UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE

SEDE SANTO DOMINGO DE LOS TSÁCHILAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN - DCCO-SS

CARRERA DE INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

PERIODO	:	202450 mayo– septiembre 2024
ASIGNATURA	:	/Programación Orientada a Objetos
TEMA	:	Investigación silabo parcial 3
ESTUDIANTE	:	López De La Cruz Rayson Steve
NIVEL-PARALELO - NRC:		Segundo A NRC: 15279
DOCENTE	:	Ing. Verónica Martínez C., Mgs.
FECHA DE ENTREGA	:	29/07/2024

SANTO DOMINGO – ECUADOR

I. Introducción

II. 2. Objetivos

2.1. Objetivo General:

Investigar sobre temas del silabo Parcial 3

2.2. Objetivos Específicos:

- Investigar temas del silabo del Parcial 3 con ejemplos de cada uno
- Parafraseo de temas, comprensión del tema, con analogías.

III. 3. Desarrollo / Marco Teórico/ Práctica

3.1. 22. Principios SOLID

SOLID Son un conjunto de cinco principios de diseños de sotware que promueven la creación de código limpio, modular y fácil de mantener

Acuñado por Michael Feathers, basándose en los principios de la programación orientada a objetos.

¿Cuales son los objetivos que hay que tener en cuenta a escribir código?

- Crear un **software eficaz**: que cumpla con su cometido y que sea **robusto y estable**.
- Escribir un **código limpio y flexible** ante los cambios: que se pueda modificar fácilmente según necesidad, que sea **reutilizable** y **mantenible**.
- Permitir **escalabilidad**: que acepte ser ampliado con nuevas funcionalidades de manera ágil.

S(SRP)Single Responsibility Principle

El patrón Singleton resuelve dos problemas al mismo tiempo, vulnerando el Principio de responsabilidad única:

O(OCP)Open- Close Principle

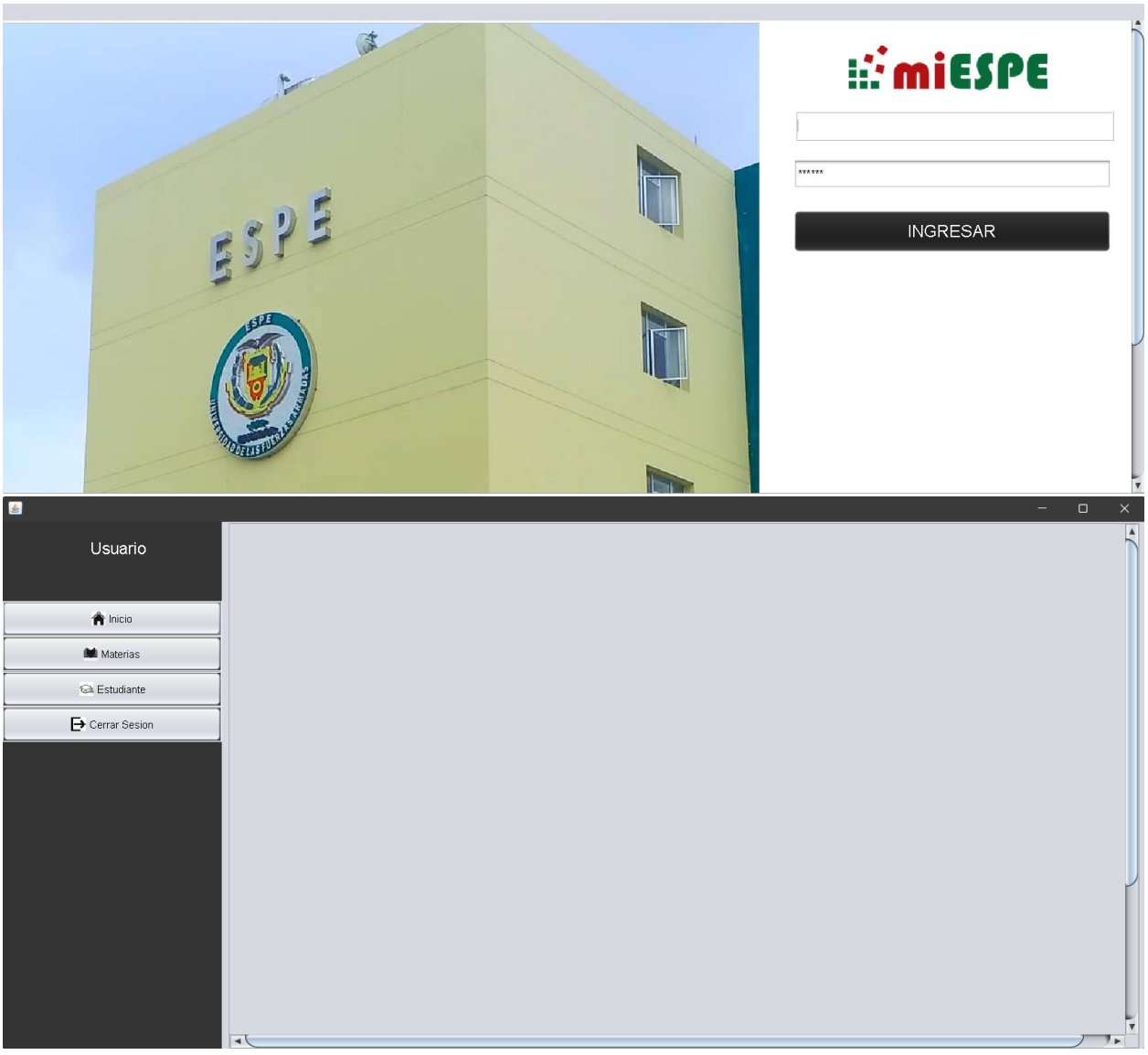
Este principio establece que las entidades de software(clases,módulos,funciones

L(LSP) Liskov Subtitution Principle

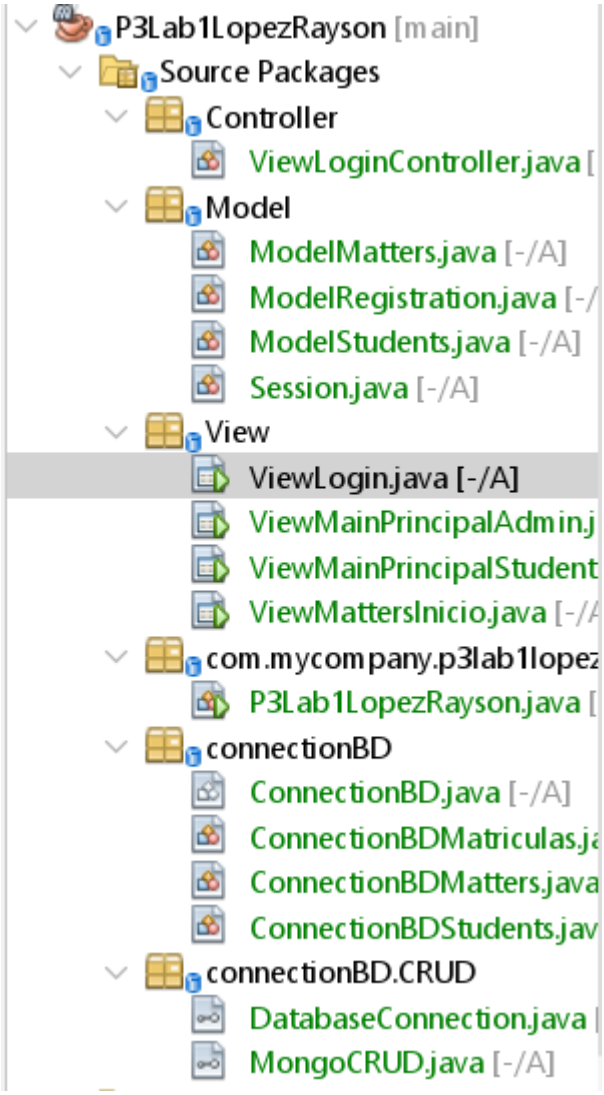
I(ISP) Interface Segregation Principle

D(DIP) Dependency Inversion Principle

Interfaces creadas



Separacion en Modelo Vista Controlador



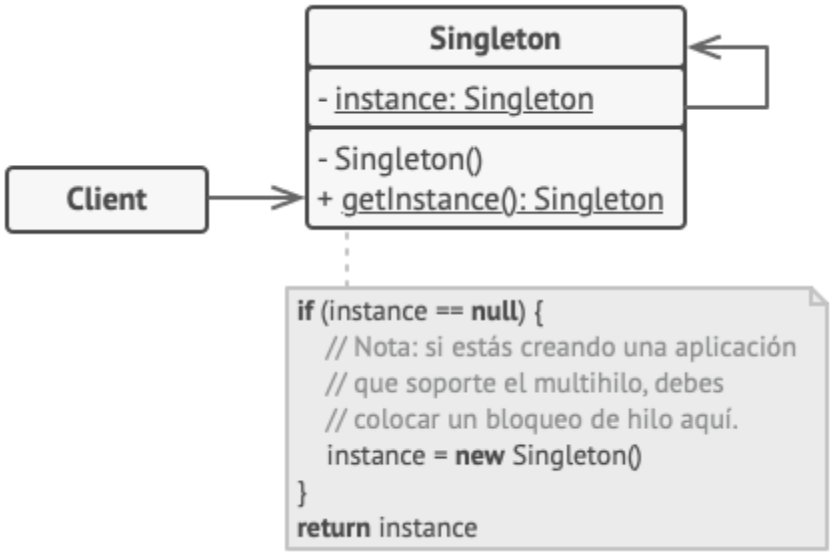
3.2. 22.1 Single Responsibility

Singleton

Permite asegurarnos de que una clase, tenga una sola única instancia, a la vez, que proporciona un punto de acceso global a dicha instancia.
Controlar el acceso, de una una clase.

(Ejemplo, conexión de la base de datos)

El patrón singleton, o singleton pattern en inglés, pertenece a la categoría de **patrones creativos** dentro del grupo de los patrones de diseño. También se le conoce simplemente como “singleton”. El propósito de este patrón es evitar que sea creado más de un objeto por clase. Esto se logra creando el objeto deseado en una clase y recuperándolo como una instancia estática. El singleton es uno de los **patrones más simples**, pero **más poderosos** en el desarrollo de software.



EJEMPLO SINGLETON

```
public class ConnectionBDStudents extends ConnectionBD
    private MongoClient mongoClient;
    private MongoDatabase database;
    private MongoCollection<Document> collection;
    static ConnectionBDStudents instance;
```

Creamos una ConnectionBDStudents que es una instancia de la clase Connection

Y creamos los getter de la instancia

```
public static ConnectionBDStudents getInstance() {
    if(instance==null){
        instance= new ConnectionBDStudents();
    }
    return instance;
}
```

¿Qué significa?
si la instancia no ha sido inicializada, o, está vacía, el programa pregunta si se inicio o no, si no se inició, lo regresa creado

3.3. 22.2 Open/Closed

El segundo principio de SOLID lo formuló Bertrand Meyer en 1988 en su libro “Object Oriented Software Construction” y dice: “Deberías ser capaz de extender el comportamiento de una clase, sin modificarla”. En otras palabras: las clases que usas deberían estar abiertas para poder extenderse y cerradas para modificarse.

En su blog Robert C. Martin defendió este principio que a priori puede parecer una paradoja. Es importante tener en cuenta el Open/Closed Principle (OCP) a la hora de desarrollar clases, librerías o frameworks.

EJEMPLO

(Un método de calculo)

3.4. 3.1.3 Liskov Sustitución

El principio establece que, si una clase base tiene ciertos comportamientos y propiedades, sus clases derivadas deben ser capaces de heredar y utilizar esos comportamientos y propiedades sin cambiar el correcto funcionamiento del programa.

El Principio de Sustitución de Liskov afirma que los objetos de una clase derivada deben ser sustituibles por objetos de la clase base sin alterar el funcionamiento del programa. Este principio asegura que una clase derivada puede sustituir a su clase base sin introducir errores.

Ejemplo: Si tienes una clase base Ave con un método volar, y una clase derivada Pingüino que no puede volar, entonces Pingüino no debería heredar de Ave si el método volar es esencial para Ave.

3.5. 22.4 Interfaz Segregación

Este principio nos dice que una clase nunca debe extender de interfaces con métodos que no usa, por el principio de segregación de interfaces busca que las interfaces sean lo más pequeñas y específicas posible de modo que cada clase solo implemente los métodos que necesita.

Ejemplo: Si tienes una interfaz Trabajador con métodos cobrarSalario y tomarVacaciones, no deberías forzar a una clase Contratista que no recibe salario a implementar cobrarSalario. En su lugar, podrías tener interfaces separadas como Empleado y Vacacionable.

VEHICULOS MOTO AUTOMOVIL

Métodos

Acelera-retroceder-frenar

3.6. 22.5 Dependencia Inversión

El principio de inversión de dependencias indica que las clases de un sistema deben depender de las abstracciones/interfaces y no de las implementaciones concretas. Esto significa que las clases

no deben depender directamente de clases específicas, sino de interfaces o clases abstractas. Esto lo haremos inyectando dependencias en el constructor de la clase, pero estas dependencias serán interfaces o clases abstractas, no clases finales.

El módulo de altos nivel , no deben depender de módulos de bajo nivel.

Utilizar un JSON, si es que la base de datos falla, hay el respaldo de JSON

High-level modules should not depend on low-level modules. Both should depend on abstractions.
Abstractions should not depend on details. Details should depend on abstractions.

Robert C. Martin

EJEMPLO

La creación de métodos abstractos, para la posterior implementación.

3.7. 22.6 Código entendible, flexible y mantenible

Para lograr un código que sea entendible, flexible y mantenible, se deben seguir buenas prácticas de programación y principios como SOLID. Esto incluye escribir código claro y bien documentado, utilizar patrones de diseño apropiados, y asegurarse de que el código sea fácil de probar y modificar.

Ejemplo:

- **Entendible:** Utilizar nombres de variables y métodos descriptivos.
- **Flexible:** Diseñar el código para que pueda adaptarse a cambios futuros sin grandes refactorizaciones.
- **Mantenible:** Asegurarse de que el código esté bien estructurado y modularizado, facilitando su actualización y corrección de errores.

IV. 4. Conclusiones

Comprensión Profunda:

- La investigación y el estudio detallado de los temas del sílabo del Parcial 3 han permitido una comprensión profunda y precisa de los conceptos clave. La reescritura y parafraseo de la información, junto con el uso de analogías y ejemplos prácticos, han facilitado la asimilación y retención de la materia.

Aplicabilidad Práctica:

- Proveer ejemplos claros y aplicables ha demostrado ser una estrategia efectiva para entender cómo los conceptos teóricos se pueden utilizar en situaciones reales. Esta práctica no solo ayuda a comprender mejor la teoría, sino que también prepara para la aplicación práctica de los conocimientos adquiridos.

 Preparación Eficaz:

- La combinación de investigación, parafraseo, y ejemplos detallados ha resultado en una preparación eficaz para el Parcial 3. Este enfoque ha asegurado que los temas sean bien entendidos y se puedan aplicar con confianza durante el examen y en situaciones prácticas futuras.

V. 5. Recomendaciones

VI. 6. Bibliografía/ Referencias

Bibliografía

Ferrer, B. (2024, febrero 13). Principios SOLID: (4) Interface Segregation Principle. *Secture*. https://secture.com/blog/principios-solid-interface-segregation-principle/

Janssen, T. (2018, marzo 28). *SOLID design principles explained: The open/closed principle with code examples*. Stackify. https://stackify.com/solid-design-open-closed-principle/

Principio de Inversión de dependencias. (2017, junio 22). Web development. https://miguelgarciaponceblog.wordpress.com/2017/06/22/principio-de-inversion-de-dependencias/

Singleton. (s/f). Refactoring.guru. Recuperado el 29 de julio de 2024, de +https://refactoring.guru/es/design-patterns/singleton

Vicente, J. (2023, abril 16). *Principios SOLID VI: Principio de inversión de dependencias*. DEV Community. https://dev.to/josavicente/principios-solid-vi-principio-de-inversion-de-dependencias-eea

VII. 7. Anexos:

VIII. 8. Legalización de documento

Nombres y Apellidos:

CI:

Firma: