CS2013: Programación III

Laboratorio: Patrones de Diseño I

Laboratorio 10A

José Chávez

Agenda

- Patrones de Creación

Ejercicios

Ejercicio 1: Singleton

Implementa una clase **`Logger`** utilizando el patrón Singleton. La clase Logger debería tener un método llamado **`log()`** que tome un mensaje como entrada y lo imprima en la consola. Asegúrate de que solo se pueda crear y acceder a una única instancia de la clase Logger en todo el programa.

Completar los espacios en blanco del siguiente programa.

Ejercicio 1: Singleton

```
class Logger {
public:
    // Atributo estático para almacenar la instancia única

    static Logger getInstance() {
        // Implementar el método para obtener la instancia única
    }

    void log(const std::string& message) {
        // Implementar la lógica para imprimir el mensaje en la consola
    }
private:
    Logger() {} // Constructor privado para evitar la creación de instancias
};
```

Ejercicio 1: Singleton

```
int main() {
    Logger logger = Logger::getInstance();
    logger.log("Este es un mensaje de prueba.");

    // Resultado esperado en consola:
    //
    // Este es un mensaje de prueba.

    return 0;
}
```

Ejercicio 2: Abstract Factory

Crea una fábrica abstracta llamada **`ShapeFactory`** que pueda producir diferentes tipos de formas, como **`Circle`** (círculo) y **`Square`** (cuadrado).

Cada forma debería tener un método llamado **`draw()`** que imprima un mensaje indicando la forma que se está dibujando. Implementa fábricas concretas **`CircleFactory`** y **`SquareFactory`** que produzcan instancias de **`Circle`** y **`Square`** respectivamente. Crea un programa cliente que utilice la fábrica abstracta para crear y dibujar formas.

Completar los espacios en blanco del siguiente programa.

Ejercicio 2: Abstract Factory

```
// Interfaz para las formas
class Shape {
public:
    virtual void draw() = 0;
};

// Clase concreta para el círculo
class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Dibujando un círculo." << std::endl;
    }
};

// Clase concreta para el cuadrado
class Square : public Shape {
public:
    void draw() override {
        std::cout << "Dibujando un cuadrado." << std::endl;
    }
};
```

Ejercicio 2: Abstract Factory

```
// Fábrica abstracta
class ShapeFactory {
public:
    virtual Shape* createShape() = 0;
};

class CircleFactory : public ShapeFactory {
public:
    Shape* createShape() override {
        // TODO: Implementar la creación de un círculo
    }
};

class SquareFactory : public ShapeFactory {
public:
    Shape* createShape() override {
        // TODO: Implementar la creación de un cuadrado
    }
};
```

Ejercicio 2: Abstract Factory

```
int main() {
    // Implementar un circulo usando la fábrica

    Shape* shape = factory->createShape();
    shape->draw();

    // Resultado esperado:
    // Dibujando un círculo.

    delete shape;
    delete factory;

    return 0;
}
```

Ejercicio 3: Factory Method

Implementar el patrón Factory Method para crear diferentes tipos de personajes. Completa el código de las clases **`Character`** y **`CharacterCreator`** para que puedan ser heredadas por diferentes tipos de personajes y creadores concretos respectivamente. Luego, completa la función **`main()`** para usar un creador concreto y crear un personaje, mostrando su información.

Ejercicio 3: Factory Method

```
// Clase abstracta Character
class Character {
public:
    virtual std::string getInfo() = 0;
};
// Clase concreta NonPlayerCharacter
class PlayerCharacter : public Character { // Clase concreta PlayerCharacter
public:
    std::string getInfo() override {
        return "Personaje del jugador";
    }
};

class NonPlayerCharacter : public Character { // Clase concreta NonPlayerCharacter
public:
    std::string getInfo() override {
        return "Personaje no jugador";
    }
};
```

Ejercicio 3: Factory Method

```
// Clase abstracta CharacterCreator
class CharacterCreator {
public:
    virtual Character* createCharacter() = 0;
};

class PlayerCharacterCreator : public CharacterCreator {
public:
    Character* createCharacter() override {
        // Crear y retornar una instancia de PlayerCharacter
    }
};

class NonPlayerCharacterCreator : public CharacterCreator {
public:
    Character* createCharacter() override {
        // Crear y retornar una instancia de NonPlayerCharacter
    }
};
```

Ejercicio 3: Factory Method

```
int main() {
    CharacterCreator* creator = new PlayerCharacterCreator();
    Character* character = creator->createCharacter();

    // Completa el código para mostrar la información del personaje

    // Resultado esperado en la consola:
    // Información del personaje: Personaje del jugador

    delete character;
    delete creator;

    return 0;
}
```

Ejercicio 4: Builder

En el desarrollo de un videojuego, implementa el patrón Builder para construir personajes de forma flexible. Completa el código de las clases **CharacterBuilder** y **CharacterDirector** para que puedan ser utilizadas para construir diferentes tipos de personajes con distintas características. Luego, completa la función **main()** para crear un personaje utilizando un director y un constructor concreto, mostrando su información.

Ejercicio 4: Builder

```
// Clase Character con atributos
class Character {
public:
    std::string name;
    std::string type;
    int level;
    // Completa con más atributos para el personaje
    // . . .

    void info() {
        cout << "Informacion del personaje:" << endl;
        cout << "Nombre: " << name << endl;
        cout << "Tipo: " << type << endl;
        cout << "Nivel: " << level << endl;
        // Completa para mostrar más atributos relevantes para el personaje
        // . . .
    }
};
```

Ejercicio 4: Builder

```
// Clase CharacterBuilder
class CharacterBuilder {
public:
    virtual void setName(const std::string& name) = 0;
    virtual void setType(const std::string& type) = 0;
    virtual void setLevel(int level) = 0;
    // Completa el código con métodos
    // para configurar más atributos relevantes para el personaje

    virtual Character* getCharacter() = 0;
};
```

Ejercicio 4: Builder

```
// Clase CharacterDirector
class CharacterDirector {
public:
    void constructCharacter(CharacterBuilder* builder) {
        builder->setName("Personaje");
        builder->setType("Tipo de personaje");
        builder->setLevel(1);
        // Completa el código para configurar
        // más atributos relevantes para el personaje
    }
};
```

Ejercicio 4: Builder

```
class ConcreteCharacterBuilder : public CharacterBuilder {
private:
    Character* character;
public:
    ConcreteCharacterBuilder() { character = new Character();}
    void setName(const std::string& name) override { character->name = name;}
    void setType(const std::string& type) override { character->type = type;}
    void setLevel(int level) override { character->level = level; }

    // Completa el código para configurar más atributos relevantes para el personaje

    Character* getCharacter() override {
        return character;
    }
};
```

Ejercicio 4: Builder

```
int main() {
    CharacterBuilder* builder = new ConcreteCharacterBuilder();
    CharacterDirector director;
    director.constructCharacter(builder);
    Character* character = builder->getCharacter();

    // Completa el código para mostrar la información del personaje

    // Resultado esperado en la consola:
    // Información del personaje:
    // Nombre: Personaje
    // Tipo: Tipo de personaje
    // Nivel: 1
    // ...

    delete character;
    delete builder;

    return 0;
}
```

Resumen

En esta sesión se trataron los tópicos siguientes:

- Patrones de Creación