Algoritmos y Estructuras de Datos II

Trabajo Práctico N° 1	Unidad 1-002
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

Responder el siguiente cuestionario en función de la bibliografía Obligatoria.

1. Describir el Tipos de Patrón Estructural.

Los patrones estructurales son un tipo de patrón de diseño que explica cómo ensamblar objetos y clases en estructuras más grandes. Su objetivo es mantener la flexibilidad y la eficiencia de la estructura resultante.
Los tipos de patrones estructurales mencionados en la bibliografía son:

- Adapter (Adaptador)
- Bridge (Puente)
- Composite (Compuesto)
- Decorator (Decorador)
- Facade (Fachada)
- Flyweight (Peso mosca)
- Proxy
- Plantilla curiosamente recurrente (CRTP)
- Programación basada en interfaces (IBP)

2. ¿Qué “tipos” de Problema resuelve.

Los patrones estructurales resuelven problemas relacionados con la composición y organización de clases y objetos.
Por ejemplo:
Adapter resuelve el problema de incompatibilidad. Permite que clases con interfaces incompatibles trabajen juntas, actuando como un traductor entre ellas.

Decorator resuelve la necesidad de añadir funcionalidades o responsabilidades extra a un objeto de forma dinámica, en tiempo de ejecución, sin tener que recurrir a la herencia (que es estática).

Composite resuelve el problema de querer tratar a un grupo de objetos (como una estructura de árbol) de la misma manera que a un objeto individual. Permite componer objetos en estructuras de árbol y trabajar con ellas como si fuesen un solo objeto.

Proxy resuelve la necesidad de controlar el acceso a un objeto. Se usa para crear un sustituto o representante que permite hacer algo antes o después de que la solicitud llegue al objeto original, como en la inicialización diferida (crear un objeto pesado solo cuando se usa) o el control de acceso.

3. Dar ejemplos de Adapter o Decorator en el TP “Ticket to Ride”.

```cpp
// Sistema nuevo: usa "Card" con getColor(), getValue()
// Sistema viejo: usa "OldCard" con color(), points()

#include <string>
class OldCard {
public:
    string color() const { return "Rojo"; }
    int points() const { return 5; }
};

// Interfaz moderna
class Card {
public:
    virtual string getColor() const = 0;
    virtual int getValue() const = 0;
    virtual ~Card() = default;
};

// ADAPTER: permite usar OldCard en sistema nuevo
class OldCardAdapter : public Card {
    OldCard old;
public:
    string getColor() const override { return old.color(); }
    int getValue() const override { return old.points(); }
};
```

```cpp
class Route {
public:
    virtual int getPoints() const = 0;
    virtual ~Route() = default;
};

class BasicRoute : public Route {
    int length;
public:
    BasicRoute(int l) : length(l) {}
    int getPoints() const override { return length * 2; }
};

// DECORATOR: añade bonos
class BonusDecorator : public Route {
    Route* route;
    int bonus;
public:
    BonusDecorator(Route* r, int b) : route(r), bonus(b) {}
    int getPoints() const override { return route->getPoints() + bonus; }
};
```

4. Dar 1 ejemplo del Patrón Composite.

```cpp
#include <iostream>
#include <vector>
using namespace std;

class Furniture {
public:
    virtual void show() const = 0;
    virtual ~Furniture() = default;
};

class Part : public Furniture {
    string name;
public:
    Part(string n) : name(n) {}
    void show() const override { cout << "  - " << name << endl; }
};

class Assembly : public Furniture {
    string name;
    vector<Furniture*> items;
public:
    Assembly(string n) : name(n) {}
    void add(Furniture* f) { items.push_back(f); }
    void show() const override {
        cout << name << ":\n";
        for (auto* i : items) i->show();
    }
};

int main() {
    Part* leg = new Part("Pata");
    Part* seat = new Part("Asiento");

    Assembly* chair = new Assembly("Silla");
    chair->add(leg);
    chair->add(seat);
    chair->add(new Part("Respaldo"));

    Assembly* table = new Assembly("Mesa");
    table->add(leg);  // ¡Comparte la pata!

    chair->show();
    cout << endl;
    table->show();

    delete chair; delete table; delete leg; delete seat;
}
```

```
Silla:
  - Pata
  - Asiento
  - Respaldo

Mesa:
  - Pata
```

5. ¿Qué significa en C++ Mixin-Style?

La técnica de **"Mixin-Style"** (o "Mixins") se describe en la presentación en el contexto del patrón "Plantilla curiosamente recurrente".
Un Mixin es una clase que ofrece una funcionalidad específica para ser heredada por una subclase. La clave es que esta clase no está diseñada para ser autónoma (usarse sola).
Heredar de un mixin no es una forma de especialización (como decir "un Círculo es una Forma"), sino que es un medio para obtener u "heredar" funcionalidad. Una subclase puede incluso usar herencia múltiple para heredar de varios mixins y combinar sus funcionalidades.

6. ¿Por qué Programar con Interfaces? ¿Qué Proporciona o Facilita?

La programación basada en interfaces (IBP) define la aplicación como una colección de módulos que se conectan e interactúan entre sí a través de sus interfaces.

Programar de esta manera proporciona o facilita:

Mantenibilidad y Modularidad: Permite que los módulos (partes del sistema) puedan ser desconectados, reemplazados o actualizados sin comprometer o afectar a los otros módulos. Esto aumenta la modularidad y, por lo tanto, la capacidad de mantenimiento de la aplicación.

Reducción de la Complejidad: Al dividir el sistema en componentes que cooperan a través de interfaces, la complejidad total se reduce considerablemente.

Trabajo en Equipo: Es muy útil cuando diferentes equipos desarrollan diferentes módulos.

Extensibilidad: Facilita que terceros desarrollen componentes adicionales. Esos desarrolladores solo necesitan crear componentes que cumplan con la interfaz especificada, sin necesitar conocer el resto del sistema.

La interfaz actúa como un "acuerdo contractual": el proveedor asegura que no cambiará la interfaz, y el suscriptor (quien la implementa) acepta implementarla en su totalidad

Marco Práctico

1. Supongamos que nos encargan construir una aplicación para poder buscar de hoteles, debemos brindar un presupuesto según las características que se seleccionen. Además de los hoteles, se desea que en el futuro se oferten diferentes tipos de alojamientos (contemplar la extensibilidad) , como apartamentos u Hostels.
   Construir una aplicación en C++ que permita la extensibilidad de Tipos de Hoteles sin afectar el código, e implementar un patrón Decorator para Tal fin.

   Patrones de Diseño (X): Patrones Estructurales - Decorator. Programación en Castellano. (programacion.net)

```cpp
#include <iostream>
#include <string>
using namespace std;

// ========================================
// 1. COMPONENTE BASE (Alojamiento)
// ========================================
class Accommodation {
public:
    virtual double getPrice() const = 0;
    virtual string getDescription() const = 0;
    virtual ~Accommodation() = default;
};

// ========================================
// 2. ALOJAMIENTOS BÁSICOS (EXTENSIBLE)
// ========================================
class Hotel : public Accommodation {
    string name;
    double basePrice;
public:
    Hotel(string n, double p) : name(n), basePrice(p) {}
    double getPrice() const override { return basePrice; }
    string getDescription() const override { return "Hotel " + name; }
};

class Apartment : public Accommodation {
    double getPrice() const override { return 80.0; }
    string getDescription() const override { return "Apartamento"; }
};

class Hostel : public Accommodation {
    double getPrice() const override { return 30.0; }
    string getDescription() const override { return "Hostel"; }
};

```

```cpp
// ========================================
// 3. DECORATOR: Añade extras al presupuesto
// ========================================
class AccommodationDecorator : public Accommodation {
protected:
    Accommodation* base;
public:
    AccommodationDecorator(Accommodation* b) : base(b) {}
    double getPrice() const override { return base->getPrice(); }
    string getDescription() const override { return base->getDescription(); }
};

// Extras concretos
class WiFi : public AccommodationDecorator {
public:
    WiFi(Accommodation* b) : AccommodationDecorator(b) {}
    double getPrice() const override { return base->getPrice() + 10; }
    string getDescription() const override { return base->getDescription() + " + WiFi"; }
};

class Breakfast : public AccommodationDecorator {
public:
    Breakfast(Accommodation* b) : AccommodationDecorator(b) {}
    double getPrice() const override { return base->getPrice() + 15; }
    string getDescription() const override { return base->getDescription() + " + Desayuno"; }
};

class Parking : public AccommodationDecorator {
public:
    Parking(Accommodation* b) : AccommodationDecorator(b) {}
    double getPrice() const override { return base->getPrice() + 20; }
    string getDescription() const override { return base->getDescription() + " + Parking"; }
};
```

```cpp
// ========================================
// 4. MAIN - PRESUPUESTO
// ========================================
int main() {
    // Cliente elige
    Accommodation* stay = new Hotel("Palace", 120);
    stay = new WiFi(stay);
    stay = new Breakfast(stay);

    cout << stay->getDescription() << endl;
    cout << "Presupuesto: $" << stay->getPrice() << endl;

    delete stay;
    return 0;
}
```

Lic. Oemig José Luis.