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	:	Patrones de Diseño
NOMBRES	:	Ordoñez Eduardo Kelvin Quezada Riera Michael Sanchez Lander
NIVEL-PARALELO	:	Segundo “A”
DOCENTE	:	Ing. Cevallos Farias Javier Moyota
FECHA DE ENTREGA	:	11/08/2023

Grupo 4

SANTO DOMINGO - ECUADOR

2023

Contenido

1. Introducción 4
2. Objetivos 5
    2.1.1. Objetivos Generales 5
    2.1.2. Objetivos Específicos 5
3. Desarrollo 5
3.1. Que es patrones de diseños design patterns 5
3.2. Tipos de patrones de diseños 6
    3.2.1. Patrones de Creacionales 7
    3.2.2. Patrones estructurales 8
    3.2.3. Patrones de comportamiento 9
3.3. Ejemplo de Abstract Factory 10
    3.3.1. Planteamiento de Problema 10
    3.3.2. Solución del Problema 10
    3.3.3. Diagrama de Clase 11
    3.3.4. Diagrama de Uml 12
    3.3.5. Diagrama de Secuencia 13
    3.3.6. Código del Ejercicio 13
    3.3.7. Anexos de Código 18
    3.3.8. Anexos de Ejecución del Programa 22
3.4 Ejemplo de Patrón de Diseño de Comportamiento: Strategy 23
    3.4.1 Planteamiento de Problema 23
    3.4.2 Solución del Problema 23
    3.4.3 Diagrama de Clase 24
    3.4.4 Código del Ejercicio 25
    3.4.5 Anexos de Código 29
    3.5. Ejemplo de patrón de diseño estructural: Adapter 35
    3.5.1 Planteamiento del problema 35
    3.5.2 Solución del Problema 35
    3.4.3 Diagrama de Clase 36
    3.5.4 Código del Ejercicio 37
    3.5.5 Anexos del código 41
4. Conclusiones 47
5. Recomendaciones 47
6. Bibliografía 48

Imagen 1. Tipos de Patrones de Diseños..... 6
Imagen 2. Diagrama de clase por Kelvin Quezada.....10
Imagen 3. Diagrama de Uml por Kelvin Quezada.....11
Imagen 4. Diagrama de secuencia por Kelvin Quezada..... 12
Imagen 5. Interface ServicioFactory por Kelvin Quezada.....17
Imagen 6. ServicioFactory por Kelvin Quezada.....17
Imagen 7. Clase DesignFactory por Kelvin Quezada.....17
Imagen 8. SercicioftwareEducacional por Kelvin Quezada..... 18
Imagen 9. Clase ServicioTrabajo por Kelvin Quezada.....18
Imagen 10. Clase ServicioWebsites por Kelvin Quezada.....19
Imagen 11. SoftWareFactory por Kelvin Quezada..... 19
Imagen 12. WebsiFactory Por Kelvin Quezada.....20
Imagen 13. Clase Principal por Kelvin Quezada.....20
Imagen 14. Ejecución 1 Diseño Gráfico por Kelvin Quezada.....21
Imagen 15. Ejecución 2 Software educacional por Kelvin Quezada.....21
Imagen 16. Ejecución 3 Sitio Web por Kelvin Quezada.....22
Imagen 17. Diagrama de Clase por Lánder Sánchez..... 23
Imagen 18. Clase Principal por Lánder Sánchez..... 28
Imagen 19. Clase Estrategia por Lánder Sánchez.....28
Imagen 20. Clase Contexto por Lánder Sánchez..... 29
Imagen 21. Clase Análisis simple por Lánder Sánchez.....30
Imagen 22. Clase Antivirus simple por Lánder Sánchez.....30
Imagen 23. Clase Análisis avanzado por Lánder Sánchez..... 31
Imagen 24. Clase Antivirus avanzado por Lánder Sánchez..... 32
Imagen 25. Ejecución 1 por Lánder Sánchez.....33
Imagen 26. Ejecución 2 por Lánder Sánchez.....33
Imagen 27. Diagrama de clases por Eduardo Ordoñez.....35

Imagen 28. Clase Motor por Eduardo Ordoñez..... 40

Imagen 29. Clase MotorComun por Eduardo Ordoñez..... 41

Imagen 30. Clase MotorEconomico por Eduardo Ordoñez.....42

Imagen 31. Clase MotorElectrico por Eduardo Ordoñez.....43

Imagen 32. Clase MotorElectricoAdapter por Eduardo Ordoñez.....44

Imagen 33. Clase Aplicacion (principal) por Eduardo Ordoñez.....44

1. Introducción

Según (Canelo, 2020) nos dice que los patrones de diseños son soluciones general, reutilizables y se aplica en diferentes tipos de problemas del diseño del Software ya que se trata de plantillas que identifica un problema en el sistema y proporciona soluciones ya que los desarrolladores se enfrentan durante un periodo de tiempo a las pruebas y errores.

En este informe hablaremos sobre los patrones de diseño, son soluciones probadas y comunes para problemas recurrentes en el desarrollo de software. Se dividen en tres categorías: creacionales, estructurales y de comportamiento.

Su objetivo es mejorar la flexibilidad, modularidad y mantenibilidad del código, evitando repeticiones y mejorando la comunicación entre los elementos del sistema.

2. Objetivos

2.1.1. Objetivos Generales

- Comprender los conceptos y principios fundamentales de los patrones de diseño en programación.

2.1.2. Objetivos Específicos

- Aplicar de manera efectiva los patrones de diseño adecuados para mejorar la calidad y la estructura del código en proyectos de desarrollo de software.
- Utilizar los tres tipos de patrones de diseño como creacional, estructural y comportamiento para establecer relaciones entre objetos y gestionar la comunicación entre ellos.

3. Desarrollo

3.1. Que es patrones de diseños design patterns

Según (Canelo, 2020) los patrones de diseño, también conocidos como design patterns, son soluciones generales y reutilizables aplicables a diversos problemas de diseño de software. Estas plantillas identifican problemas comunes en el sistema y ofrecen soluciones probadas a través de la experiencia acumulada a lo largo del tiempo.

Mediante (Sánchez, 2017) cada patrón de diseño tiene un propósito específico y aborda problemas comunes de diseño. Podemos decir que, si el modelo de solución encontrado se adapta a múltiples ámbitos, entonces nos encontramos ante un posible patrón de diseño de software. Estos modelos de solución son considerados patrones de diseño si su efectividad fue probada resolviendo problemas similares en otras ocasiones. Su reutilización también debe estar probada.

Daremos a conocer las ventajas y desventajas de los patrones de diseño:

Ventajas

- Reutilización de soluciones probadas.
- Mejora de la calidad del código.
- Facilita la comunicación y colaboración en el equipo.

Desventajas

- Puede introducir complejidad innecesaria.
- Existe el riesgo de caer en el sobre diseño.

3.2. Tipos de patrones de diseños

Hay un total de 23 patrones de diseño diferentes, y las tres categorías más populares contienen los patrones de diseño más utilizados.

- patrones en la creación.
- patrones arquitectónicos.
- patrones de comportamiento

Imagen 1. Tipos de Patrones de Diseños

Resumen: *Los patrones de diseños son herramientas que desarrolla a resolver los problemas comunes de manera probada y estructurada ya que promueve las buenas prácticas de programación.* (Canelo, 2020)

3.2.1. Patrones de Creacionales

Los patrones de compilación ofrecen una variedad de mecanismos de creación de objetos que aumentan la flexibilidad y permiten la reutilización adecuada de la situación del código preexistente. Como resultado, el programa tiene más libertad para elegir qué objetos crear para un caso de uso particular.

Estos son los patrones de la creación.

1. **Abstract Factory:** Este patrón utiliza interfaz para crear conjuntos o familias de objetos relacionados sin especificar el nombre de la clase.
2. **Buider Patterns:** Abstrae la construcción de objetos complejos paso a paso, permitiendo diferentes representaciones del mismo proceso de construcción.
3. **Factory Method:** Define una interfaz para crear objetos, pero permite a las subclases decidir qué clase instanciar.
4. **Prototype:** Permite copiar objetos existentes sin hacer que su código dependa de sus clases.
5. **Singleton:** Garantiza que una clase tenga solo una instancia y proporciona un punto global de acceso a ella.

3.2.2. Patrones estructurales

Según (Soto, 2021) el objetivo de los patrones estructurales es facilitar el ensamblaje de elementos de clases estructurales más grandes manteniendo la flexibilidad y la eficiencia.

1. **Adapter:** Adaptador es un patrón que se utiliza para que objetos con interfaces incompatibles colaboren entre sí.
2. **Bridge:** resuelve un problema habitual en la herencia de clases dividiendo clases relacionadas en dos jerarquías diferentes: implementación y abstracción, para que estas puedan desarrollarse independientemente.
3. **Composite:** Solo se recomienda utilizar Composite cuando el modelo de código está creado a partir de un sistema ramificado en forma de árbol.
4. **Decorator:** Se utiliza para extender el comportamiento de un objeto añadiendo funcionalidades al mismo a través de objetos encapsuladores que presentan dichas funcionalidades.

5. **Facade:** Proporciona una interfaz simplificada para una biblioteca, un marco o cualquier otro conjunto complejo de clases.
6. **Flyweight:** Ayuda a reducir el tamaño de los objetos almacenando en su interior solo el estado intrínseco (información constante) del mismo y compartiendo el resto de la información (estado extrínseco) entre varios objetos similares.
7. **Proxy:** Se utiliza para crear objetos que pueden representar funciones de otras clases u objetos y la interfaz se utiliza para acceder a estas funcionalidades.

3.2.3. Patrones de comportamiento

Los patrones de comportamiento buscan resolver la comunicación entre diferentes áreas

1. **Chain of responsibility:** podremos evitar que la petición emitida por un emisor sea acoplada a un solo receptor permitiendo que más de un objeto pueda responder a dicha petición.
2. **Command:** Se utiliza cuando es necesario encapsular dentro de un objeto todos los parámetros que una acción requiere para ejecutarse.
3. **Interpreter:** Utilizando Interpreter podremos evaluar un lenguaje a través de una interfaz que indique el contexto en el cual se interpreta.
4. **Iterator:** Este patrón de comportamiento se utiliza cuando necesitamos iterar en colecciones o conjuntos de objetos sin la necesidad de intercambiar información relevante.
5. **Mediator:** Se utiliza cuando necesitamos controlar las comunicaciones directas entre objetos y disminuir sus dependencias caóticas.

6. **Memento:** Este patrón es capaz de almacenar y restaurar la información de un objeto.
7. **Observer:** A través de este patrón de comportamiento varios objetos interesados (suscriptores) en un objeto en particular (notificador) pueden recibir notificaciones de su comportamiento mientras estén suscriptos a sus notificaciones.
8. **State:** Se utiliza para modificar el comportamiento de una clase de objetos dependiendo del estado actual (comportamiento interno) de dichos objetos.
9. **Strategy:** Permite separar todos los algoritmos de una clase específica en nuevas clases separadas donde los objetos pueden intercambiarse.

3.3. Ejemplo de Abstract Factory

3.3.1. Planteamiento de Problema

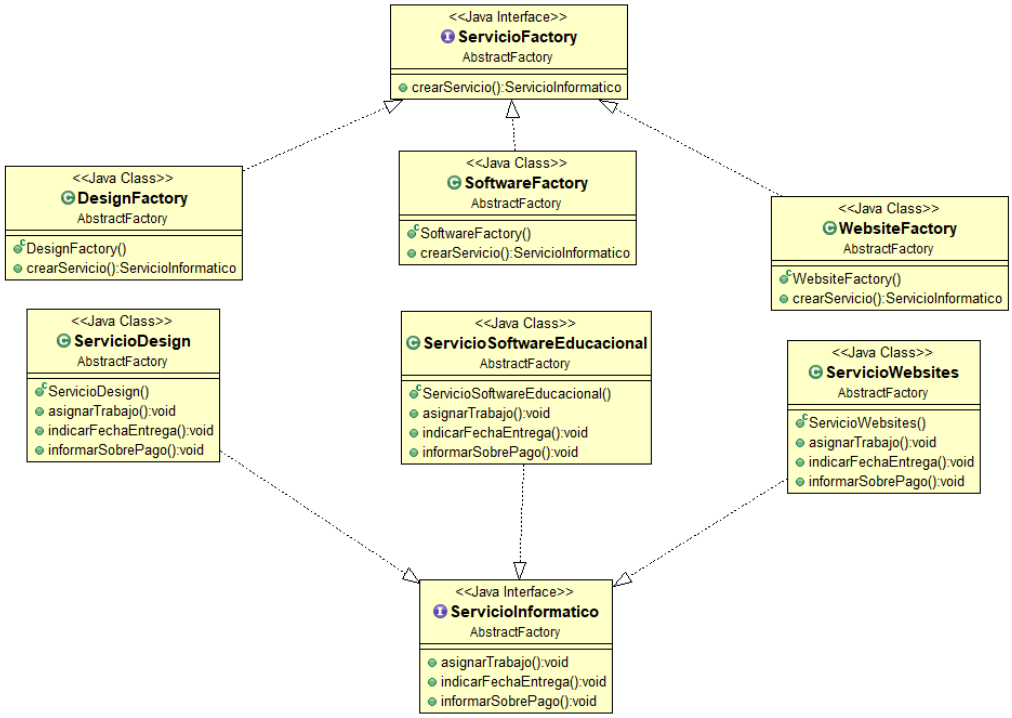
La empresa Nacional necesita realizar un servicio informático de información donde quiere saber sobre un diseño gráfico, un desarrollo web educacional y creación de sitio web indicar el trabajo, la fecha de entrega y saber sobre el pago del servicio.

3.3.2. Solución del Problema

Mediante un software realizaremos un ejercicio de información donde la empresa nacional podrá elegir las opciones de información que requiere saber por medio de interfaz para crear conjuntos o familias de objetos relacionados que ayudan al sitio de ayuda.

3.3.3. Diagrama de Clase

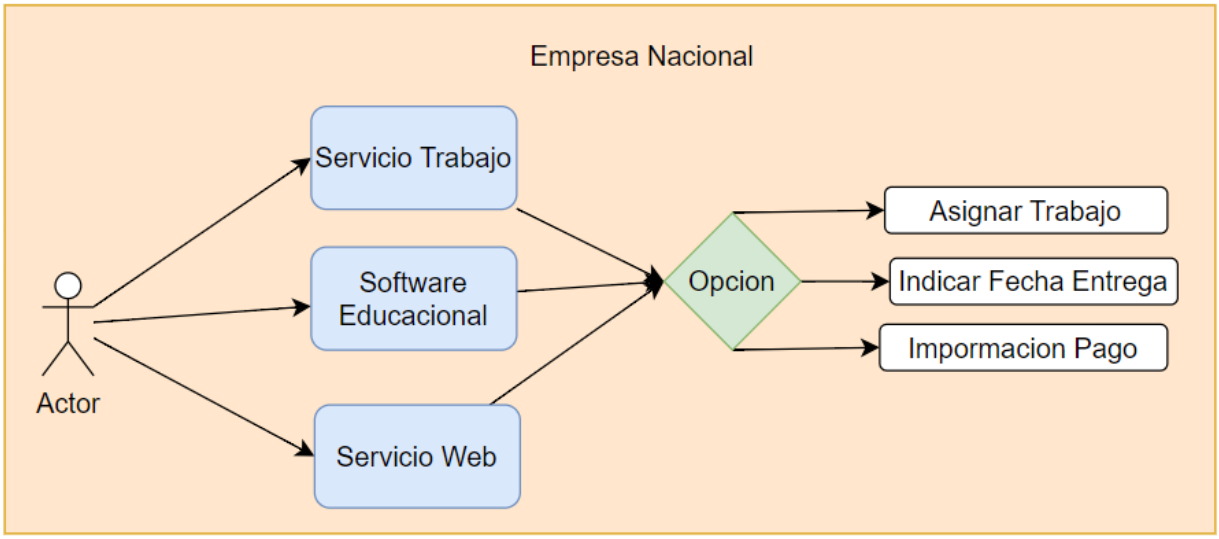
Imagen 2. Diagrama de clase por Kelvin Quezada



En este diagrama se elaboró una estructura para el servicio de información donde el usuario va a elegir tres arias en específica para saber la información que desea saber en las cuales son Servicio de trabajo, servicio de Educación, Servicio de sitio web.

3.3.4. Diagrama de Uml

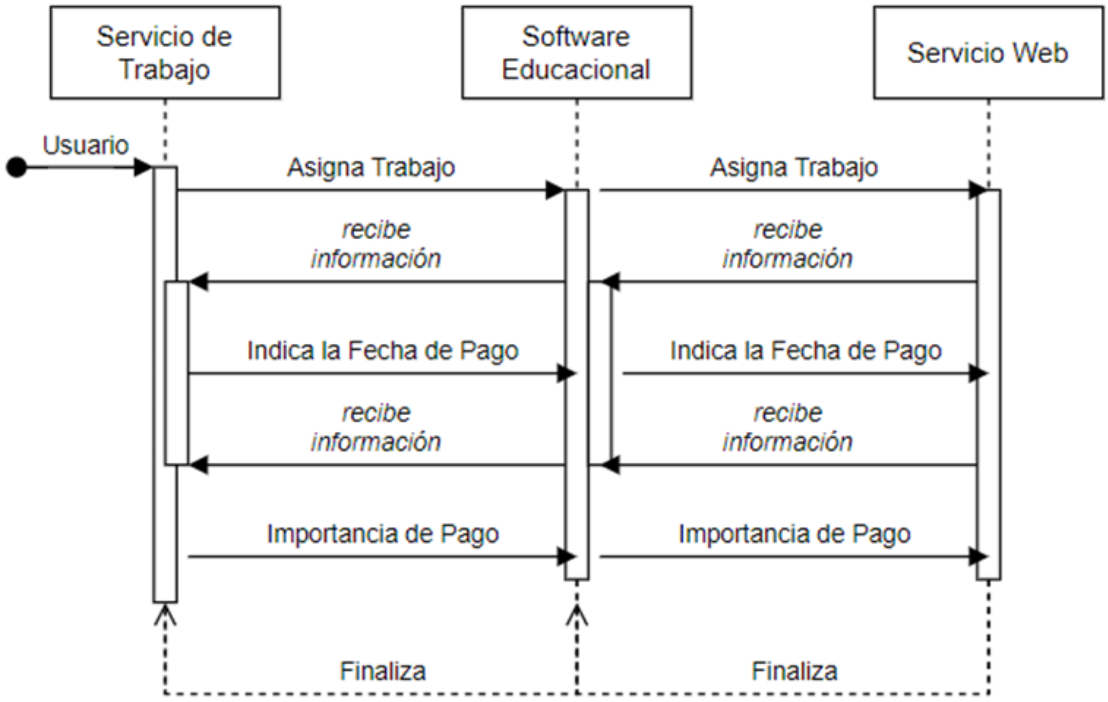
Imagen 3. Diagrama de Uml por Kelvin Quezada



Daremos a conocer un diagrama uml ya que nos ayuda en las interacciones entre el sistema y los actores mediante este diagrama uml nos indica que el usuario de una empresa necesita saber tres informaciones de tres arias en especifica mediante el autor Kelvin Quezada.

3.3.5. Diagrama de Secuencia

Imagen 4. Diagrama de secuencia por Kelvin Quezada



El usuario se va al registro de información ingresa al servicio de trabajo dará a conocer la información del trabajo que se genera, también conocerá la fecha de pago de ese trabajo y la importancia del pago y también conocerá información del software educación y servicio web, ya que nos permite producir familias de objetos relacionados sin especificar sus clases concretas según Kelvin Quezada.

3.3.6. Código del Ejercicio

```
package AbstractFactory;
//los metodos aparecen a los productos abstracto
public interface ServicioInformatico {
    public void asignarTrabajo();
    public void indicarFechaEntrega();
    public void informarSobrePago();
```

```
}
```

```
package AbstractFactory;
//factoria abstracto
public interface ServicioFactory {
// define un método público llamado crearServicio que no acepta ningún argumento y devuelve un objeto del tipo "ServicioInformatico".
   public ServicioInformatico crearServicio();
}
```

```
package AbstractFactory;
//tendremos una clase DesignFactory que sera implementada a la interface ServicioFactory
public class DesignFactory implements ServicioFactory {
//se crea un metodo para sobrescribiendo de la interfaz
//el método crearServicio() está definido en una clase que implementa la interfaz ServicioInformatico
   @Override
   public ServicioInformatico crearServicio() {
      return new ServicioTrabajo();
   }

}
```

```
package AbstractFactory;
//creamos una clase ServicioSoftwareEducacional implements ServicioInformatico ya que son
//interfaces que definen un conjunto de métodos que una clase que las implementa debe proporcionar.
public class ServicioSoftwareEducacional implements ServicioInformatico {
   @Override
   public void asignarTrabajo() {
       System.out.println("Nuestros programadores han sido informados del programa que deben realizar.");
   }

   @Override
   public void indicarFechaEntrega() {
      System.out.println("Se ha fijado como fecha de entrega el dia 01/10/2023.");
   }

   @Override
   public void informarSobrePago() {
       System.out.println("El monto a pagar sera proporcional a la cantidad de estudiantes que haran uso del software.");
   }

}
```

```
package AbstractFactory;

public class ServicioTrabajo implements ServicioInformatico {
```

```
//creamos tres metodos que estan sobrescribiendo es nuestra clase interfaz de ServicioInformatico
   @Override
   public void asignarTrabajo() {
      System.out.println("El trabajado ha sido asignado a disenadores graficos disponibles.");
   }

   @Override
   public void indicarFechaEntrega() {
         System.out.println("Ellos han determinado terminar el trabajo como maximo para el dia 09/08/2023.");
   }

   @Override
   public void informarSobrePago() {
          System.out.println("Debe realizar el pago en efectivo al momento de recoger el logo completamente terminado.");
   }

}
```

```
package AbstractFactory;

public class ServicioWebsites implements ServicioInformatico {
//creamos tres metodos que estan sobrescribiendo es nuestra clase interfaz de ServicioInformatico
   @Override
   public void asignarTrabajo() {
         System.out.println("El personal encargado del desarrollo de sitios web ha aceptado el trabajo.");
   }

   @Override
   public void indicarFechaEntrega() {
            System.out.println("El sitio web con Responsabilida Designara terminado el día 26/11/2023.");
   }

   @Override
   public void informarSobrePago() {
      System.out.println("El monto a pagar no incluye el pago por dominio y hosting.");
   }

}
```

```
package AbstractFactory;

public class SoftwareFactory implements ServicioFactory {
```

```
//se crea un metodo para sobrescribiendo de la interfaz
//el método crearServicio() está definido en una clase que implementa la interfaz ServicioInformatico
   @Override
   public ServicioInformatico crearServicio() {
      return new ServicioSoftwareEducacional();
   }

}
```

```
package AbstractFactory;

public class WebsiteFactory implements ServicioFactory {
//sobrescribiendo un método de la interfaz
   @Override
       //el método crearServicio() está definido en una clase que implementa la interfaz ServicioInformatico
   public ServicioInformatico crearServicio() {
      return new ServicioWebsites();
   }

}
```

```
package Prueba;

import AbstractFactory.DesignFactory;
import AbstractFactory.ServicioFactory;
import AbstractFactory.ServicioInformatico;
import AbstractFactory.SoftwareFactory;
import AbstractFactory.WebsiteFactory;
import java.util.Scanner;

public class Principal {
public static void main(String[] args) {
    Scanner leer = new Scanner(System.in);
    int opcion;
    do {
        System.out.print(
             "MENU DE OPCIONES\n"
           + "---- -- -------->\n"
           + "1. Solicitar servicio de diseno grafico.\n"
           + "2. Solicitar desarrollo de software educacional.\n"
           + "3. Solicitar creacion de sitios web.\n"
           + "4. Cerrar programa.\n"
           + "Seleccione opcion: "
       );
        opcion=leer.nextInt();
        switch (opcion) {
```

```
            case 1:
               usarServicio(new DesignFactory());
               break;
            case 2:
               usarServicio(new SoftwareFactory());
               break;
            case 3:
               usarServicio(new WebsiteFactory());
               break;
            case 4:
               System.out.println("A finalizado el programa ");
               break;
            default:
               System.out.println("A opcion es incorrecta ");

         }

   } while (opcion!=4);
   }
//creamos un metodo pública y estática que recibe un objeto de tipo ServicioFactory como argumento.
//La función se encarga de utilizar el factory para obtener un objeto ServicioInformatico y luego realizar algunas acciones relacionadas con ese servicio.
//https://www.youtube.com/watch?v=xNsPGA7zrVQ

public static void usarServicio(ServicioFactory factory) {
   //el ServicioInformatico es una clase interface que resive informacion
   ServicioInformatico servicio = factory.crearServicio();
      servicio.asignarTrabajo();
      servicio.indicarFechaEntrega();
      servicio.informarSobrePago();
   }
}
```

3.3.7. Anexos de Código

Imagen 5. Interface ServicioFactory por Kelvin Quezada

```
package AbstractFactory;
//los metodos aparecen a los productos abstracto
public interface ServicioInformatico {
    public void asignarTrabajo();
    public void indicarFechaEntrega();
    public void informarSobrePago();
}
```

Aquí se dará a conocer los tres métodos abstracto por Kelvin Quezada

Imagen 6. ServicioFactory por Kelvin Quezada

```
package AbstractFactory;
//factoria abstracto
public interface ServicioFactory {
// define un método público llamado crearServicio que no acepta
//ningún argumento y devuelve un objeto del tipo "ServicioInformatico".
    public ServicioInformatico crearServicio();
}
```

La interface servicioFactory devuelve un objeto del servicioInformacion por Kelvin Quezada

Imagen 7. Clase DesignFactory por Kelvin Quezada

```
package AbstractFactory;
//tendremos una clase DesignFactory que sera implementada a la interface ServicioFactory
public class DesignFactory implements ServicioFactory {
//se crea un metodo para sobrescribiendo de la interfaz
//el método crearServicio() está definido en una clase que implementa la interfaz ServicioInformatico
    @Override
    public ServicioInformatico crearServicio() {
        return new ServicioTrabajo();
    }

}
```

Creamos la clase DesignFacto es donde llamaremos al servicio de información y tiene un retorno de los objetos de la clase ServicioTrabajo por Kelvin Quezada.

Imagen 8. SercicioftwareEducacional por Kelvin Quezada

```java
package AbstractFactory;
//creamos una clase ServicioSoftwareEducacional implements ServicioInformatico ya que son
//interfaces que definen un conjunto de métodos que una clase que las implementa debe proporcionar.
public class ServicioSoftwareEducacional implements ServicioInformatico {
    @Override
    public void asignarTrabajo() {
        System.out.println(x: "Nuestros programadores han sido informados del programa que deben realizar.");
    }

    @Override
    public void indicarFechaEntrega() {
        System.out.println(x: "Se ha fijado como fecha de entrega el dia 01/10/2023.");
    }

    @Override
    public void informarSobrePago() {
        System.out.println(x: "El monto a pagar sera proporcional a la cantidad de estudiantes que haran uso del software.");
    }

}
```

Tendremos la clase serviciosoftwareaeducacional se dará a conocer la sobre escritura de los tres métodos por Kelvin Quezada.

Imagen 9. Clase ServicioTrabajo por Kelvin Quezada

```java
package AbstractFactory;

public class ServicioTrabajo implements ServicioInformatico {
//creamos tres metodos que estan sobrescribiendo es nuestra clase interfaz de ServicioInformatico
    @Override
    public void asignarTrabajo() {
        System.out.println(x: "El trabajado ha sido asignado a disenadores graficos disponibles.");
    }

    @Override
    public void indicarFechaEntrega() {
        System.out.println(x: "Ellos han determinado terminar el trabajo como maximo para el dia 09/08/2023.");
    }

    @Override
    public void informarSobrePago() {
        System.out.println(x: "Debe realizar el pago en efectivo al momento de recoger el logo completamente terminado.");
    }

}
```

Tendremos la clase servicioTrabajo se dará a conocer la sobre escritura de los tres métodos por Kelvin Quezada.

Imagen 10. Clase ServicioWebsites por Kelvin Quezada

```java
package AbstractFactory;

public class ServicioWebsites implements ServicioInformatico {
//creamos tres metodos que estan sobrescribiendo es nuestra clase interfaz de ServicioInformatico
    @Override
    public void asignarTrabajo() {
        System.out.println(x: "El personal encargado del desarrollo de sitios web ha aceptado el trabajo.");
    }

    @Override
    public void indicarFechaEntrega() {
        System.out.println(x: "El sitio web con Responsabilida Designara terminado el día 26/11/2023.");
    }

    @Override
    public void informarSobrePago() {
        System.out.println(x: "El monto a pagar no incluye el pago por dominio y hosting.");
    }

}
```

Tendremos la clase servicioWebsties se dará a conocer la sobre escritura de los tres métodos por Kelvin Quezada

Imagen 11. SoftWareFactory por Kelvin Quezada

```java
package AbstractFactory;

public class SoftwareFactory implements ServicioFactory {
//se crea un metodo para sobrescribiendo de la interfaz
//el método crearServicio() está definido en una clase que implementa la interfaz ServicioInformatico
    @Override
    public ServicioInformatico crearServicio() {
        return new ServicioSoftwareEducacional();
    }

}
```

Creamos la clase sotfwareFactory es donde llamaremos al servicio de información y tiene un retorno de los objetos de la clase ServiciosotfwareEducacional por Kelvin Quezada.

Imagen 12. WebsiFactory Por Kelvin Quezada

```
package AbstractFactory;

public class WebsiteFactory implements ServicioFactory {
//sobrescribiendo un método de la interfaz
    @Override
    //el método crearServicio() está definido en una clase que implementa la interfaz ServicioInformatico
    public ServicioInformatico crearServicio() {
        return new ServicioWebsites();
    }

}
```

Creamos la clase webFactory es donde llamaremos al servicio de información y tiene un retorno de los objetos de la clase ServicioWeb por Kelvin Quezada.

Imagen 13. Clase Principal por Kelvin Quezada

```
public class Principal {
public static void main(String[] args) {
    Scanner leer = new Scanner(source:System.in);
    int opcion;
    do {
        System.out.print(
                "MENU DE OPCIONES\n"
              + "---- -- -------->\n"
              + "1. Solicitar servicio de diseno grafico.\n"
              + "2. Solicitar desarrollo de software educacional.\n"
              + "3. Solicitar creacion de sitios web.\n"
              + "4. Cerrar programa.\n"
              + "Seleccione opcion: "
       );
        opcion=leer.nextInt();
        switch (opcion) {
            case 1:
                usarServicio(new DesignFactory());
                break;
            case 2:
                usarServicio(new SoftwareFactory());
                break;
            case 3:
                usarServicio(new WebsiteFactory());
                break;
            case 4:
                System.out.println(x: "A finalizado el programa ");
                break;
            default:
                System.out.println(x: "A opcion es incorrecta ");

        }

public static void usarServicio(ServicioFactory factory) {
    //el ServicioInformatico es una clase interface que resive informacion
    ServicioInformatico servicio = factory.crearServicio();
        servicio.asignarTrabajo();
        servicio.indicarFechaEntrega();
        servicio.informarSobrePago();
    }
}
```

Tendremos un menú de opciones donde llamaremos nuestros métodos, también creamos un usarServicios donde el servicioInformacion es una clase interfaz que reduce la información de todo los métodos de sobreescritura por Kelvin Quezada.

3.3.8. Anexos de Ejecución del Programa

Imagen 14. Ejecución 1 Diseño Gráfico por Kelvin Quezada

```
run:
MENU DE OPCIONES
---- -- -------->
1. Solicitar servicio de diseno grafico.
2. Solicitar desarrollo de software educacional.
3. Solicitar creacion de sitios web.
4. Cerrar programa.
Seleccione opcion: 1
El trabajado ha sido asignado a disenadores graficos disponibles.
Ellos han determinado terminar el trabajo como maximo para el dia 09/08/2023.
Debe realizar el pago en efectivo al momento de recoger el logo completamente terminado.
```

Tendremos la primera opción que dará a conocer el servicio de diseño gráfico donde llamaremos los tres métodos de la sobreescritura

Imagen 15. Ejecución 2 Software educacional por Kelvin Quezada

```
MENU DE OPCIONES
---- -- -------->
1. Solicitar servicio de diseno grafico.
2. Solicitar desarrollo de software educacional.
3. Solicitar creacion de sitios web.
4. Cerrar programa.
Seleccione opcion: 2
Nuestros programadores han sido informados del programa que deben realizar.
Se ha fijado como fecha de entrega el dia 01/10/2023.
El monto a pagar sera proporcional a la cantidad de estudiantes que haran uso del software.
```

En la segunda opción dar a conocer el servicio de información de desarrollo de software educacional y nos dará a conocer la sobreescritura de los tres métodos por Kelvin Quezada.

Imagen 16. Ejecución 3 Sitio Web por Kelvin Quezada

```
MENU DE OPCIONES
---- -- -------->
1. Solicitar servicio de diseno grafico.
2. Solicitar desarrollo de software educacional.
3. Solicitar creacion de sitios web.
4. Cerrar programa.
Seleccione opcion: 3
El personal encargado del desarrollo de sitios web ha aceptado el trabajo.
El sitio web con Responsabilida Designara terminado el d�a 26/11/2023.
El monto a pagar no incluye el pago por dominio y hosting.
```

En la tercera opción nos dará a conocer la información del sitio web que se aplicó la sobreescritura de los tres métodos por Kelvin Quezada.

3.4 Ejemplo de Patrón de Diseño de Comportamiento: Strategy

3.4.1 Planteamiento de Problema

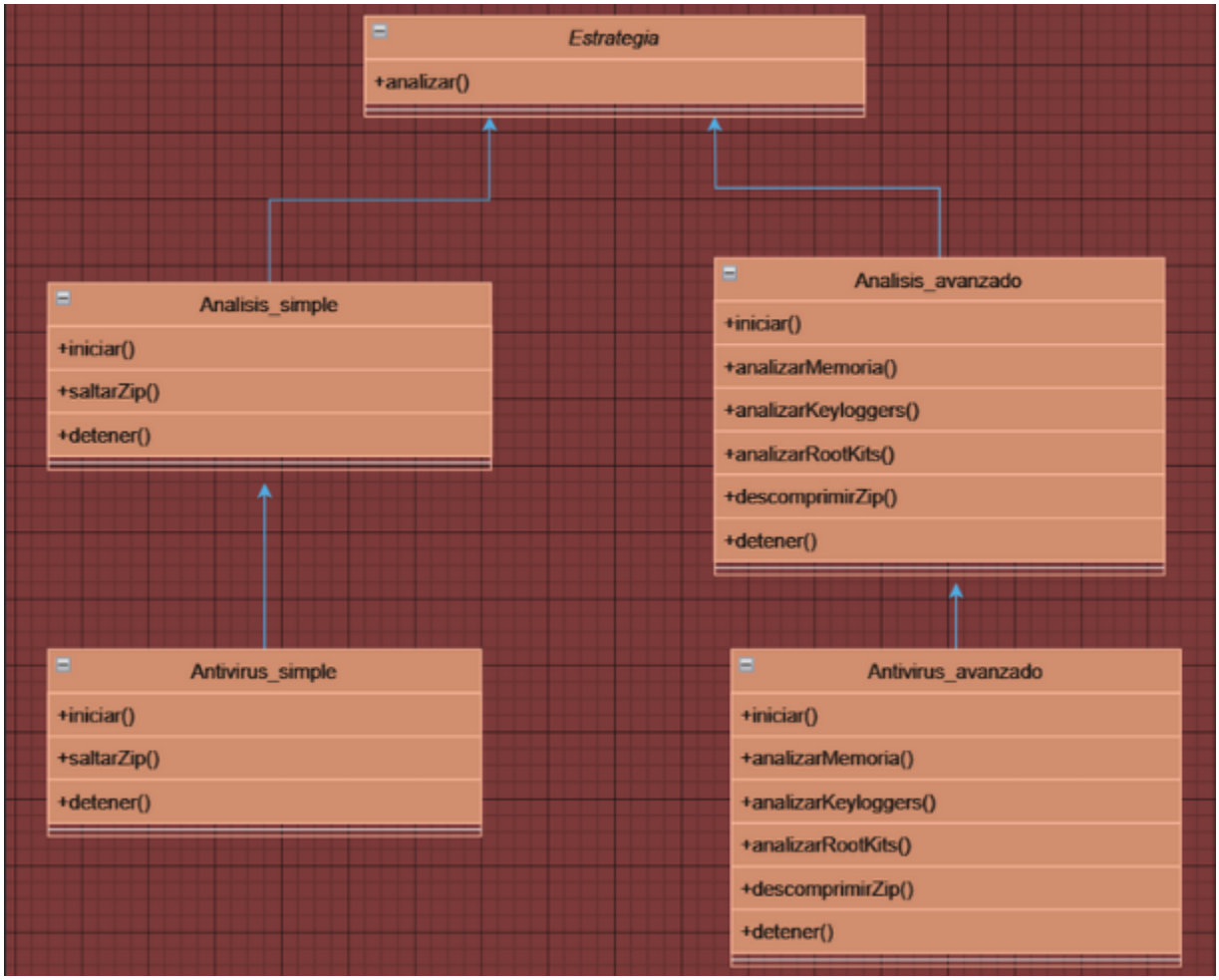
La empresa de seguridad cibernética “CyberShield” busca fortalecer su capacidad de protección contra virus y malware. Necesitan implementar un sistema de análisis de seguridad que ofrezca enfoques flexibles y adaptados a diferentes niveles de riesgo.

3.4.2 Solución del Problema

Mediante un software realizaremos un sistema de análisis de virus donde la empresa de seguridad podrá elegir entre realizar un análisis de antivirus simple o un análisis de antivirus avanzado.

3.4.3 Diagrama de Clase

Imagen 17. Diagrama de Clase por Lánder Sánchez



El diagrama de clases refleja la estructura del patrón de diseño Strategy implementado en el sistema de análisis de antivirus. Permite intercambiar y ejecutar distintas estrategias de análisis. La interfaz `Estrategia` define el contrato para las estrategias de análisis, mientras que las clases abstractas “Analisis_simple” y “Analisis_avanzado” implementan métodos comunes. Las clases concretas “Antivirus_simple” y “Antivirus_avanzado” heredan de las clases abstractas y proporcionan detalles específicos de implementación.

3.4.4 Código del Ejercicio

Clase Principal

```
// Importando las clases necesarias del paquete Strategy
package packagepoo_programa3;

import Strategy.Contexto;
import Strategy.Analisis_avanzado;
import Strategy.Antivirus_avanzado;
import Strategy.Analisis_simple;
import Strategy.Antivirus_simple;

// Definiendo la clase principal
public class Codigo_strategy {
    public static void main(String[] args) {
        // Creando un objeto Contexto con una estrategia de antivirus simple (podemos cambiarla
a una de antivirus avanzada dependiendo de lo que desee el usuario)
        Contexto contexto = new Contexto(new Antivirus_simple());
        // Ejecutando la estrategia seleccionada
        contexto.ejecutar();
    }
}
```

Clase Estrategia

```
// Definiendo la interfaz Estrategia
package Strategy;

public interface Estrategia {
    void analizar(); // Método de la interfaz para realizar el análisis
}
```

Clase Análisis Simple

```
// Definiendo la clase abstracta Analisis_simple que implementa Estrategia
package Strategy;

public abstract class Analisis_simple implements Estrategia {
    public void analizar() {
        iniciar(); // Llamando a los métodos abstractos para los pasos de análisis
        saltarZip();
        detener();
    }
    abstract void iniciar(); // Método abstracto para iniciar el análisis
```

```
    abstract void saltarZip(); // Método abstracto para saltar archivos ZIP
    abstract void detener(); // Método abstracto para detener el análisis
}
```

Clase Antivirus simple

```
// Definiendo la clase Antivirus_simple que extiende Analisis_simple
package Strategy;

public class Antivirus_simple extends Analisis_simple {
    @Override
    void iniciar() {
        System.out.println("Antivirus simple - Análisis simple iniciando....");
    }

    @Override
    void saltarZip() {
        try {
            System.out.println("Analizando....");
            Thread.sleep(5000); // Simulando el tiempo de análisis
            System.out.println("No se pudo analizar los archivos con extensión '.zip'.....Realice un
escaneo avanzado");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    void detener() {
        System.out.println("Antivirus simple - Análisis simple a finalizado");
    }
}
```

Clase Análisis Avanzado

```
// Definiendo la clase abstracta Analisis_avanzado que implementa Estrategia
package Strategy;

public abstract class Analisis_avanzado implements Estrategia {
    public void analizar() {
        iniciar(); // Llamando a los métodos abstractos para los pasos de análisis
        analizarMemoria();
        analizarKeyloggers();
        analizarRootKits();
        descomprimirZip();
        detener();
```

```
    }
    abstract void iniciar(); // Método abstracto para iniciar el análisis
    abstract void analizarMemoria(); // Método abstracto para analizar la memoria
    abstract void analizarKeyloggers(); // Método abstracto para analizar keyloggers
    abstract void analizarRootKits(); // Método abstracto para analizar rootkits
    abstract void descomprimirZip(); // Método abstracto para descomprimir archivos ZIP
    abstract void detener(); // Método abstracto para detener el análisis
}
```

Clase Antivirus Avanzado

```
// Definiendo la clase Antivirus_avanzado que extiende Analisis_avanzado
package Strategy;

public class Antivirus_avanzado extends Analisis_avanzado {
    @Override
    void iniciar() {
        System.out.println("Antivirus avanzado - Análisis avanzado iniciando....");
    }

    @Override
    void analizarMemoria() {
        try {
            System.out.println("Analizando Memoria RAM....");
            Thread.sleep(7000); // Simulando el tiempo de análisis
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    void analizarKeyloggers() {
        try {
            System.out.println("Analizando en busca de KeyLoggers....");
            Thread.sleep(3000); // Simulando el tiempo de análisis
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    void analizarRootKits() {
        try {
            System.out.println("Analizando en busca de RootKits....");
            Thread.sleep(4000); // Simulando el tiempo de análisis
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
    }

    @Override
    void descomprimirZip() {
        try {
            System.out.println("Analizando los archivos '.zip'....");
            Thread.sleep(10000); // Simulando el tiempo de análisis
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    void detener() {
        System.out.println("Antivirus avanzado - Análisis avanzado a finalizado");
    }
}
```

Clase Contexto

```
// Definiendo la clase Contexto
package Strategy;

public class Contexto {
    private Estrategia estrategia;

    public Contexto(Estrategia estrategia) {
        this.estrategia = estrategia;
    }

    public void ejecutar() {
        this.estrategia.analizar(); // Llamando al método de análisis de la estrategia seleccionada
    }
}
```

3.4.5 Anexos de Código

Imagen 18. Clase Principal por Lánder Sánchez

```
package packagepoo_programa3;

import Strategy.Contexto;
import Strategy.Analisis_avanzado;
import Strategy.Antivirus_avanzado;
import Strategy.Analisis_simple;
import Strategy.Antivirus_simple;

public class Codigo_strategy {
public static void main(String[] args) {
  Contexto contexto = new Contexto(new Antivirus_simple());
  contexto.ejecutar();


    }
}
```

En la clase “main”, se crean instancias de `Contexto` con diferentes estrategias (“Antivirus_simple” y “Antivirus_avanzado”) y se ejecutan los análisis correspondientes.

Imagen 19. Clase Estrategia por Lánder Sánchez

```
package Strategy;

public interface Estrategia {
 void analizar();
}
```

Es una interfaz que define el método “analizar()”, que será implementado por las clases concretas de estrategias.

Imagen 20. Clase Contexto por Lánder Sánchez

```
package Strategy;

public class Contexto {

 private Estrategia estrategia;

 public Contexto(Estrategia estrategia) {
 this.estrategia = estrategia;
 }
 public void ejecutar() {
 this.estrategia.analizar();
 }
}
```

Es una clase que recibe una estrategia en su constructor y tiene un método “ejecutar()”, que llama al método “analizar()” de la estrategia proporcionada.

Imagen 21. Clase Análisis simple por Lánder Sánchez

```
package Strategy;

public abstract class Analisis_simple implements Estrategia {
 public void analizar(){
 iniciar();
 saltarZip();
 detener();
}
 abstract void iniciar();
 abstract void saltarZip();
 abstract void detener();

}
```

Esta es una clase abstracta que implementa la interfaz `Estrategia`. Define el flujo de análisis simple en los métodos “analizar()”, “iniciar()”, “saltarZip()” y “detener()”.

Imagen 22. Clase Antivirus simple por Lánder Sánchez

```
package Strategy;

public class Antivirus_simple extends Analisis_simple {

 @Override
 void iniciar() {
System.out.println(x: "Antivirus simple - Análisis simple iniciando....");
 }

 @Override
 void saltarZip() {
 try {
 System.out.println(x: "Analizando....");
 Thread.sleep(millis:5000);
 System.out.println(x: "No se pudo analizar los archivos con extensión '.zip'.....Realice un escaneo avanzado");
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 @Override
 void detener() {
 System.out.println(x: "Antivirus simple - Análisis simple a finalizado");
 }
}
```

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.

Imagen 23. Clase Análisis avanzado por Lánder Sánchez

```
package Strategy;

public abstract class Analisis_avanzado implements Estrategia {

    public void analizar() {
iniciar();
analizarMemoria();
analizarKeyloggers();
analizarRootKits();
descomprimirZip();
detener();
}
abstract void iniciar();
abstract void analizarMemoria();
abstract void analizarKeyloggers();
abstract void analizarRootKits();
abstract void descomprimirZip();
abstract void detener();
}
```

Es una clase abstracta similar a “Analisis_simple”, pero define un flujo de análisis más detallado con métodos “analizarMemoria()”, “analizarKeyloggers()”, “analizarRootKits()” y “descomprimirZip()”.

Imagen 24. Clase Antivirus avanzado por Lánder Sánchez

```
package Strategy;

public class Antivirus_avanzado extends Analisis_avanzado {

@Override
void iniciar() {
System.out.println(x: "Antivirus avanzado - Análisis avanzado iniciando....");
}

@Override
void analizarMemoria() {
try {
System.out.println(x: "Analizando Memoria RAM....");
Thread.sleep(millis:7000);
} catch (InterruptedException e) {
e.printStackTrace();
}
}

@Override
void analizarKeyloggers() {
try {
System.out.println(x: "Analizando en busca de KeyLoggers....");
Thread.sleep(millis:3000);
} catch (InterruptedException e) {
e.printStackTrace();
}
}

@Override
void analizarRootKits() {
try {
System.out.println(x: "Analizando en busca de RootKits....");
Thread.sleep(millis:4000);
} catch (InterruptedException e) {
e.printStackTrace();
}
}

@Override
void descomprimirZip() {
try {
System.out.println(x: "Analizando los archivos '.zip'....");
Thread.sleep(millis:10000);
} catch (InterruptedException e) {
e.printStackTrace();
}
}

@Override
void detener() {
System.out.println(x: "Antivirus avanzado - Análisis avanzado a finalizado");
}
}
```

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.

Imagen 25. Ejecución 1 por Lánder Sánchez

```
------------------< codigo_strategy:codigo_strategy >-------------------
Building codigo_strategy 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.1.0:exec (default-cli) @ codigo_strategy ---
Antivirus simple - Análisis simple iniciando....
Analizando....
No se pudo analizar los archivos con extensión '.zip'.....Realice un escaneo avanzado
Antivirus simple - Análisis simple a finalizado
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

```
------------------< codigo_strategy:codigo_strategy >-------------------
Building codigo_strategy 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.1.0:exec (default-cli) @ codigo_strategy ---
Antivirus avanzado - Análisis avanzado iniciando....
Analizando Memoria RAM....
Analizando en busca de KeyLoggers....
Analizando en busca de RootKits....
Analizando los archivos '.zip'....
Antivirus avanzado - Análisis avanzado a finalizado
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Imagen 26. Ejecución 2 por Lánder Sánchez

En la ejecución del programa podemos ver la simulación tanto del análisis del antivirus simple como del antivirus avanzado, en donde en el primero se hace un análisis breve y en el segundo ya simula realizar más pasos demostrando que se trata de un análisis más detallado y complejo.

3.5. Ejemplo de patrón de diseño estructural: Adapter

3.5.1 Planteamiento del problema

En una empresa automotriz, dando órdenes para que determinados motores se activen. Sin embargo, se ha adquirido un nuevo motor y su programa para funcionamiento es distinto al que manejamos hasta entonces.

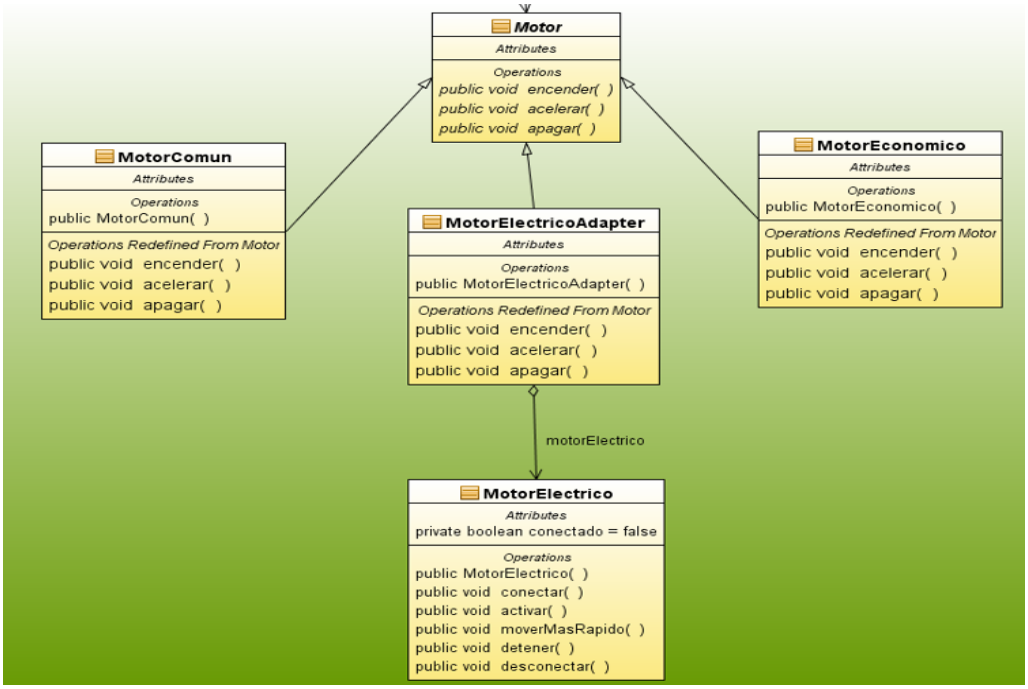
Sucede que no podemos modificar el código fuente de tal clase, sólo hacer uso de ella, pero no es compatible con nuestra interfaz Motor que define a los métodos de modo general, y que son implementados por los distintos tipos de motores.

3.5.2 Solución del Problema

Crear una clase adaptadora, que sirva como puente entre la clase Adapter (clase a adaptar) y el Target interface o clase abstracta que define los métodos que queremos adecuar.

3.4.3 Diagrama de Clase

Imagen 27. Diagrama de clases por Eduardo Ordoñez



Nuestro diagrama de clase nos muestra que tenemos una clase padres abstracta la cual es motor de esta clase salen 3 clases hijas las cuales heredarán los métodos abstractos creados en la clase padre esta clase hija serán MotorComun, MotorElectricoAdapter y MotorEconomico.

De nuestra clase MotorElectrico, que se creó ya que se necesito ingresar el nuevo motor obtenido en la empresa crearemos esta clase la cual tendrá los métodos necesarios para encender el nuevo motor obtenido, para obtener estos datos en nuestra clase MotorElectricoAdapter declararemos como atributo a nuestra clase MotorElectrico y así poner el funcionamiento en nuevo motor ingresado.

3.5.4 Código del Ejercicio

```
package backend;

public abstract class Motor {
   abstract public void encender();
   abstract public void acelerar();
   abstract public void apagar();
}
```

```
package backend;

public class MotorComun extends Motor {

    public MotorComun(){
        super();
        System.out.println("Creando motor comun...");
    }

    @Override
    public void encender() {
        System.out.println("Encendiendo motor comun.");
    }

    @Override
    public void acelerar() {
        System.out.println("Acelerando motor comun.");
    }

    @Override
    public void apagar() {
        System.out.println("Apagando motor comun.");
    }
}
```

```
package backend;

public class MotorElectrico {

    private boolean conectado = false;

    public MotorElectrico() {
        System.out.println("Creando motor electrico...");
        this.conectado = false;
    }
```

```
    public void conectar() {
        System.out.println("Conectando motor electrico.");
        this.conectado = true;
    }

    public void activar() {
        if (!this.conectado) {
            System.out.println("No se puede activar porque no esta conectado el motor electrico.");
        } else {
            System.out.println("Esta conectado, activando motor electrico.");
        }
    }

    public void moverMasRapido() {
        if (!this.conectado) {
            System.out.println("No se puede mover rapido el motor electrico porque no esta
conectado.");
        } else {
            System.out.println("Moviendo mas rapido, aumentando voltaje del motor electrico.");
        }
    }

    public void detener() {
        if (!this.conectado) {
            System.out.println("No se puede detener motor electrico porque no esta conectado.");
        } else {
            System.out.println("Deteniendo motor electrico.");
        }
    }

    public void desconectar() {
        System.out.println("Desconectando motor electrico.");
        this.conectado = false;
    }
}
```

```
package backend;

public class MotorElectricoAdapter extends Motor{
    private MotorElectrico motorElectrico;

    public MotorElectricoAdapter(){
        super();
        System.out.println("Creando motor electrico adapter...");
        this.motorElectrico = new MotorElectrico();
```

```
    }
    @Override
    public void encender() {
        System.out.println("Encendiendo motor electrico adapter.");
        this.motorElectrico.conectar();
        this.motorElectrico.activar();
    }

    @Override
    public void acelerar() {
        System.out.println("Acelerando motor electrico adapter.");
        this.motorElectrico.moverMasRapido();
    }

    @Override
    public void apagar() {
        System.out.println("Apagando motor electrico adapter.");
        this.motorElectrico.detener();
        this.motorElectrico.desconectar();
    }
}
```

```
package backend;

public class MotorEconomico extends Motor {

    public MotorEconomico(){
        super();
        System.out.println("Creando motor economico...");
    }

    @Override
    public void encender() {
        System.out.println("Encendiendo motor economico.");
    }

    @Override
    public void acelerar() {
        System.out.println("Acelerando el motor economico.");
    }

    @Override
    public void apagar() {
        System.out.println("Apagando motor economico.");
    }
```

```
}
```

```
package frontend;

import backend.Motor;
import backend.MotorComun;
import backend.MotorEconomico;
import backend.MotorElectricoAdapter;
import java.util.Scanner;

public class Aplicacion {

   private static Scanner S = new Scanner(System.in);
   private static Motor motor;

   public static void main(String[] args) {
      System.out.println("");
      int opcion;
      do{
         opcion = preguntarOpcion();
         switch(opcion) {
            case 1:
               motor = new MotorComun();
               usarMotor();
               break;
            case 2:
               motor = new MotorEconomico();
               usarMotor();
               break;
            case 3:
               motor = new MotorElectricoAdapter();
               usarMotor();
               break;
            case 4:
               System.out.println("¡Cerrando programa!");
               break;
            default:
               System.out.println("La opción ingresada NO es valida.");
         }
         System.out.print("\n\n");
      }while(opcion!=4);
   }

   private static int preguntarOpcion() {
      System.out.print(
            "MENU DE OPCIONES\n"
```

```
            +"---- -- --------\n"
            +"1. Encender motor comun.\n"
            +"2. Encender motor economico.\n"
            +"3. Encender motor electrico.\n"
            +"4. Salir.\n"
            +"Seleccione opcion: "
        );
        return Integer.parseInt( S.nextLine() );
    }

    private static void usarMotor() {
        motor.encender();
        motor.acelerar();
        motor.apagar();
    }
}
```

3.5.5 Anexos del código

Imagen 28. Clase Motor por Eduardo Ordoñez

```
package backend;

public abstract class Motor {
    abstract public void encender();
    abstract public void acelerar();
    abstract public void apagar();
}
```

Declaramos los métodos a usar en nuestra clase padre abstracta para heredarlos a nuestra clase de hijas.

Imagen 29. Clase MotorComun por Eduardo Ordoñez

```java
package backend;

public class MotorComun extends Motor {

    public MotorComun(){
        super();
        System.out.println(x: "Creando motor comun...");
    }

    @Override
    public void encender() {
        System.out.println(x: "Encendiendo motor comun.");
    }

    @Override
    public void acelerar() {
        System.out.println(x: "Acelerando motor comun.");
    }

    @Override
    public void apagar() {
        System.out.println(x: "Apagando motor comun.");
    }
}
```

Heredamos los métodos abstractos de nuestra clase padre y hacemos un constructor vacío que no se declararon variables ni en la clase padre ni en la clase hija.

Imagen 30. Clase MotorEconomico por Eduardo Ordoñez

```
package backend;

public class MotorEconomico extends Motor {

    public MotorEconomico(){
        super();
        System.out.println(x: "Creando motor economico...");
    }

    @Override
    public void encender() {
        System.out.println(x: "Encendiendo motor economico.");
    }

    @Override
    public void acelerar() {
        System.out.println(x: "Acelerando el motor economico.");
    }

    @Override
    public void apagar() {
        System.out.println(x: "Apagando motor economico.");
    }
}
```

Lo mismo que en nuestra clase anterior, heredamos los métodos abstractos de nuestra clase padre y hacemos un constructor vacío que no se declararon variables ni en la clase padre ni en la clase hija.

Imagen 31. Clase MotorElectrico por Eduardo Ordoñez

```
package backend;
public class MotorElectrico {
    private boolean conectado = false;
    public MotorElectrico() {
        System.out.println(x: "Creando motor electrico...");
        this.conectado = false;
    }
    public void conectar() {
        System.out.println(x: "Conectando motor electrico.");
        this.conectado = true;
    }
    public void activar() {
        if (!this.conectado) {
            System.out.println(x: "No se puede activar porque no esta conectado el motor electrico.");
        } else {
            System.out.println(x: "Esta conectado, activando motor electrico.");
        }
    }
    public void moverMasRapido() {
        if (!this.conectado) {
            System.out.println(x: "No se puede mover rapido el motor electrico porque no esta conectado.");
        } else {
            System.out.println(x: "Moviendo mas rapido, aumentando voltaje del motor electrico.");
        }
    }
    public void detener() {
        if (!this.conectado) {
            System.out.println(x: "No se puede detener motor electrico porque no esta conectado.");
        } else {
            System.out.println(x: "Deteniendo motor electrico.");
        }
    }
    public void desconectar() {
        System.out.println(x: "Desconectando motor electrico.");
        this.conectado = false;
    }
}
```

En esta clase declaramos un variables boolean para crear nuestro procesos a seguir para poder encender el motor solicitado, junto a los nuevos métodos.

Imagen 32. Clase MotorElectricoAdapter por Eduardo Ordoñez

```
package backend;

public class MotorElectricoAdapter extends Motor{
    private MotorElectrico motorElectrico;

    public MotorElectricoAdapter(){
        super();
        System.out.println(x: "Creando motor electrico adapter...");
        this.motorElectrico = new MotorElectrico();
    }
    @Override
    public void encender() {
        System.out.println(x: "Encendiendo motor electrico adapter.");
        this.motorElectrico.conectar();
        this.motorElectrico.activar();
    }

    @Override
    public void acelerar() {
        System.out.println(x: "Acelerando motor electrico adapter.");
        this.motorElectrico.moverMasRapido();
    }

    @Override
    public void apagar() {
        System.out.println(x: "Apagando motor electrico adapter.");
        this.motorElectrico.detener();
        this.motorElectrico.desconectar();
    }
}
```

Aquí como podemos ver declaramos como atributos a nuestra clase MotorElectrico para poder así relacionar nuestra clase MotorElectricoAdapter con MotorElectrico y por último usamos una función switch para cada uno de los motores que posee la empresa automotriz

Imagen 33. Clase Aplicacion (principal) por Eduardo Ordoñez

```java
package frontend;

import backend.Motor;
import backend.MotorComun;
import backend.MotorEconomico;
import backend.MotorElectricoAdapter;
import java.util.Scanner;

public class Aplicacion {

    private static Scanner S = new Scanner(source:System.in);
    private static Motor motor;

    public static void main(String[] args) {
        System.out.println(x: "");
        int opcion;
        do{
            opcion = preguntarOpcion();
            switch(opcion) {
                case 1:
                    motor = new MotorComun();
                    usarMotor();
                    break;
                case 2:
                    motor = new MotorEconomico();
                    usarMotor();
                    break;
                case 3:
                    motor = new MotorElectricoAdapter();
                    usarMotor();
                    break;
```

```
                case 4:
                    System.out.println(x: "¡Cerrando programa!");
                    break;
                default:
                    System.out.println(x: "La opción ingresada NO es valida.");
            }
            System.out.print(s: "\n\n");
        }while(opcion!=4);
    }

    private static int preguntarOpcion() {
        System.out.print(
                "MENU DE OPCIONES\n"
               +"---- -- --------\n"
               +"1. Encender motor comun.\n"
               +"2. Encender motor economico.\n"
               +"3. Encender motor electrico.\n"
               +"4. Salir.\n"
               +"Seleccione opcion: "
        );
        return Integer.parseInt( s: S.nextLine() );
    }

    private static void usarMotor() {
        motor.encender();
        motor.acelerar();
        motor.apagar();
    }
}
```

4. Conclusiones

- Mejora la calidad del código al proporcionar soluciones probadas para problemas comunes.
- Fomenta la reutilización de soluciones exitosas, acelerando el desarrollo y reduciendo errores.
- Facilita la comunicación entre desarrolladores al establecer un lenguaje común.

5. Recomendaciones

- Aplica los patrones de manera adecuada según el contexto y requisitos del proyecto.

- Familiarízate con patrones específicos de dominio para abordar problemas particulares.

6. Bibliografía

Canelo, M. M. (24 de Junio de 2020). *profile*. profile: https://profile.es/blog/patrones-de-diseno-de-software/#%C2%BFQue_son_los_patrones_de_diseno_design_patterns

Sánchez, M. Á. (22 de Noviembre de 2017). *medium*. Patrones de Diseño de Software: https://medium.com/all-you-need-is-clean-code/patrones-de-dise%C3%B1o-b7a99b8525e

Soto, N. (2021, Julio 2). *¿QUÉ SON LOS PATRONES DE DISEÑO?* Craft Code. https://craft-code.com/que-son-los-patrones-de-diseno/#:~:text=Los%20patrones%20estructurales%20buscan%20facilitar,interfaces%20incompatibles%20colaboren%20entre%20s%C3%AD