Laboratorio 15

Patrones de Diseño: Builder

1. Competencias

1.1. Competencias del curso

Conoce, comprende e implementa programas con el lenguaje de programación C++ siguiendo la correcta utilización de los diferentes patrones de diseño.

1.2. Competencia del laboratorio

Conoce, comprende e implementa programas usando el lenguaje de programación C++, utilizando el patrón Builder

2. Equipos y Materiales

- Un computador.
- IDE para C++.
- Compilador para C++.

3. Marco Teórico

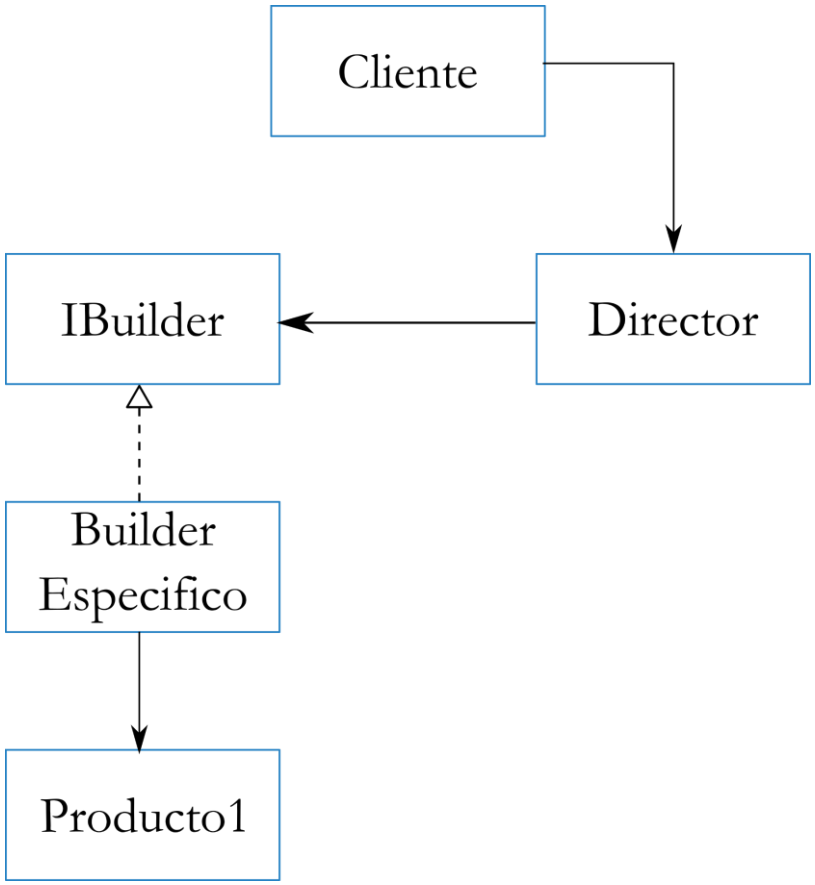
3.1. Introducción

El patrón Builder permite la construcción de objetos complejos paso a paso. Con la ayuda de este patrón se pueden producir diferentes representaciones de un objeto usando el mismo código.

Dado el problema de crear un objeto complejo en el cual hay varias características o parámetros que pueden ser inicializados de diferentes formas. Un Objeto A podría inicializar sólo dos parámetros y los demás vacíos. Un objeto B podría inicializar todos los parámetros disponibles, y un objeto C puede inicializar la mitad de parámetros.

De los tres Objetos (**A, B y C**), existirán paramentos que no sean utilizados o serán inicializados con poca frecuencia. Lo que será un problema en la creación de objetos.

En el siguiente ejemplo se muestra un código que permite utilizar el patrón Builder para evitar declarar cada una de las características de forma manual al momento de crear un nuevo objeto. Mientras un objeto vaya requiriendo de cierta cantidad de parámetros, estos se irán definiendo de acuerdo a la necesidad del objeto.



En la imagen anterior se muestra el diagrama de clases a utilizar.

La *Interfaz IBuilder*, permitirá tener la lista de funciones a las características disponibles que los objetos pueden seleccionar e implementar.

La clase *BuilderEspecifico* permite la selección e implementación de las diferentes características disponibles, estas serán devueltas como un objeto de la clase *Producto* donde se encontrarán las características solicitadas. En la clase *Producto*, se mantendrá la lista de características asignadas, así como operaciones para trabajar con ellas.

La clase *director* se encarga de ofrecer objetos construidos con cantidades diferentes de implementaciones.

Finalmente, la clase Cliente (*ClienteCode*) solicita diferentes tipos de objetos a la clase *director*, pudiendo incluso generar su propio tipo personalizado.

3.1.1. Clase IBuider

```
class IBuilder {
public:
    virtual ~IBuilder() {}
    virtual void ProducirParteA() const = 0;
    virtual void ProducirParteB() const = 0;
    virtual void ProducirParteC() const = 0;
};
```

3.1.2. Clase BuilderEspecifico

```
class BuilderEspecifico : public IBuilder {
private:
       Producto1* product;
public:
       BuilderEspecifico() {
           this->Reset();
       }
       ~BuilderEspecifico() {
           delete product;
       }
       void Reset() {
           this->product = new Producto1();
       }
       void ProducirParteA()const override {
           this->product->componentes.push_back("ParteA1");
       }
       void ProducirParteB()const override {
           this->product->componentes.push_back("ParteB1");

       }
```

```
        void ProducirParteC()const override {
            this->product->componentes.push_back("ParteC1");
        }
     Producto1* GetProducto() {
            Producto1* resultado = this->product;
            this->Reset();
            return resultado;
    }
};
```

3.1.3. Clase Producto1

```
class Producto1 {
public:
       std::vector<std::string> componentes;
       void ListaComp()const {
          std::cout << "Componentes : ";
          for (size_t i = 0; i < componentes.size(); i++) {
              if (componentes[i] == componentes.back()) {
                  std::cout << componentes[i];
              } else {
                  std::cout << componentes[i] << ", ";
              }
          }
          std::cout << "\n\n";
       }
};
```

3.1.4. Clase Director

```
class Director {
private:
       IBuilder* builder;
public:
```

```
        void set_builder(IBuilder* builder) {
            this->builder = builder;
        }
        void BuildProductoMin() {
            this->builder->ProducirParteA();
        }
        void BuildProductoCompleto() {
            this->builder->ProducirParteA();
            this->builder->ProducirParteB();
            this->builder->ProducirParteC();
    }
};
```

3.1.5. Función ClienteCode

```
void ClienteCode(Director& director)
{
        BuilderEspecifico* builder = new BuilderEspecifico();
        director.set_builder(builder);
        std::cout << "Producto Basico:\n";
        director.BuildProductoMin();

        Producto1* p = builder->GetProducto();
        p->ListaComp();
        delete p;

        std::cout << "Producto Completo:\n";
        director.BuildProductoCompleto();

        p = builder->GetProducto();
        p->ListaComp();
        delete p;

        std::cout << "Producto Personalizado:\n";
        builder->ProducirParteA();
```

```
        builder->ProducirParteC();
        p = builder->GetProducto();
        p->ListaComp();
        delete p;

        delete builder;
}
```

3.1.6. Main

```
int main() {
        Director* director = new Director();
        ClienteCode(*director);
        delete director;
        return 0;
}
```

La ejecución del código mostrará el resultado:

Producto Basico:
Componentes : ParteA1

Producto Completo:
Componentes : ParteA1, ParteB1, ParteC1

Producto Personalizado:
Componentes : ParteA1, ParteC1

1. Ejercicios

Resolver los siguientes ejercicios planteados:

1. El alumno deberá de implementar un conjunto de clases que permita seleccionar las piezas de un automóvil, es decir, se podrán tener componentes a disposición del cliente (puertas, llantas, timón, asientos, motor, espejos, vidrios, etc.). Del cual el cliente puede indicar que características de color puede tener cada pieza. Al

final mostrar opciones al Cliente o permitirle que él pueda escoger las piezas e indicar el color. Utilizar el patrón Builder.

*Pista, en lugar de trabajar el producto con una lista de componentes, se puede alojar una estructura o clase.

2. Entregables

Al final estudiante deberá:

1. Compactar el código elaborado y subirlo al aula virtual de trabajo. Agregue sus datos personales como comentario en cada archivo de código elaborado.
2. Elaborar un documento que incluya tanto el código como capturas de pantalla de la ejecución del programa. Este documento debe de estar en formato PDF.
3. El nombre del archivo (comprimido como el documento PDF), será su LAB13_GRUPO_A/B/C_CUI_1erNOMBRE_1erAPELLIDO.

   (Ejemplo: LAB13_GRUPO_A _2022123_PEDRO_VASQUEZ).
4. Debe remitir el documento ejecutable con el siguiente formato:

   LAB13_GRUPO_A/B/C_CUI_ EJECUTABLE_1erNOMBRE_1erAPELLIDO

   (Ejemplo: LAB13_GRUPO_A_EJECUTABLE_2022123_PEDRO_VASQUEZ).

En caso de encontrarse trabajos similares, los alumnos involucrados no tendrán evaluación y serán sujetos a sanción.