que las clases derivadas (`Cliente`, `Profesional`, `Administrativo`) se pueden utilizar en lugar de la clase base (`Usuario`) sin problemas.

4. Principio de Segregación de Interfaces (ISP - Interface Segregation Principle):

Este principio establece que una clase no debe depender de interfaces que no utiliza. En el código proporcionado, se cumple el principio de Segregación de Interfaces, ya que las interfaces están diseñadas de manera específica y las clases implementan solo los métodos que necesitan. La interfaz `Asesoria` define el método `analizarUsuario()`, que todas las clases implementan según sea necesario.

5. Principio de Inversión de Dependencias (DIP - Dependency Inversion Principle):

Este principio establece que los módulos de alto nivel no deben depender de módulos de bajo nivel, sino que ambos deben depender de abstracciones. Se cumple parcialmente el principio de Inversión de Dependencias, ya que la clase `Listado` depende de la interfaz `Asesoria`, que es una abstracción.

Sin embargo, el código aún podría mejorarse aplicando este principio utilizando dependencias y separando la creación de objetos de su uso. El código cumple en su mayoría con los principios SOLID, pero podría aplicar el principio Abierto/Cerrado y mejorar la aplicación del principio de Inversión de Dependencias mediante el uso de inyección de dependencias.