Parcial práctico

David Felipe Gil Laverde 20232020081

Universidad Distrital Francisco José de Caldas

Programación Avanzada

Lilia Marcela Espinosa Rodríguez

Facultad de ingeniería

Bogotá, Colombia

2024

Introducción

En el mundo de la programación, especialmente en la programación orientada a objetos, es crucial contar con un diseño óptimo. Este diseño no solo define cómo se estructuran nuestras clases, sino que también asegura que nuestro código respeta los principios de diseño, lo que resulta en una mayor sostenibilidad a largo plazo.

Este ensayo describe los principios de diseño de software aplicados en el parcial práctico, que consisten en desarrollar una página que solicita datos a un usuario y genera una rutina de gimnasio en una segunda página. Además, se presentarán los diagramas de clases, casos de uso y paquetes del programa, destacando su relevancia en la creación de software de alta calidad.

Principios utilizados:

Para lograr un correcto funcionamiento del programa y un diseño óptimo y sostenible, se usaron los cinco principios de diseño de software más conocidos y utilizados por los desarrolladores. Estos principios, conocidos como SOLID, fueron formulados para guiar a los programadores en la creación de software más eficiente, comprensible y mantenible. Cada uno de los principios SOLID aborda aspectos clave del diseño orientado a objetos y promueve prácticas que minimizan el riesgo de errores y facilitan las modificaciones futuras.

1. **Single Responsibility Principle (SRP):** Toda clase debería tener una sola responsabilidad: esta debería tener un solo propósito en el sistema, y solo debería haber una razón para cambiarla.

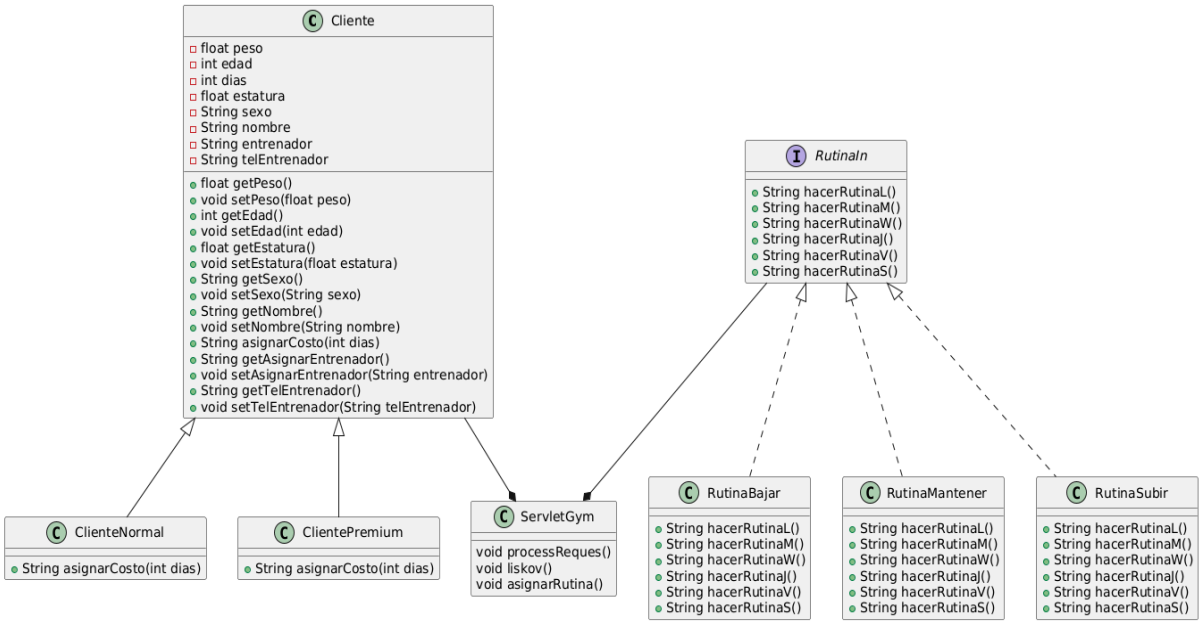
2. **Open/Closed Principle (OCP):** El código debería estar abierto a la extensión pero cerrado a la modificación. Si tenemos un buen diseño de código no tenemos que cambiar tanto código para agregar nuevas funciones.

3. **Liskov Substitution Principle (LSP):** Los objetos de una clase derivada deben poder sustituir a los objetos de la clase base sin afectar el funcionamiento del programa.

4. **Interface Segregation Principle (ISP):** Es mejor contar con muchas interfaces para clientes específicos que una sola interfaz general para muchos tipos de clientes.

5. **Dependency Inversion Principle (DIP):** Los módulos de alto nivel no deben depender de módulos de bajo nivel, sino de abstracciones.

Diagrama de clases:



Observando el diagrama de clases es más sencillo identificar los cinco principios:

1. Single Responsibility: Clases como Cliente, “ClienteNormal”, “ClientePremium” : Cada clase se ocupa exclusivamente de gestionar la información y el comportamiento de sus respectivos tipos de clientes. “ClienteNormal” y “ClientePremium” manejan diferencias en los costos y otras propiedades, pero cada una tiene su propia lógica.

Clases de rutina (“RutinaBajar”, “RutinaMantener”, “RutinaSubir”): Cada clase se centra en proporcionar una rutina de ejercicios específica para el objetivo del cliente. Esto significa que si hay un cambio en la lógica de cómo se debe definir una rutina, solo se tendrá que modificar la clase correspondiente.

2. Open/Closed Principle (OCP): Uso de interfaces: Al definir la interfaz “RutinaIn”, se pueden crear nuevas clases que implementen esta interfaz (por ejemplo, “RutinaCulturista”) sin necesidad de modificar las clases existentes. Esto permite extender el comportamiento del sistema sin alterar la funcionalidad ya implementada.

Nueva funcionalidad: Si se desea agregar un nuevo tipo de cliente o una nueva rutina, se hace simplemente creando nuevas clases que implementen o extiendan las existentes, manteniendo el resto del sistema intacto.

3. Liskov Substitution Principle (LSP): Las clases “ClienteNormal” y “ClientePremium” son subclases de Cliente. Estas dos se pueden utilizar en lugar de Cliente en cualquier contexto que requiera un objeto de tipo Cliente, sin causar problemas. Por ejemplo, si se llama a “asignarCosto()” en un objeto Cliente, debe funcionar correctamente independientemente de si es un “ClienteNormal” o un “ClientePremium”.

4. Interface Segregation Principle (ISP): El diseño utiliza la interfaz “RutinaIn” para definir el comportamiento de las clases de rutina. Cada clase de rutina implementa sólo los métodos que necesita, evitando que implementen métodos innecesarios. Por ejemplo, si más tarde se decide agregar otra rutina, solo se implementa la interfaz según se requiera.

5. Dependency Inversion Principle (DIP): Las clases de rutina dependen de la interfaz “RutinaIn”, en lugar de depender de implementaciones concretas. Esto significa que cualquier cambio en las implementaciones específicas de rutina no afectará a las clases que las utilizan. Si por cualquier motivo hay cambiar la lógica de “RutinaBajar”, no hay necesidad de modificar las clases que interactúan con ella, siempre que sigan cumpliendo con la interfaz.

Finalmente se expondrán los diagramas de paquetes y de casos de uso.

Diagrama de paquetes:
En este diagrama se presentan cuatro paquetes. El primero, llamado logic, contiene dos subpaquetes que abarcan toda la lógica del programa. El primer subpaquete, cliente, incluye las clases que definen el comportamiento de los usuarios del gimnasio, diferenciando entre usuarios premium y normales. El segundo subpaquete, rutina, abarca todas las rutinas de ejercicios disponibles en el gimnasio. Por último, el paquete servletGym utiliza las funcionalidades de ambos subpaquetes dentro de logic

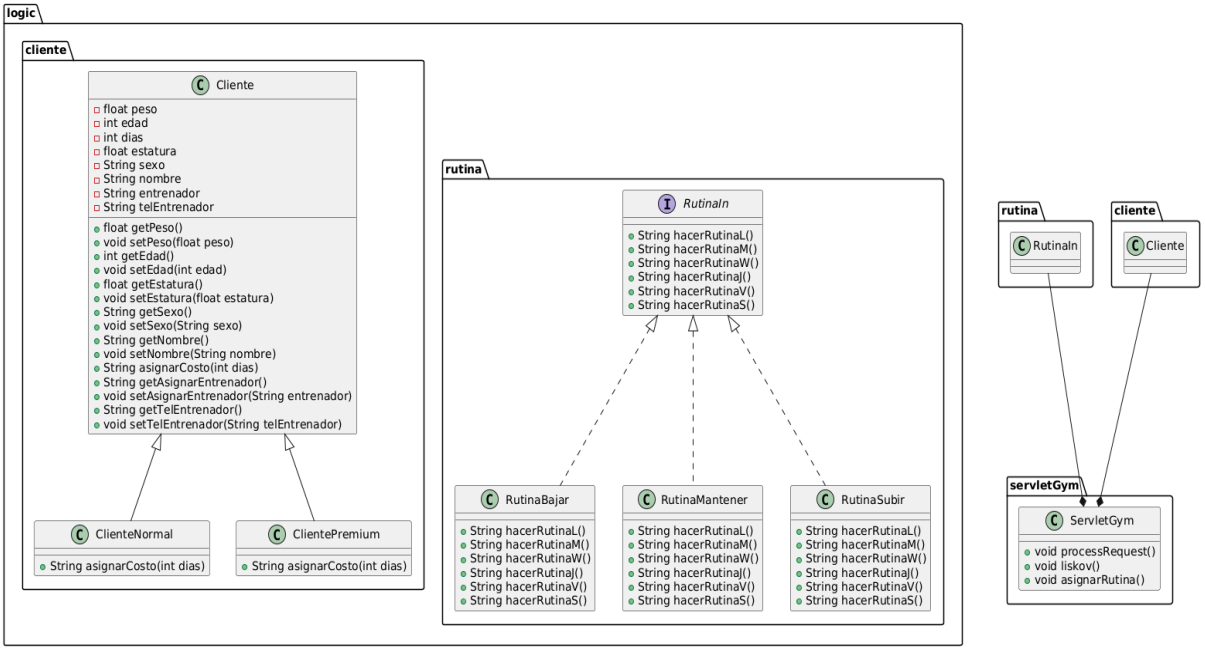


Diagrama de casos de uso:
En este diagrama tenemos un actor que simula el comportamiento del usuario con el programa de inscripción al gimnasio. El diagrama muestra una secuencia en la que como primer paso el usuario completa el formulario de inscripción, luego el programa analiza los datos y con base a lo que el usuario escogió se asigna una rutina y un costo. Finalmente se muestra la rutina y el costo de la rutina en la página dos.