Algoritmos y Estructuras de Datos II

Trabajo Práctico N° 1	Unidad 1-003
Modalidad: Semi -Presencial	**Estratégica Didáctica:** Trabajo individual
Metodología de Desarrollo: Det. docente	**Metodología de Corrección:** Via Classroom.
Carácter de Trabajo: Obligatorio – Con Nota	**Fecha Entrega:** A confirmar por el Docente.

MARCO TEÓRICO

PATRONES DE COMPORTAMIENTO

Responder el siguiente cuestionario en función de la bibliografía Obligatoria.

1. Describir los Tipos de Patrones de Comportamiento

Los patrones de comportamiento son una categoría de patrones de diseño que se centran en cómo los objetos interactúan y colaboran entre sí. Se utilizan para administrar y organizar el flujo de control y la comunicación entre los componentes de un programa.

Patrón Command: Encapsula una solicitud como un objeto.

Patrón Observer: Establece una relación de uno a muchos. Cuando un objeto cambia su estado, notifica automáticamente a todos sus dependientes.

Patrón State: Permite que un objeto cambie su comportamiento cuando su estado interno cambia.

Patrón Strategy: Define una familia de algoritmos intercambiables y encapsula cada uno.

Patrón Iterator: Proporciona una forma uniforme de acceder a los elementos de una colección sin exponer su estructura interna.

Patrón Mediator: Centraliza la comunicación entre múltiples objetos, forzándolos a colaborar solo a través del mediador.

Patrón Template Method: Define la estructura de un algoritmo en una superclase, permitiendo que las subclases modifiquen algunos pasos sin cambiar la estructura general.

Patrón Visitor: Se utiliza para separar un algoritmo de la estructura de un objeto sobre el que opera.

Patrón Chain of Responsibility: Crea una cadena de objetos receptores para manejar una solicitud, permitiendo que cada objeto decida si procesa la solicitud o la pasa al siguiente.

2. Que tipos de Problema Resuelve

Cada patrón de comportamiento resuelve un problema específico relacionado con la interacción y la asignación de responsabilidades:

Command: Resuelve el desacoplamiento entre el emisor de una solicitud y su receptor. Facilita la creación de secuencias de comandos, operaciones reversibles (deshacer) y la gestión de comandos.

Observer: Resuelve la necesidad de mantener la coherencia entre objetos dependientes sin acoplarlos fuertemente.

State: Resuelve la gestión de objetos que pueden tener múltiples estados y transiciones. Facilita el manejo de un objeto que se comporta de manera diferente según su estado interno.

Strategy: Resuelve la necesidad de seleccionar un algoritmo en tiempo de ejecución. Permite modificar el comportamiento de un objeto (usando diferentes algoritmos) sin cambiar su estructura.

Iterator: Resuelve el problema de la navegación y el recorrido de colecciones de datos, ocultando la complejidad de la estructura interna (listas, árboles, etc.).

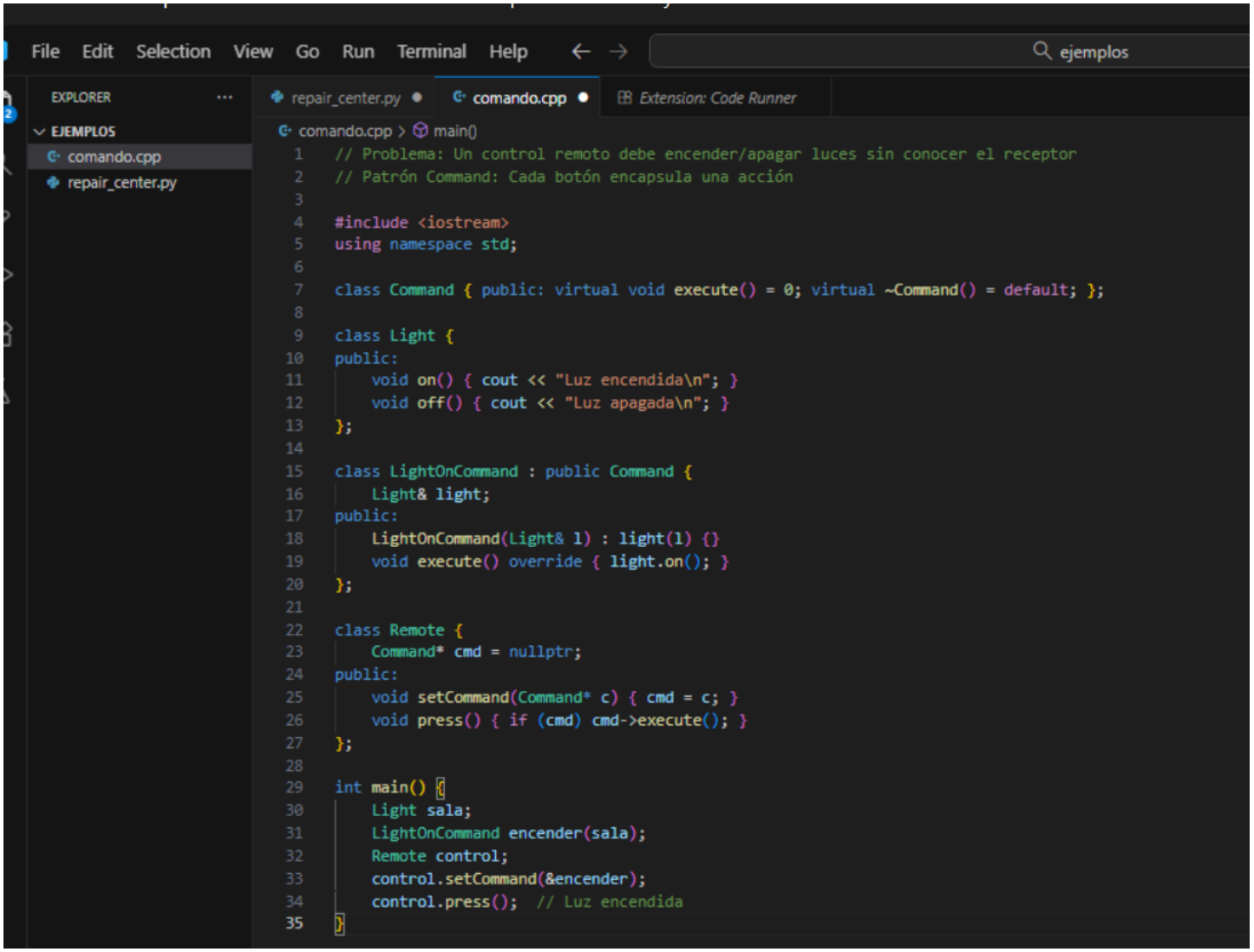
Mediator: Resuelve el problema de la comunicación caótica entre múltiples objetos ("dependencias caóticas"). Actúa como un intermediario para que los objetos no necesiten comunicarse directamente entre sí.

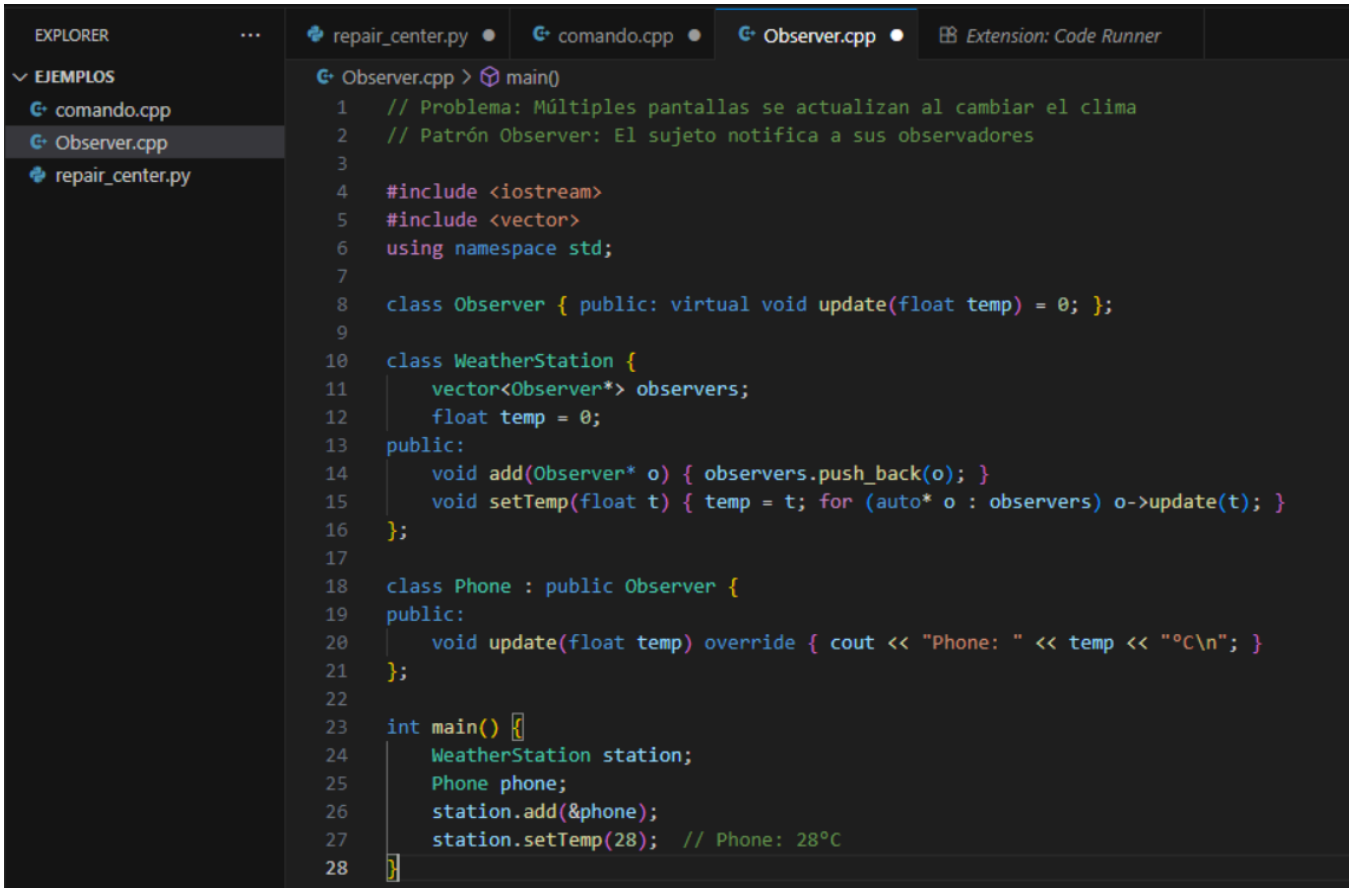
Template Method: Resuelve el problema de tener algoritmos con una estructura similar pero con pasos específicos que varían. Permite que las subclases modifiquen esos pasos específicos sin cambiar la estructura general del algoritmo.

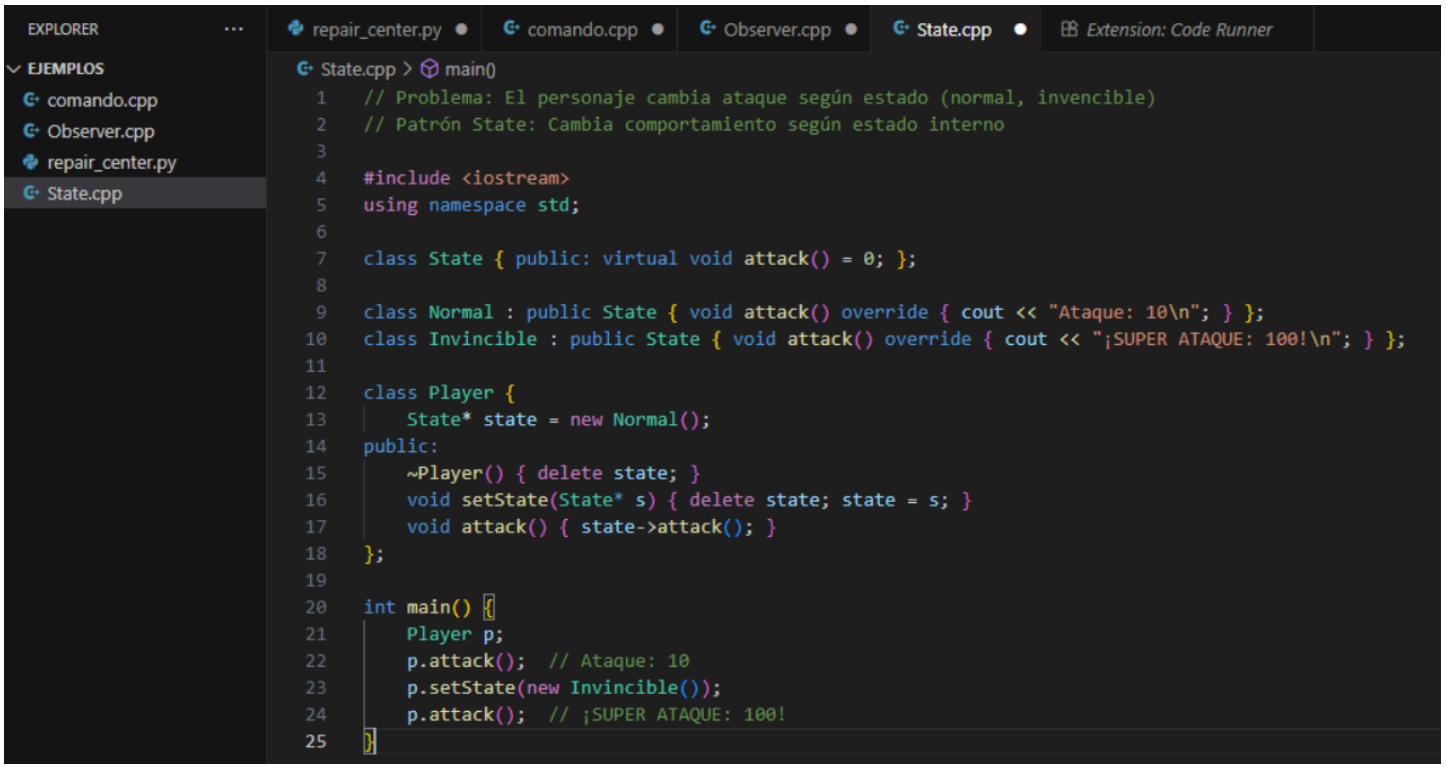
Visitor: Resuelve la necesidad de agregar nuevas operaciones a objetos sin tener que modificar sus clases. Separa el algoritmo de la estructura del objeto.

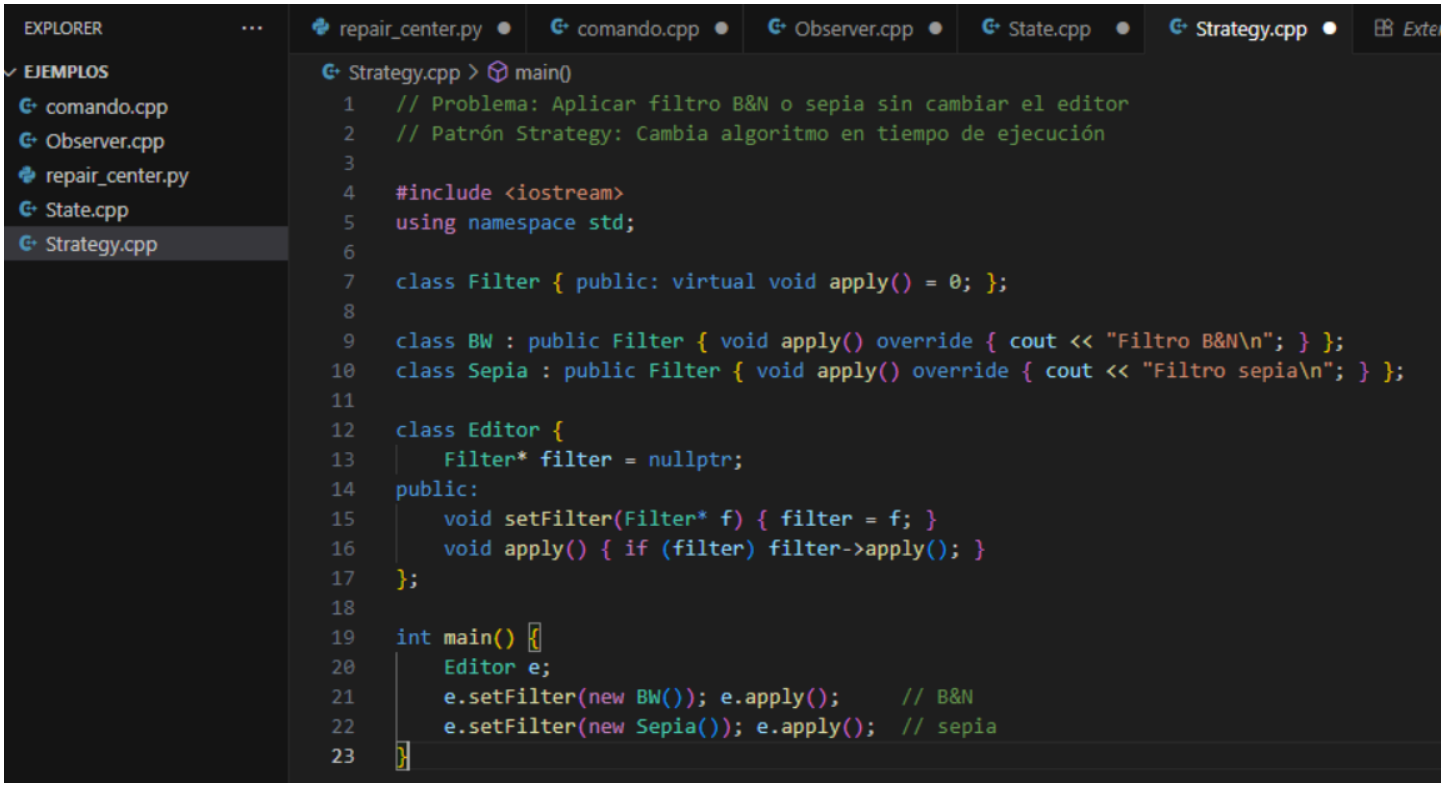
Chain of Responsibility: Resuelve el acoplamiento fuerte entre el remitente de una solicitud y sus receptores. Permite que una solicitud pase a través de una cadena de posibles manejadores hasta que uno de ellos la procese.

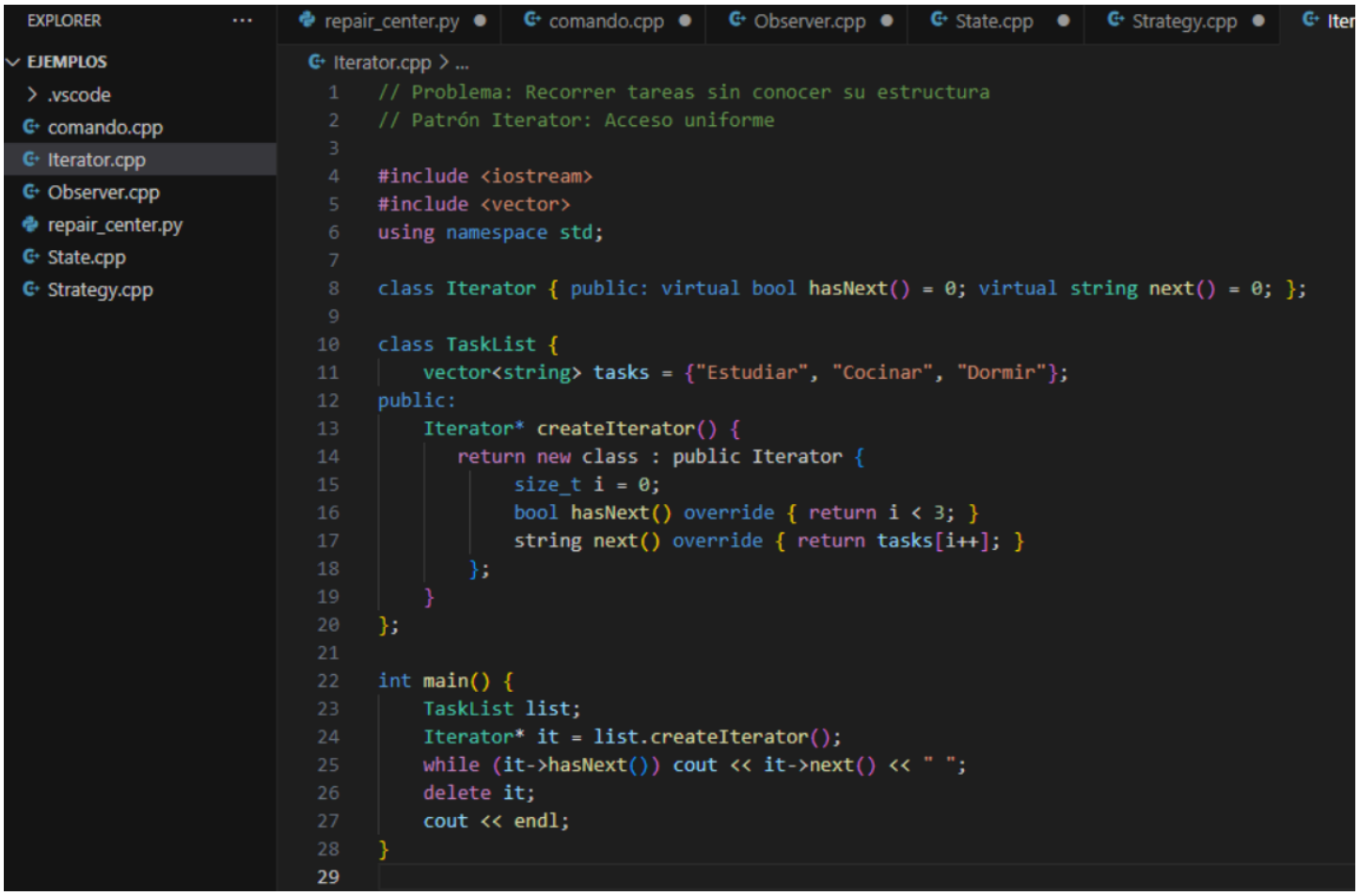
3. Dar ejemplos de Problema – Patrones de Comportamiento.

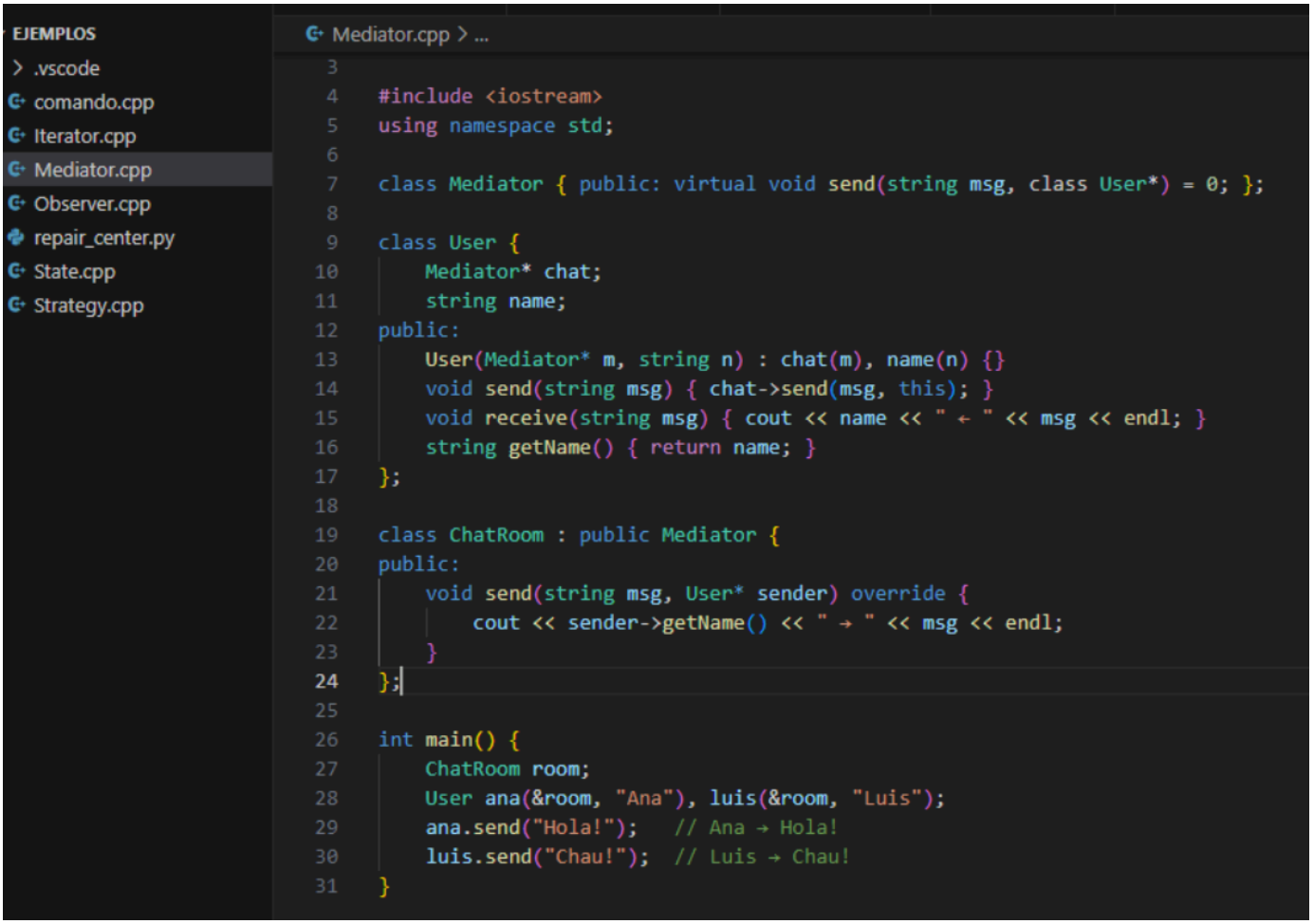

File Edit Selection View Go Run Terminal Help
ejemplos
EXPLORER
EJEMPLOS
comando.cpp
repair_center.py
repair_center.py
comando.cpp
Extension: Code Runner
comando.cpp > main()
// Problema: Un control remoto debe encender/apagar luces sin conocer el receptor
// Patrón Command: Cada botón encapsula una acción
#include <iostream>
using namespace std;
class Command { public: virtual void execute() = 0; virtual ~Command() = default; };
class Light {
public:
void on() { cout << "Luz encendida\n"; }
void off() { cout << "Luz apagada\n"; }
};
class LightOnCommand : public Command {
Light& light;
public:
LightOnCommand(Light& l) : light(l) {}
void execute() override { light.on(); }
};
class Remote {
Command* cmd = nullptr;
public:
void setCommand(Command* c) { cmd = c; }
void press() { if (cmd) cmd->execute(); }
};
int main() {
Light sala;
LightOnCommand encender(sala);
Remote control;
control.setCommand(&encender);
control.press(); // Luz encendida
}

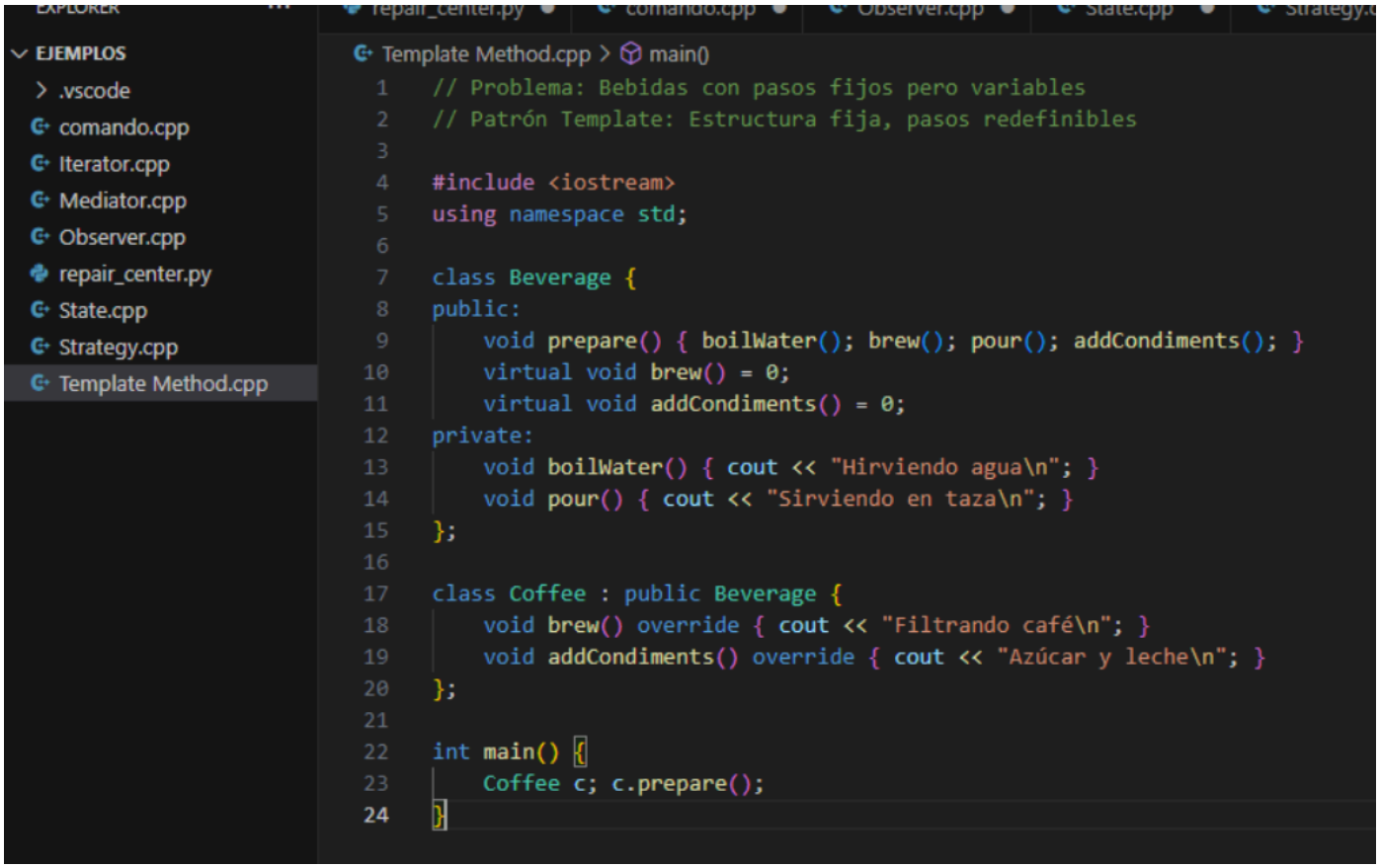

EXPLORER
EJEMPLOS
comando.cpp
Observer.cpp
repair_center.py
repair_center.py
comando.cpp
Observer.cpp
Extension: Code Runner
Observer.cpp > main()
// Problema: Múltiples pantallas se actualizan al cambiar el clima
// Patrón Observer: El sujeto notifica a sus observadores
#include <iostream>
#include <vector>
using namespace std;
class Observer { public: virtual void update(float temp) = 0; };
class WeatherStation {
vector<Observer*> observers;
float temp = 0;
public:
void add(Observer* o) { observers.push_back(o); }
void setTemp(float t) { temp = t; for (auto* o : observers) o->update(t); }
};
class Phone : public Observer {
public:
void update(float temp) override { cout << "Phone: " << temp << "°C\n"; }
};
int main() {
WeatherStation station;
Phone phone;
station.add(&phone);
station.setTemp(28); // Phone: 28°C
}

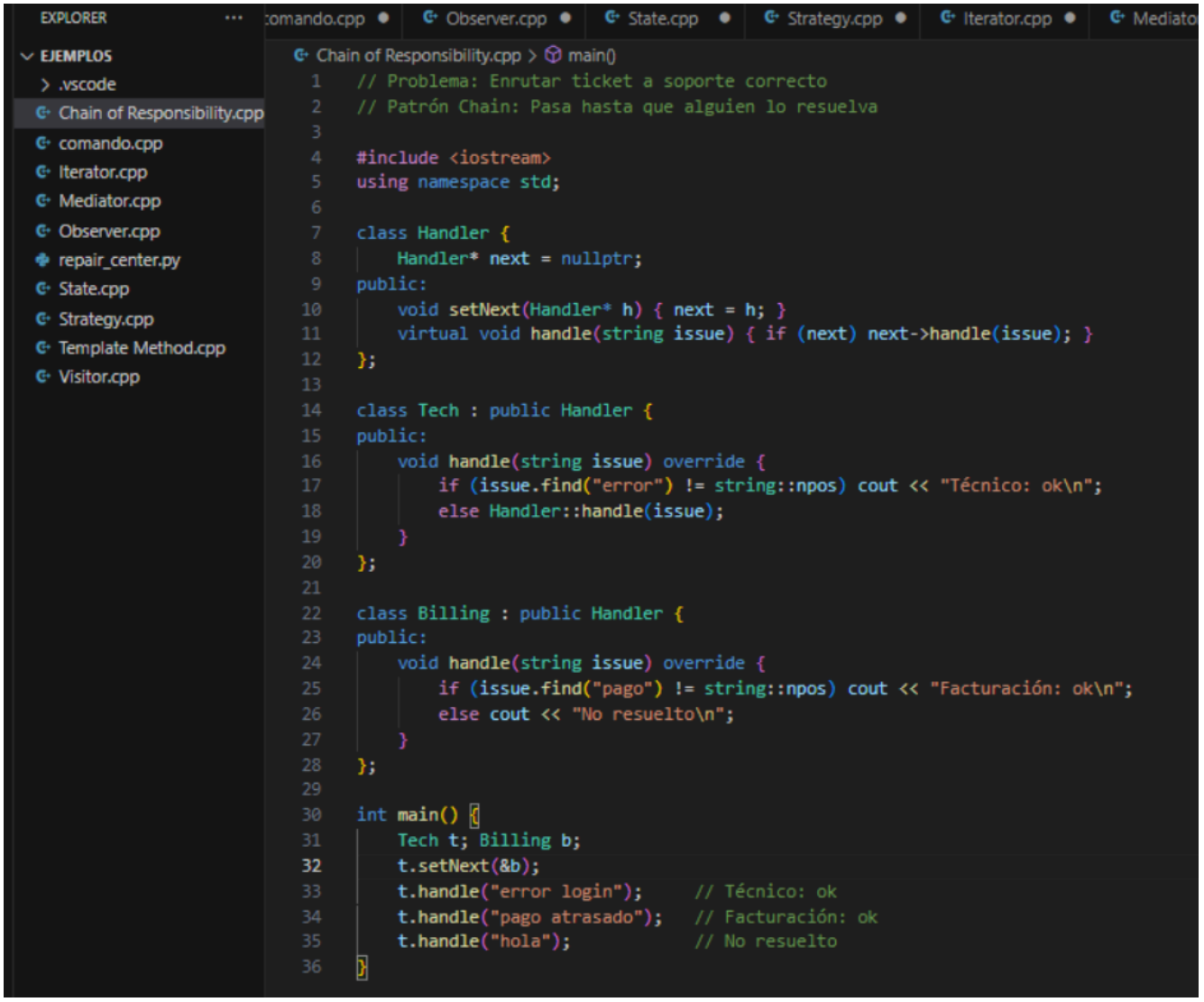

EXPLORER
EJEMPLOS
comando.cpp
Observer.cpp
repair_center.py
State.cpp
repair_center.py
comando.cpp
Observer.cpp
State.cpp
Extension: Code Runner
State.cpp > main()
// Problema: El personaje cambia ataque según estado (normal, invencible)
// Patrón State: Cambia comportamiento según estado interno
#include <iostream>
using namespace std;
class State { public: virtual void attack() = 0; };
class Normal : public State { void attack() override { cout << "Ataque: 10\n"; } };
class Invincible : public State { void attack() override { cout << "¡SUPER ATAQUE: 100!\n"; } };
class Player {
State* state = new Normal();
public:
~Player() { delete state; }
void setState(State* s) { delete state; state = s; }
void attack() { state->attack(); }
};
int main() {
Player p;
p.attack(); // Ataque: 10
p.setState(new Invincible());
p.attack(); // ¡SUPER ATAQUE: 100!
}

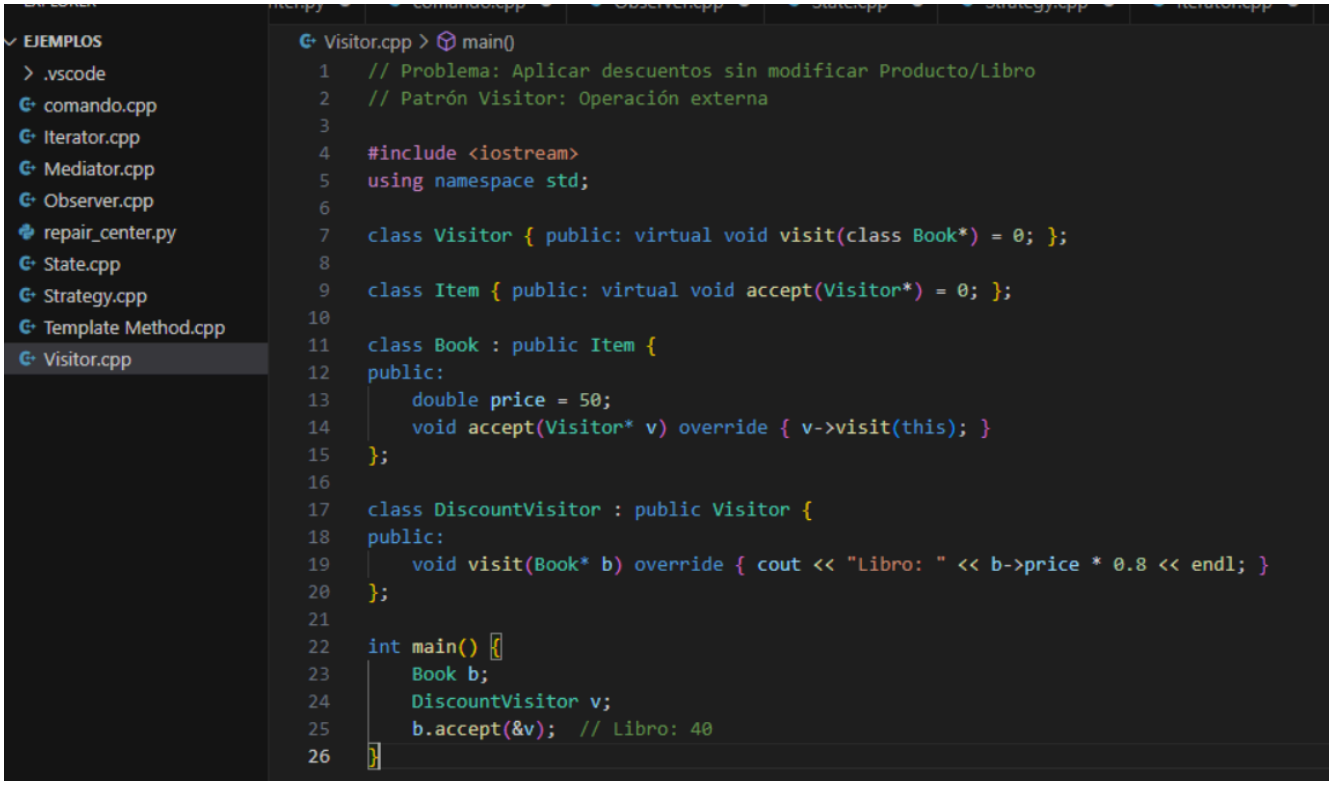

EXPLORER
EJEMPLOS
comando.cpp
Observer.cpp
repair_center.py
State.cpp
Strategy.cpp
repair_center.py
comando.cpp
Observer.cpp
State.cpp
Strategy.cpp
Strategy.cpp > main()
// Problema: Aplicar filtro B&N o sepia sin cambiar el editor
// Patrón Strategy: Cambia algoritmo en tiempo de ejecución
#include <iostream>
using namespace std;
class Filter { public: virtual void apply() = 0; };
class BW : public Filter { void apply() override { cout << "Filtro B&N\n"; } };
class Sepia : public Filter { void apply() override { cout << "Filtro sepia\n"; } };
class Editor {
Filter* filter = nullptr;
public:
void setFilter(Filter* f) { filter = f; }
void apply() { if (filter) filter->apply(); }
};
int main() {
Editor e;
e.setFilter(new BW()); e.apply(); // B&N
e.setFilter(new Sepia()); e.apply(); // sepia
}


EXPLORER
EJEMPLOS
.vscode
comando.cpp
Iterator.cpp
Observer.cpp
repair_center.py
State.cpp
Strategy.cpp
repair_center.py
comando.cpp
Observer.cpp
State.cpp
Strategy.cpp
Iterator.cpp > ...
// Problema: Recorrer tareas sin conocer su estructura
// Patrón Iterator: Acceso uniforme
#include <iostream>
#include <vector>
using namespace std;
class Iterator { public: virtual bool hasNext() = 0; virtual string next() = 0; };
class TaskList {
vector<string> tasks = {"Estudiar", "Cocinar", "Dormir"};
public:
Iterator* createIterator() {
return new class : public Iterator {
size_t i = 0;
bool hasNext() override { return i < 3; }
string next() override { return tasks[i++]; }
};
}
};
int main() {
TaskList list;
Iterator* it = list.createIterator();
while (it->hasNext()) cout << it->next() << " ";
delete it;
cout << endl;
}


EJEMPLOS
.vscode
comando.cpp
Iterator.cpp
Mediator.cpp
Observer.cpp
repair_center.py
State.cpp
Strategy.cpp
Mediator.cpp > ...
#include <iostream>
using namespace std;
class Mediator { public: virtual void send(string msg, class User*) = 0; };
class User {
Mediator* chat;
string name;
public:
User(Mediator* m, string n) : chat(m), name(n) {}
void send(string msg) { chat->send(msg, this); }
void receive(string msg) { cout << name << " ← " << msg << endl; }
string getName() { return name; }
};
class ChatRoom : public Mediator {
public:
void send(string msg, User* sender) override {
cout << sender->getName() << " → " << msg << endl;
}
};
int main() {
ChatRoom room;
User ana(&room, "Ana"), luis(&room, "Luis");
ana.send("Hola!"); // Ana → Hola!
luis.send("Chau!"); // Luis → Chau!
}


EJEMPLOS
.vscode
comando.cpp
Iterator.cpp
Mediator.cpp
Observer.cpp
repair_center.py
State.cpp
Strategy.cpp
Template Method.cpp
Template Method.cpp > main()
// Problema: Bebidas con pasos fijos pero variables
// Patrón Template: Estructura fija, pasos redefinibles
#include <iostream>
using namespace std;
class Beverage {
public:
void prepare() { boilWater(); brew(); pour(); addCondiments(); }
virtual void brew() = 0;
virtual void addCondiments() = 0;
private:
void boilWater() { cout << "Hirviendo agua\n"; }
void pour() { cout << "Sirviendo en taza\n"; }
};
class Coffee : public Beverage {
void brew() override { cout << "Filtrando café\n"; }
void addCondiments() override { cout << "Azúcar y leche\n"; }
};
int main() {
Coffee c; c.prepare();
}


EXPLORER
EJEMPLOS
.vscode
Chain of Responsibility.cpp
comando.cpp
Iterator.cpp
Mediator.cpp
Observer.cpp
repair_center.py
State.cpp
Strategy.cpp
Template Method.cpp
Visitor.cpp
comando.cpp
Observer.cpp
State.cpp
Strategy.cpp
Iterator.cpp
Chain of Responsibility.cpp > main()
// Problema: Enrutar ticket a soporte correcto
// Patrón Chain: Pasa hasta que alguien lo resuelva
#include <iostream>
using namespace std;
class Handler {
Handler* next = nullptr;
public:
void setNext(Handler* h) { next = h; }
virtual void handle(string issue) { if (next) next->handle(issue); }
};
class Tech : public Handler {
public:
void handle(string issue) override {
if (issue.find("error") != string::npos) cout << "Técnico: ok\n";
else Handler::handle(issue);
}
};
class Billing : public Handler {
public:
void handle(string issue) override {
if (issue.find("pago") != string::npos) cout << "Facturación: ok\n";
else cout << "No resuelto\n";
}
};
int main() {
Tech t; Billing b;
t.setNext(&b);
t.handle("error login"); // Técnico: ok
t.handle("pago atrasado"); // Facturación: ok
t.handle("hola"); // No resuelto
}



4. Que rol juega la Interfaz en los Patrones de Comportamiento, que facilita?

En los patrones de comportamiento, la interfaz juega un rol fundamental. Ayuda a definir y estandarizar las relaciones y colaboraciones entre los objetos.

Facilita:

Abstracción y Encapsulación: La interfaz define un conjunto de métodos (un contrato) que las clases concretas deben implementar. Esto oculta los detalles complejos de la implementación y encapsula el comportamiento específico dentro de cada clase.

Estandarización: Asegura que todas las clases que implementan la interfaz (por ejemplo, todas las "Estrategias" o todos los "Observadores") tengan un conjunto común de métodos. Esto promueve la consistencia.

Polimorfismo: Este es uno de los mayores beneficios. Permite que las clases que implementan la misma interfaz sean tratadas de manera uniforme. Por ejemplo, un objeto Context no necesita saber si está usando ConcreteStrategyA o ConcreteStrategyB; solo sabe que está usando un objeto que cumple con la interfaz Strategy.

Desacoplamiento: Este es el resultado principal. Al interactuar con objetos a través de interfaces en lugar de sus clases concretas, se reduce el acoplamiento. Esto hace que el sistema sea mucho más flexible y fácil de mantener, ya que los cambios en una clase concreta (como arreglar un algoritmo en ConcreteStrategyA) no afectan al Context ni a otras estrategias.

MARCO TEÓRICO

Desarrollar el Problema expresado debajo e Implementarlo en C++.

REPASO STRATEGY - ¿Que es?

¿En qué consiste el strategy pattern?

El strategy pattern pertenece a los llamados behavioural patterns o patrones de comportamiento, que equipan un software con diferentes métodos de resolución. Estas estrategias consisten en una familia de algoritmos que están separados del programa real y son autónomos, es decir intercambiables. Un strategy pattern también incluye algunas pautas y ayudas para los desarrolladores. Por ejemplo, los strategy patterns describen cómo construir u organizar un grupo de clases y crear objetos. Una característica especial del patrón strategy es que un comportamiento variable de programas y objetos también se puede realizar en el tiempo de ejecución de un software.

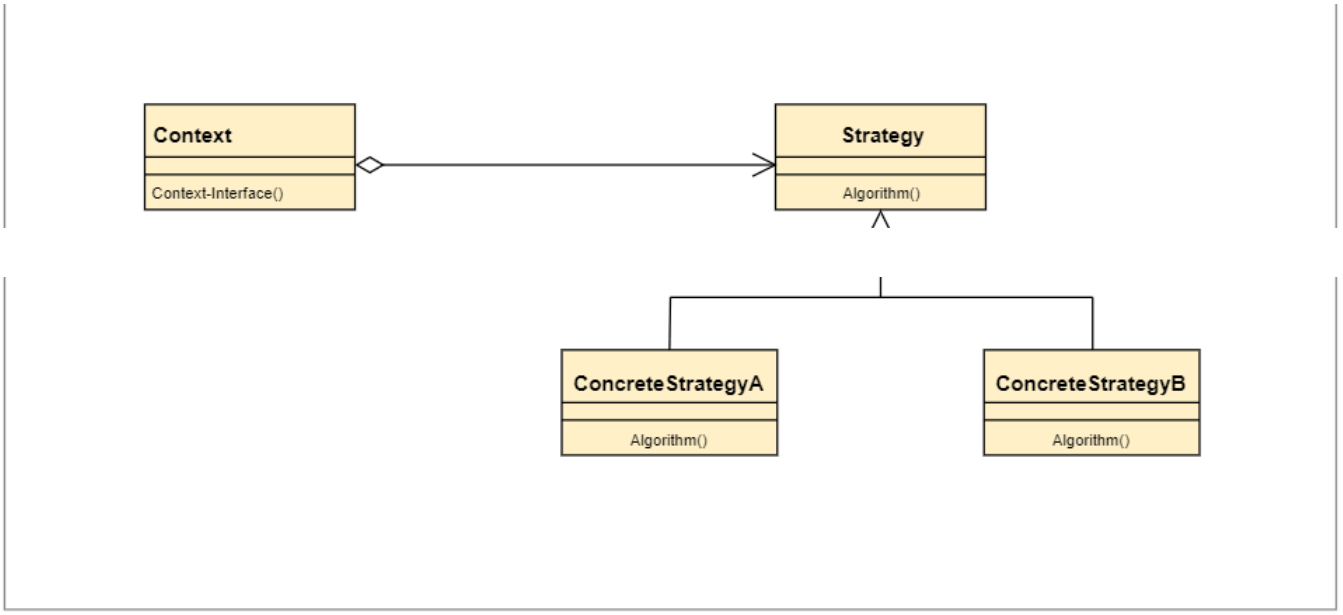
¿Cómo es la representación UML de un strategy pattern?

Los strategy patterns se diseñan generalmente con el lenguaje de modelado gráfico UML (Unified Modeling Language). Sirven para visualizar los patrones de diseño con una notación estandarizada y utilizan caracteres y símbolos especiales. El UML establece distintos tipos de diagramas para la programación orientada a objetos. Para representar un strategy pattern, se suelen utilizar los llamados diagramas de clase con al menos tres componentes básicos:

Context (contexto o clase de contexto)

Strategy (estrategia o clase de estrategia)

ConcreteStrategy (estrategia concreta)



Estructura básica de un patrón strategy en UML con tres componentes básicos: Context (clase principal), Strategy (interfaz) y ConcreteStrategies (algoritmos externalizados y especificaciones de solución para la resolución de problemas concretos).

Con el strategy pattern, los componentes básicos asumen funciones especiales. Los patrones de comportamiento de la clase Context se almacenan en diferentes clases Strategy. Estas clases separadas contienen los algoritmos llamados ConcreteStrategies. Utilizando una referencia interna, el contexto puede acceder a las variables de computación externalizadas (ConcreteStrategyA, ConcreteStrategyB, etc.) si lo necesita. No interactúa directamente con los algoritmos, sino con una interfaz.

La interfaz Strategy encapsula las variantes de cálculo y puede ser implementada simultáneamente por todos los algoritmos. Para la interacción con Context, la interfaz genérica proporciona solo un método para activar los algoritmos de ConcreteStrategy. La interacción con Context incluye, además del acceso a Strategy, intercambio de datos. La interfaz Strategy también participa en los cambios de estrategia, que pueden tener lugar, además, en el tiempo de ejecución del programa.

Repaso

La encapsulación impide el acceso directo a los algoritmos y a estructuras de datos internos. Una instancia externa (Client, Context) solo puede acceder a los cálculos y funciones a través de interfaces definidas y, además, solo puede acceder a los métodos y elementos de datos de un objeto que sean relevantes para ella.

¿Cuáles son las ventajas y e inconvenientes del strategy pattern?

Las ventajas de usar un strategy pattern son más evidentes desde la perspectiva de un programador y administrador de sistemas. El desglose en módulos y clases autónomas ayuda a estructurar mejor el código del programa. Puesto que los módulos estén delimitados en nuestra aplicación de ejemplo, la tarea del programador será más sencilla. Así pues, se puede limitar el alcance de la clase Navigator mediante la externalización de las estrategias y se puede prescindir de la creación de subclases en Context.

Las dependencias internas de los segmentos se mantienen dentro de los límites de un código más reducido y claramente definido. Por ello, los cambios tienen menos efecto, es decir, no suelen conllevar más cambios (que requieren mucho tiempo) en la programación. En algunos casos, los cambios derivados incluso pueden descartarse por completo. Los segmentos de código más claros también pueden mantenerse mejor a largo plazo y el diagnóstico y la resolución de problemas se facilitan.

Asimismo, el manejo se hace más sencillo, ya que la aplicación de ejemplo se puede equipar con una interfaz fácil de usar. Los usuarios pueden usar los botones para controlar el comportamiento del programa (cálculo de la ruta) de forma variable y elegir entre las opciones de forma sencilla.

Puesto que el Context de la aplicación de navegación solo interactúa con una interfaz que encapsula los algoritmos, es independiente de la aplicación concreta de los algoritmos individuales. Por lo tanto, si se modifican los algoritmos o se introducen nuevas estrategias posteriormente, no es necesario cambiar el código de Context. Esto permite ampliar las funciones de cálculo de ruta con estrategias

concretas adicionales para rutas en avión, transporte marítimo y tráfico de larga distancias. Las nuevas estrategias solo tienen que implementar la interfaz de Strategy correctamente.

Los strategy patterns simplifican la difícil programación del software orientado a objetos gracias a otra de sus virtudes: permiten el diseño de software reutilizable (módulos) que se puede implementar repetidamente y cuyo desarrollo se considera particularmente difícil. Esto significa que las clases Context relacionadas también podrían utilizar las estrategias externalizadas para calcular las rutas a través de la interfaz y ya no tendrían que aplicarlas por sí mismas.

A pesar de sus muchas ventajas, el strategy pattern también tiene algunos inconvenientes. Debido a su estructura más compleja, el diseño del software puede crear redundancias e ineficiencias en la comunicación interna. Por ejemplo, la interfaz strategy genérica, que todos los algoritmos deben aplicar por igual, a veces puede acabar sobredimensionada.

Un ejemplo: una vez que Context ha creado e inicializado ciertos parámetros, los pasa a la interfaz genérica y al método definido en esta. Sin embargo, la estrategia que se aplique en última instancia no necesita necesariamente todos los parámetros comunicados de Context y, por lo tanto, no los procesa. Así, la interfaz proporcionada no siempre se utiliza de manera óptima en el strategy pattern y a veces se producen transferencias de datos innecesarias, con el consiguiente esfuerzo de comunicación.

En la aplicación, también existe una estrecha dependencia interna entre el cliente y las estrategias. Client hace la selección y solicita la estrategia concreta mediante el comando de activación (en nuestro ejemplo, el cálculo de la ruta a pie), por lo que debe conocer las ConcreteStrategies. Por lo tanto, el patrón de diseño strategy solo debe utilizarse si los cambios de estrategia y comportamiento son importantes para el uso y la funcionalidad de un software.

Naturalmente, los inconvenientes mencionados se pueden compensar parcialmente o minimizar. Por ejemplo, el número de instancias de objetos que pueden producirse en grandes cantidades en el strategy pattern puede reducirse si se aplican a un flyweight pattern. La medida también tiene un efecto positivo en los requisitos de eficiencia y en la memoria de la aplicación.

¿Dónde se utiliza el patrón strategy?

Como patrón de diseño básico en el desarrollo de software, el strategy patternno está limitado a un solo ámbito de aplicación. Lo más importante a la hora de escoger este patrón de diseño es la naturaleza del problema que se quiera resolver. El patrón strategy es ideal para software que ha de resolver tareas y problemas variables, opciones de comportamiento y cambios.

Por ejemplo, los programas que ofrecen diferentes formatos de almacenamiento de archivos o varias funciones de clasificación y búsqueda utilizan el strategy pattern. En el ámbito de la compresión de datos también se utilizan programas que emplean diversos algoritmos de compresión basados en este patrón de diseño. Esto permite, por ejemplo, convertir de forma versátil los vídeos a un formato de archivo que ahorre espacio o restaurar archivos comprimidos (por ejemplo, archivos ZIP o RAR) a su estado original mediante estrategias especiales de descompresión. Otro ejemplo es el almacenamiento de un documento o archivo gráfico en diferentes formatos.

El patrón de diseño strategy también se utiliza en el desarrollo e implementación de software de juegos, que, por ejemplo, tienen que reaccionar con flexibilidad a las situaciones cambiantes del juego durante el tiempo de ejecución. Los diferentes personajes, los equipos especiales, los patrones de comportamiento de los personajes o sus movimientos especiales pueden almacenarse en forma de ConcreteStrategies.

Otra área de aplicación de los strategy patterns es el software de control.Mediante el intercambio de ConcreteStrategies, los segmentos de cálculo pueden adaptarse fácilmente a grupos ocupacionales, países y regiones. También los programas que traducen los datos a diferentes formatos gráficos (por ejemplo, como gráficos de líneas, circulares o de barras) utilizan strategy patterns.

Se pueden encontrar aplicaciones más específicas de los strategy patterns en la biblioteca estándar de Java (Java API) y en los conjuntos de herramientas de la interfaz gráfica de Java (por ejemplo, AWT, Swing y SWT), que utilizan un gestor de diseño en el desarrollo y la generación de interfaces gráficas de usuario. Este gestor puede aplicar diferentes estrategias para la disposición de los

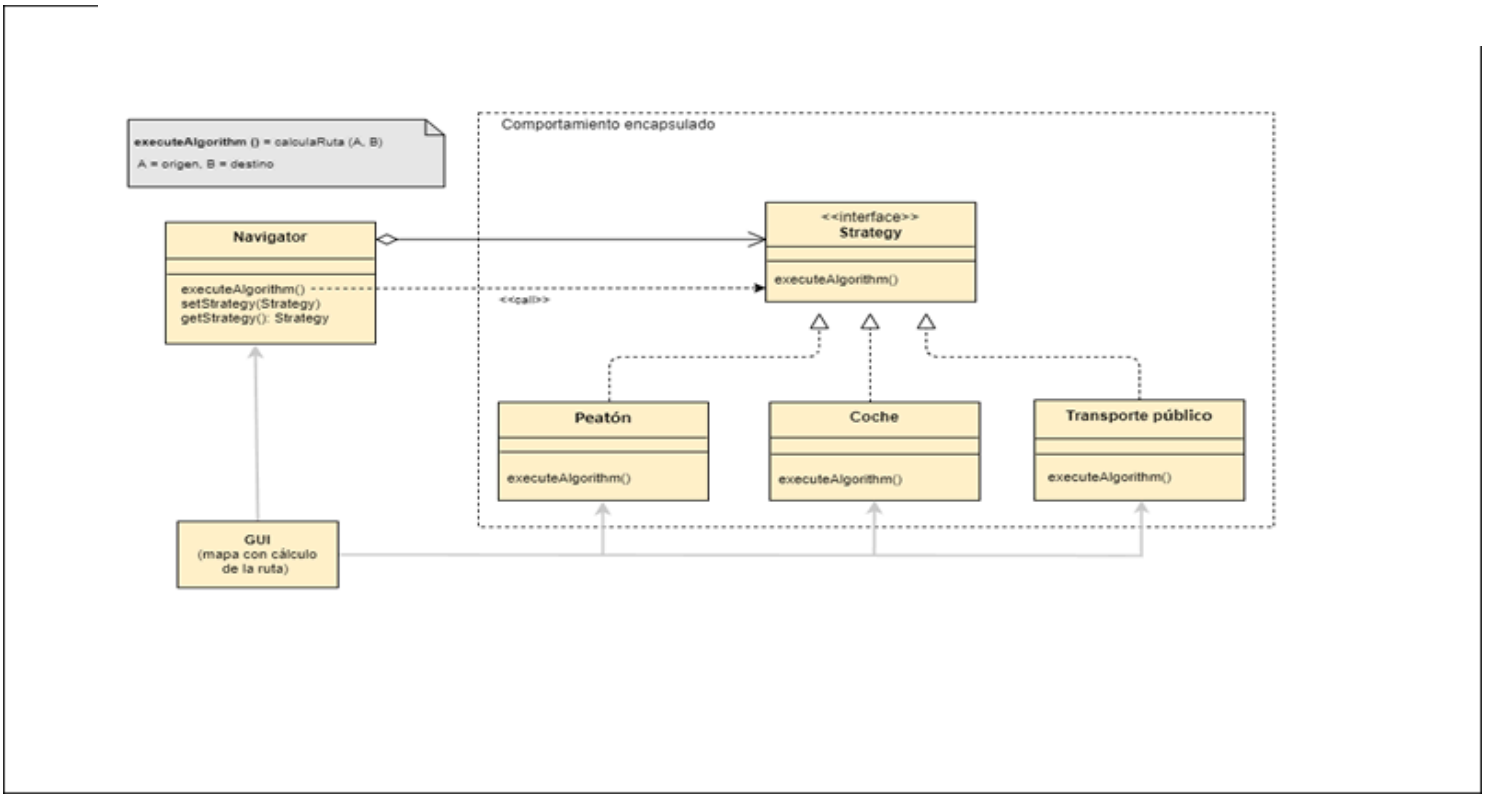
componentes durante el desarrollo de la interfaz. El strategy pattern se utiliza también en sistemas de base de datos, controladores de dispositivos y programas de servidores.

Modulo Practico

Desarrollar una Aplicación que deba calcular una ruta basada en los medios de transporte habituales. El usuario puede elegir entre tres opciones:

- Peatón (ConcreteStrategyA)
- Coche (ConcreteStrategyB)
- Transporte público (ConcreteStrategyC)

Si se transfieren estas especificaciones a un gráfico UML, se puede observar la estructura y la función del strategy pattern:



Strategy pattern en el diseño de una aplicación de navegación en UML: la clase Context recibe un comando del Client (cliente) de la aplicación (Consola) y luego establece la estrategia requerida. La interacción con los algoritmos de estrategia encapsulados (ConcreteStrategies) tiene lugar a través de la interfaz Strategy. Para poder solicitar una estrategia concreta, el Client debe conocer todas sus implementaciones.

En nuestro ejemplo, el cliente o Client es la Interfaz de Consola (Podría ser la GUI) de una aplicación de navegación con opciones para el cálculo de la ruta. Si el usuario hace una selección (o pulsa un botón en una GUI), se calcula una ruta concreta. El Context (clase Navigator) tiene la tarea de calcular y mostrar una serie de puntos de control en el mapa. La clase Navigator tiene un método para cambiar la estrategia de enrutamiento activa. Los botones permiten cambiar fácilmente entre los medios de transporte.

Si, por ejemplo, se activa un comando correspondiente con la Opcion peatonal del Client, se solicita el servicio “Calcular la ruta peatonal” (ConcreteStrategyA). El método executeAlgorithm() (en nuestro ejemplo, el método: calculaRuta(A, B)) acepta un origen y un destino y devuelve un conjunto de los puntos de control de la ruta. Context acepta el comando del Client y decide la estrategia apropiada basándose en las directivas previamente definidas (policy) (setStrategy: peatonal). Mediante Call, delega la solicitud al objeto Strategy y a su interfaz.

Con getStrategy(), la estrategia actualmente seleccionada se almacena en Context (clase Navigator). Los resultados de los cálculos de ConcreteStrategy se utilizan para el procesamiento posterior y en la visualización por consola de la ruta en la aplicación de navegación. Si el usuario elige una ruta diferente, por ejemplo, haciendo clic en la Opcion “Coche”, Context cambia a la estrategia solicitada (ConcreteStrategyB) e inicia un nuevo cálculo a través de otra llamada. Al final del procedimiento, se devuelve una descripción de ruta modificada para el coche.

Se solicita Implementar este Ejem plo en C++.

El usuario elige un medio de transporte → se calcula una ruta óptima → se muestra por consola.

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;
// Punto en el mapa (lat, lng)
struct Point {
    string name;
    Point(string n) : name(n) {}
};
// === INTERFAZ ESTRATEGIA ===
class RoutingStrategy {
public:
    virtual vector<Point> calculateRoute(const Point& from, const Point& to) = 0;
    virtual ~RoutingStrategy() = default;
};
// === ESTRATEGIAS CONCRETAS ===
// 1. Peatón
class PedestrianStrategy : public RoutingStrategy {
public:
    vector<Point> calculateRoute(const Point& from, const Point& to) override {
        cout << "Ruta PEATÓN: evitando autopistas, usando veredas\n";
        return { from, Point("Calle 1"), Point("Parque"), to };
    }
};
// 2. Coche
class CarStrategy : public RoutingStrategy {
public:
    vector<Point> calculateRoute(const Point& from, const Point& to) override {
        cout << "Ruta COCHE: usando autopistas y calles principales\n";
        return { from, Point("Autopista"), Point("Salida 5"), to };
    }
};
// 3. Transporte Público
class PublicTransportStrategy : public RoutingStrategy {
public:
    vector<Point> calculateRoute(const Point& from, const Point& to) override {
        cout << "Ruta TRANSPORTE PÚBLICO: usando metro y colectivos\n";
        return { from, Point("Estación A"), Point("Linea 1"), Point("Estación B"), to };
    }
};
```

```cpp
// === CONTEXTO (Navigator) ===
class Navigator {
    RoutingStrategy* strategy = nullptr;

public:
    void setStrategy(RoutingStrategy* s) {
        strategy = s;
    }

    RoutingStrategy* getStrategy() const {
        return strategy;
    }

    void navigate(const Point& from, const Point& to) {
        if (!strategy) {
            cout << "Error: No se ha seleccionado un medio de transporte.\n";
            return;
        }

        cout << "\n--- Calculando ruta de " << from.name << " a " << to.name << " ---\n";
        vector<Point> route = strategy->calculateRoute(from, to);

        cout << "Ruta calculada:\n";
        for (size_t i = 0; i < route.size(); ++i) {
            cout << "  " << (i + 1) << ". " << route[i].name << endl;
        }
        cout << "----------------------------------------\n\n";
    }
};
```

```cpp
// === CLIENTE (Interfaz de Consola) ===
int main() {
    Navigator app;
    // Estrategias disponibles
    PedestrianStrategy walk;
    CarStrategy car;
    PublicTransportStrategy bus;
    Point home("Casa");
    Point work("Oficina");
    int opcion;
    do {
        cout << "=== NAVEGADOR ===" << endl;
        cout << "1. Peatón" << endl;
        cout << "2. Coche" << endl;
        cout << "3. Transporte Público" << endl;
        cout << "0. Salir" << endl;
        cout << "Elige medio de transporte: ";
        cin >> opcion;

        switch (opcion) {
            case 1:
                app.setStrategy(&walk);
                app.navigate(home, work);
                break;
            case 2:
                app.setStrategy(&car);
                app.navigate(home, work);
                break;
            case 3:
                app.setStrategy(&bus);
                app.navigate(home, work);
                break;
            case 0:
                cout << "Saliendo...\n";
                break;
            default:
                cout << "Opción inválida.\n";
        }
    } while (opcion != 0);
    return 0;
}
```

```
=== NAVEGADOR ===
1. Peatón
2. Coche
3. Transporte Público
0. Salir
Elige medio de transporte: 1

--- Calculando ruta de Casa a Oficina ---
Ruta PEATÓN: evitando autopistas, usando veredas
Ruta calculada:
  1. Casa
  2. Calle 1
  3. Parque
  4. Oficina
----------------------------------------

Elige medio de transporte: 2

--- Calculando ruta de Casa a Oficina ---
Ruta COCHE: usando autopistas y calles principales
Ruta calculada:
  1. Casa
  2. Autopista
  3. Salida 5
  4. Oficina
----------------------------------------

Elige medio de transporte: 0
Saliendo...
```

Lic. Oemig José Luis.