UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE SEDE SANTO DOMINGO

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN - DCCO-SS

CARRERA DE INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

PERIODO	:	Abril 2023 – Octubre 2023
ASIGNATURA	:	POO
TEMA	:	GUIAS DE LAS CODIFICACIONES
NOMBRES	:	Ordoñez Eduardo
		Kelvin Quezada
		Riera Michael
		Sánchez Lander
NIVEL-PARALELO	:	Segundo “A”
DOCENTE	:	Ing. Cevallos Farias Javier Moyota
FECHA DE ENTREGA	:	11/08/2023

Grupo 4

SANTO DOMINGO - ECUADOR

2023

Guía patrones de diseño

Guía del ejercicio #1: Patrón de diseño creacional abstract factory

En este programa hacemos uso del patrón de diseño, tenemos relaciones entre sí, teniendo 2 paquetes. En el primer paquete hacemos uso de las clases y en el segundo es donde se desarrolla el patrón de diseño.

El patrón de diseño abstract factory permite la creación, la instanciación de objetos de distintas clases que tienen métodos en común, por ejemplo las clases producto que se derivan en, “ServicioDesign”, “ServicioSoftwareEducacional” y “ServicioWebsite”. Según este patrón de diseño se tienen siempre factorías que se pueden considerar como clases fabrica que permiten la instanciación de los objetos correspondientemente. Poseemos una factoría abstracta que es “ServicioFactory”, esta crea un método abstracto “CrearServicio()” que nos devuelve “ServicioInformatico” y de esta interfaz se derivan 3 factorías completas “DesignFactory”, “SoftwareFactory” y “WebsiteFactory” que se encargan de instanciar objetos de las clases producto, es decir, DesignFactory nos va a crear objetos de la clase ServicioDesign y así sucesivamente, estas devuelven un servicio informático.

ServicioInformatico

Es una interfaz tipo abstracta que contiene 3 métodos “asignarTrabajo”, ”indicarFechaEntrega” e “informarSobrePago()”, estos 3 métodos están implementados en los 3 productos.

Servicio de diseño: Debe tener los 3 métodos presentes, asignar el trabajo a quienes deben hacerlo, que indiquen la fecha de entrega del trabajo e informar el detalle del pago a realizar.

Las relaciones son las siguientes, las factorías concretas implementan la interfaz de la factoria abstracta y los productos concretos implementan la interfaz del producto abstracto y esta es relación es la de dependencia por instanciación.

Los métodos que deben ser implementados por las factorías concretas aparecerán en la factoría abstracta y los métodos que deben ser implementados por los productos concretos aparecen en el producto abstracto.

Los productos concretos implementan la interfaz ServicioInformatico que es el producto abstracto “ServicioDesign”, “ServicioSoftwareEducacional” y “ServicioWebsite”.

Posee un menú en el que están las opciones sobre los servicios, se hace la llamada del método usarServicio al que se le pasa una factoría concreta dependiendo de lo que ingreso el usuario, se hace uso de polimorfismo y el método recibe una factoría abstracta.

Factory es la referencia que puede recibir objetos de las clases que implementen la interfaz implementando sus métodos correspondientes, se crea un servicioInformatico que contiene objetos de cualquiera de los servicios concreto se recibe en la referencia “servicio”, dependiendo de que factoría haya pasado desde el main, se va a instanciar un tipo de producto u otro independientemente del tipo de producto, los métodos se ejecutaran con normalidad porque todos los productos tienen estos métodos ya que implementan al producto abstracto.

Guía Ejercicio 2: Patrón de diseño estructural Adapter

El programa contiene 2 paquetes para su funcionamiento y tenemos como punto de partida la clase abstracta motor y es extendida por 3 clases hijas, "MotorComun", "MotorElectricoAdapter", "MotorEconomico". La clase abstracta motor contiene 3 métodos, encender, acelerar y apagar. Contiene una clase "MotorElectrico" que se va a adaptar, esta contiene métodos que no son compatibles con el target, por lo tanto, no puede heredar estas se conocen como adaptive y se adapta a través del adapter según lo que se defina en el target. Haciendo este proceso las 3 clases pueden heredar de motor, porque ya son compatibles, entonces la clase principal llamada "Aplicación" ya puede hacer uso de motor utilizando polimorfismo.

- **Clase Abstracta "Motor**" es la clase target que define los métodos abstractos que quieren adecuarse y puede ser interfaz, debe contener métodos abstractos que se quieren adecuar, puede ser heredada a partir de ella se pueden obtener clases hijas.

El adapter es MotorElectricoAdapter, el adapter da un cuerpo a sus metidos abstractos que aparecen definidos en el target haciendo uso de la referencia que tiene de la clase que se va a adaptar, cuando se pida encender al "motorElectricoAdapter" llamara los métodos conectar y activar de la clase MotorElectrico que se está adaptando, cuando se llame al método acelerar lo que hará será llamara al método que corresponde con el método acelerar, cuando llame al método apagar, llamara al método detener y desconectar, sirviendo como puente adaptador para ser compatible MotorElectrico con motor porque directamente no son compatibles.

- **Clase Aplicación**

Se hace uso de las clases implementadas, se tiene 2 atributos privados y estáticos, se crea una referencia que recibe objetos, instancias de las clases que sin son concretas, es decir, "MotorComun", "MotorElectricoAdapter", "MotorEconomico" sin problema ya que son clases hijas de la clase abstracta "Motor", la referencia es de la clase abstracta y los objetos de las clases hijas.

En el main se presenta un menú que permite el ingreso de opciones, opciones de encender el motor presentando los 3 tipos, motor común, motor económico y motor eléctrico mas la opción de salir, dependiendo de lo seleccionado se va a instanciar la referencia motor haciendo uso del

polimorfismo ya que puede adoptar cualquiera de las 3 formas y luego se llamara a usarMotor();
El método usarMotor, llama a encender, acelerar y apagar.

Guía Ejercicio 3: Patrón de diseño de comportamiento Strategy

El patrón estrategia nos permite definir familias de algoritmos, encapsularlos y hacerlos intercambiables. Permite que los algoritmos sean independientes de los clientes que están usándolo. En este caso, se utiliza para implementar diferentes estrategias de análisis de antivirus. Se utiliza este patrón de diseño para permitir que las diferentes estrategias de análisis de antivirus se implementen de manera independiente y se puedan intercambiar fácilmente en tiempo de ejecución. Cada estrategia encapsula un conjunto particular de algoritmos y comportamientos para realizar el análisis de antivirus.

Se realizó un ejercicio que simula un análisis de un antivirus, tenemos un antivirus simple y un antivirus avanzado.

Interfaz estrategia

Esta es una interfaz que define el método analizar(), que representa la operación común a todas las estrategias de análisis de antivirus.

Análisis simple

Es una clase abstracta que corresponde en implementar la interfaz “Estrategia” en el cual se va a indicar operaciones o pasos a seguir que determinan que es un análisis simple como iniciar el análisis, saltarme los archivos zip y detener el análisis. Para ello se han definido tres métodos abstractos iniciar(), saltarzip() y detener() que otra clase que viene a ser una implementación de algún antivirus en particular va a utilizar.

Análisis avanzado

Esta también es una clase abstracta que implementa la interfaz “Estrategia”. Aquí creamos otros métodos abstractos como iniciar(), analizar memoria(), analizar key loggers(), analizar rootkits(), descomprimir zips() y detener(), es decir tiene una funcionalidad un poco más elaborada en comparación al análisis simple.

Antivirus simple

Esta clase hereda la clase “análisis simple” para poder sobrescribir sus métodos como iniciar, saltar zip, y detener cada uno de ellos con su respectiva lógica de programación. Aquí también, se imprimen mensajes y se simula el tiempo de espera.

Antivirus avanzado

Similar a Antivirus simple, esta clase hereda de “análisis avanzado” y proporciona implementaciones concretas para los métodos abstractos definidos en la clase base. Cada método imprime un mensaje y simula un tiempo de espera usando Thread.sleep() para simular la ejecución de tareas de análisis.

Contexto

Aquí se abstrae la utilización del algoritmo como una estrategia cambiable. Creamos una variable de la interfaz estrategia y en su constructor se va a recibir cualquier implementación de la interfaz para que simplemente con su método ejecutar podamos establecer el comportamiento respectivo de analizar. Es decir, que el método ejecutar() va a llamar al método analizar() de la estrategia asociada. Esto permite que las estrategias se intercambien y se ejecuten de manera flexible sin cambiar el código del contexto.

Clase principal

Esta es la clase principal que contiene el método main en donde se importan las librerías necesarias y se crea una instancia de “contexto”. Además, se pasa una implementación concreta que vendría a ser un antivirus simple, sin embargo, también podemos cambiar la palabra “Análisis simple” por “Análisis avanzado” para ejecutar la otra estrategia de análisis, esto depende del tipo de estrategia que desee usar el usuario.