Para analizar los principios SOLID vamos a coger como ejemplo todo el paquete de TenistaService

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

Este principio establece que una clase debe tener una única responsabilidad, es decir, que solo debe tener una razón para cambiar. En tu código, cada clase parece tener un propósito bien definido:

* TenistaServiceImpl: Su responsabilidad es gestionar la lógica de negocio relacionada con los tenistas, como obtener, actualizar, eliminar, cargar y almacenar información. No se encarga de cómo se guardan los datos ni de cómo se cachean, esas responsabilidades están delegadas.
* repository: Se encarga de interactuar con la capa de datos (probablemente la base de datos).
* cache: Gestiona el almacenamiento temporal de datos en memoria.
* storageCsv y storageJson: Se encargan de gestionar la persistencia de datos en archivos CSV y JSON respectivamente.

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

Este principio dicta que una clase debe estar abierta para la extensión, pero cerrada para la modificación. TenistaServiceImpl está diseñada de manera que se podría extender el comportamiento (por ejemplo, añadiendo otro tipo de almacenamiento como XML) sin modificar la clase en sí, simplemente inyectando nuevas dependencias. Los servicios como storageCsv o storageJson están inyectados, lo que permite cambiar el comportamiento del sistema sin modificar la implementación existente, solo añadiendo nuevas implementaciones de almacenamiento.

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

Este principio establece que los objetos de una clase deben poder ser reemplazados por instancias de sus subtipos sin alterar la funcionalidad del programa. En este caso, TenistaRepository o CacheTenistaImpl tienen implementaciones alternativas que cumplen con las mismas interfaces, el código debería seguir funcionando sin problemas. Por ejemplo, si TenistaRepository tiene una versión para base de datos y otra para una API externa, ambas deben comportarse de manera similar al ser usadas en TenistaServiceImpl.

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

El ISP sugiere que una clase no debe depender de interfaces que no utiliza. El servicio TenistaServiceImpl no se preocupa por los detalles internos de cada dependencia, solo por los métodos que necesita para interactuar con cada uno de ellos (como getById(), save(), load() y store()). Si hubiera una sola interfaz gigante que TenistaServiceImpl tuviera que implementar, violaría este principio.

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

Este principio sugiere que las clases de alto nivel no deben depender de las clases de bajo nivel, sino de abstracciones. la clase TenistaServiceImpl depende de interfaces, como TenistaRepository, CacheTenistaImpl, StorageCsv, y StorageJson. Al depender de estas abstracciones, puedes cambiar las implementaciones subyacentes sin cambiar la lógica del servicio. Esto sigue el DIP correctamente, ya que puedes cambiar fácilmente las implementaciones de repositorio o almacenamiento sin modificar la clase TenistaServiceImpl.