Universidad de las Fuerzas Armadas ESPE

Departamento: Ciencias de la Computación

Carrera : Ingeniería de Software

Taller académico Nª: 4

---

1. Información General

- **Asignatura:** Análisis y Diseño
- **Apellidos y nombres de los estudiantes:** Joan Cobeña, Juan Pasquel, Ruben Benavides, Edison Verdesoto
- **NRC:** 22426
- **Fecha de realización:** 14/06/2025

---

2. Objetivo del Taller y Desarrollo

Objetivo del Taller:

Comprender el alcance y utilidad de los patrones de diseño Adapter y Command a través de la descripción de su comportamiento y un ejemplo práctico de aplicación con la finalidad de aplicarlos eficazmente en futuros proyectos de desarrollo de software.

Desarrollo:

Patrones de diseño de software

Los patrones de diseño constituyen “soluciones típicas a problemas comunes de diseño” y proporcionan un lenguaje compartido para los equipos de desarrollo (Gamma et al., 1994). A continuación se describen los patrones Adapter y Command, sus necesidades o prerrequisitos y un ejemplo integrado de aplicación.

Adapter

El patrón Adapter actúa como un envoltorio (wrapper), que permite que dos clases con interfaces incompatibles colaboren sin modificar su código fuente original (Refactoring.Guru, n.d.). En la Tabla 1 se pueden observar los prerrequisitos para utilizar este patrón.

Requisito	**Justificación**
Interoperar con código de terceros	Cuando una parte de la aplicación usa una interfaz no modificable y una nueva clase produce datos en un formato distinto.
Mantener *Open/Close Principle*	Se evita tocar la implementación existente; se agrega un adaptador que traduzca las llamadas.
Facilitar migraciones graduales	Permite que componentes antiguos convivan con nuevos en la transición.

Tabla 1. Prerrequisitos Adapter

Participantes esenciales

- **Client:** Código que espera una interfaz objetivo.
- **Target:** Interfaz que el Client entiende.
- **Adaptee:** Clase existente con interfaz incompatible.
- **Adapter:** Traductor de llamadas del Target al Adaptee.

En un flujo normal de implementación de Adapter, este encapsula una instancia de Adaptee y, cuando lo invocan, transforma los parámetros antes de delegar la operación.

Command

El patrón Command "convierte una solicitud en un objeto independiente" que guarda toda la información necesaria para ejecutarla—o deshacerla—cuando corresponda (Refactoring.Guru, n.d.). En la tabla 2 se pueden observar los prerrequisitos para utilizar este patrón.

Requisito	Justificación
Desacoplar emisores y receptores de una acción	El emisor desconoce qué ni cómo el objeto realiza la acción.
Colas, historial, deshacer o rehacer	Al almacenar comandos se puede re-ejecutar, diferir o revertir las acciones.
Parametrizar objetos con operaciones	Los comandos pueden ser argumentos, configurarse dinámicamente o enviarse por red.

Tabla 2. Prerrequisitos Adapter

Participantes esenciales

- **Command (interfaz):** expone el execute() y ocasionalmente undo().
- **ConcreteCommand:** encapsula los datos de la petición y delega al Receiver.
- **Receiver:** ejecuta la lógica del negocio.
- **Invoker:** dispara la ejecución del comando.
- **Client:** configura los comandos y los vincula con los invokers.

Ejemplo de aplicación

Utilizando el código de estudiantes del anterior taller, se hacen unos pequeños arreglos para que el código implemente los patrones Adapter y Command.

Adapter

- **EstudianteArchivoCommand:** Este adaptador se encarga de gestionar la persistencia de datos en un binario. Implementa la interfaz IEstudianteDataSource para permitir operaciones CRUD desde un archivo.

```
package ec.edu.espe.adapter;

import ec.edu.espe.modelo.Estudiante;

import java.io.*;
```

```java
import java.util.ArrayList;

import java.util.List;

/**

 * Adaptador para persistencia en archivos

 */

public class EstudianteArchivoAdapter implements
IEstudianteDataSource {

    private static final String ARCHIVO = "estudiantes.dat";

    private List<Estudiante> estudiantes;

    public EstudianteArchivoAdapter() {

        cargarDatos();

    }

    /* Métodos CRUD */

    @Override public void guardar(Estudiante estudiante) {

        estudiantes.add(estudiante);

        guardarDatos();

    }

    @Override public void actualizar(Estudiante estudiante) {

        for (int i = 0; i < estudiantes.size(); i++) {

            if (estudiantes.get(i).getId() == estudiante.getId()) {

                estudiantes.set(i, estudiante);

                guardarDatos();
```

```
                break;

            }

        }

    }

    @Override public void eliminar(int id) {

        estudiantes.removeIf(e -> e.getId() == id);

        guardarDatos();

    }

    @Override public List<Estudiante> obtenerTodos() {

        return new ArrayList<>(estudiantes);

    }

    @Override public Estudiante buscarPorId(int id) {

        return estudiantes.stream()

                .filter(e -> e.getId() == id)

                .findFirst()

                .orElse(null);

    }

    /* Utilidades de serialización */

    @SuppressWarnings("unchecked")

    private void cargarDatos() {

        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(ARCHIVO))) {
```

```
            estudiantes = (List<Estudiante>) ois.readObject();

        } catch (Exception e) {

            estudiantes = new ArrayList<>();

        }

    }

    private void guardarDatos() {

        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(ARCHIVO))) {

            oos.writeObject(estudiantes);

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

- **EstudianteRepositorioAdapter:** Permite la interacción del código Cliente con el repositorio en memoria de manera que respete la interfaz común. Esto asegura que la fuente de datos se pueda cambiar fácilmente sin modificar el código del cliente.

```
package ec.edu.espe.adapter;

import ec.edu.espe.modelo.Estudiante;

import ec.edu.espe.repositorio.EstudianteRepositorio;

import java.util.List;

/**
```

```
 * Adaptador para el repositorio en memoria existente

 */

public class EstudianteRepositorioAdapter implements
IEstudianteDataSource {

    private final EstudianteRepositorio repositorio;

    public EstudianteRepositorioAdapter(EstudianteRepositorio
repositorio) {

        this.repositorio = repositorio;

    }

    @Override public void guardar(Estudiante estudiante)    {
repositorio.agregar(estudiante); }

    @Override public void actualizar(Estudiante estudiante){
repositorio.actualizar(estudiante); }

    @Override public void eliminar(int id)                  {
repositorio.eliminar(id); }

    @Override public List<Estudiante> obtenerTodos()        { return
repositorio.obtenerTodos(); }

    @Override public Estudiante buscarPorId(int id)         { return
repositorio.buscarPorId(id); }

}
```

- **IEstudianteDataSource:** Define las operaciones básicas que debe implementar cualquier fuente de datos que se utilice para gestionar estudiantes, sea en memoria, archivo o base de datos.

```
package ec.edu.espe.adapter;

import ec.edu.espe.modelo.Estudiante;

import java.util.List;
```

```java
/**

 * Interfaz que define las operaciones de una fuente de datos de estudiantes

 */

public interface IEstudianteDataSource {

    void guardar(Estudiante estudiante);

    void actualizar(Estudiante estudiante);

    void eliminar(int id);

    List<Estudiante> obtenerTodos();

    Estudiante buscarPorId(int id);

}
```

Command

- **ActualizarEstudianteCommand:** Encapsula la lógica para actualizar un estudiante, guarda el estado anterior del estudiante para permitir un undo en caso de que se desee revertir la operación.

```java
package ec.edu.espe.command;

import ec.edu.espe.modelo.Estudiante;

import ec.edu.espe.repositorio.EstudianteRepositorio;

/**

 * Comando para actualizar un estudiante

 */
```

```java
public class ActualizarEstudianteCommand implements Command {



    private final EstudianteRepositorio repositorio;

    private final Estudiante estudianteNuevo;

    private final Estudiante estudianteAnterior;

    public ActualizarEstudianteCommand(EstudianteRepositorio
repositorio, Estudiante estudianteNuevo) {

        this.repositorio        = repositorio;

        this.estudianteNuevo    = estudianteNuevo;

        this.estudianteAnterior =
repositorio.buscarPorId(estudianteNuevo.getId());

    }

    @Override

    public void execute() {

        repositorio.actualizar(estudianteNuevo);

    }

    @Override

    public void undo() {

        if (estudianteAnterior != null) {

            repositorio.actualizar(estudianteAnterior);

        }

    }

}
```

- **AgregarEstudianteCommand:** Maneja la lógica para agregar un estudiante. Permite revertir la acción mediante undo, eliminando al estudiante en el caso de ser necesario.

```
package ec.edu.espe.command;

import ec.edu.espe.modelo.Estudiante;

import ec.edu.espe.repositorio.EstudianteRepositorio;

/**

 * Comando para agregar un estudiante

 */

public class AgregarEstudianteCommand implements Command {

    private final EstudianteRepositorio repositorio;

    private final Estudiante estudiante;

    public AgregarEstudianteCommand(EstudianteRepositorio
repositorio, Estudiante estudiante) {

        this.repositorio = repositorio;

        this.estudiante  = estudiante;

    }

    @Override

    public void execute() {

        repositorio.agregar(estudiante);

    }

    @Override
```

```
    public void undo() {
        repositorio.eliminar(estudiante.getId());
    }
}
```

- **Command:** Convierte una solicitud en un objeto, encapsula la información necesaria para ejecutarla o deshacerla. Permite mantener un historial de las operaciones y desacoplar el emisor de la acción del receptor de la misma.

```
package ec.edu.espe.command;

/**
 * Interfaz Command que define el método execute para ejecutar comandos
 */
public interface Command {
    void execute();
    void undo();    // Para deshacer la operación
}
```

- **CommandInvoker:** Es el responsable de ejecutar comandos y mantener el historial que permite realizar el undo de la última operación realizada.

```
package ec.edu.espe.command;

import java.util.Stack;

/**
 * Invocador de comandos; mantiene un historial para deshacer
 */
```

```java
public class CommandInvoker {

    private final Stack<Command> historial = new Stack<>();

    public void ejecutarComando(Command comando) {

        comando.execute();

        historial.push(comando);

    }

    public void deshacer() {

        if (!historial.isEmpty()) {

            historial.pop().undo();

        }

    }

    public boolean puedeDeshacer() {

        return !historial.isEmpty();

    }

}
```

- **EliminarEstudianteCommand:** Permite eliminar un estudiante, y tambíen guarda el estado del estudiante antes de la eliminación para dar paso a la restauración si se deshace la operación.

```java
package ec.edu.espe.command;

import ec.edu.espe.modelo.Estudiante;

import ec.edu.espe.repositorio.EstudianteRepositorio;

/**
```

```java
 * Comando para eliminar un estudiante

 */

public class EliminarEstudianteCommand implements Command {

    private final EstudianteRepositorio repositorio;

    private final int id;

    private Estudiante estudianteEliminado;



    public EliminarEstudianteCommand(EstudianteRepositorio
repositorio, int id) {

        this.repositorio = repositorio;

        this.id= id;

    }

    @Override

    public void execute() {

        estudianteEliminado = repositorio.buscarPorId(id); //
respaldo

        repositorio.eliminar(id);

    }

    @Override

    public void undo() {

        if (estudianteEliminado != null) {

            repositorio.agregar(estudianteEliminado);

        }
```

```
    }

}
```

Conclusiones:

1. La combinación de los patrones demostró que es posible ampliar y modificar el comportamiento de la aplicación sin alterar la lógica de presentación ni el dominio, confirmando una alta cohesión y un bajo acoplamiento entre capas.
2. Durante las pruebas, la solución resultó más mantenible, ya que añadir un nuevo comando o un adaptador requiere crear únicamente una clase e inyectarla, sin modificar el código existente; esto respeta el principio *Open/Closed* y facilita evoluciones futuras.

Recomendaciones:

1. Crear y ejecutar pruebas unitarias para cada ConcreteCommand y a cada adapter para garantizar que las nuevas extensiones se integran sin romper funcionalidades previas y mantendrá la confiabilidad del mecanismo undo/redo y del cambio en la persistencia.
2. Extender el patrón Command para soportar funciones adicionales, como redo y macros de acciones encadenadas, diseñar una interfaz que permita elegir fuentes de datos mediante la incorporación de Adapters sin modificar la capa de servicio.