Laboratorio 15

Patrones de diseño: Builder

1. Competencias

1.1. Competencias del curso

Conoce, comprende e implementa programas con el lenguaje de programación C++.

1.2. Competencia del laboratorio

Conoce, comprende e implementa programas usando patrones de diseño del lenguaje de programación C++.

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 del mismo código.

Cuando un objeto es muy complejo, requiere muchos datos para inicializarse o está construido por muchos otros objetos, surgen problemas al tratar de instanciarlos.

Ejemplo 1:

```
Class Casa{
      Casa(bool puerta, int ventanas, bool garage, bool pileta, int arboles,
      int arbustos, int pisos, int habitaciones, int banios, bool comedor,
      bool living, techo tipo_techo, int escaleras);
   }
```

Si se tiene que construir un objeto, la lectura seria compleja y habrá parámetros que no se utilicen los cuales se llenan de 0:

```
...
# Main
Casa *miCasa = new Casa(true, 1, false, false, 1, 0, 2, 2, 2, 0, 0, plano, 1);
```

Ejemplo 2:

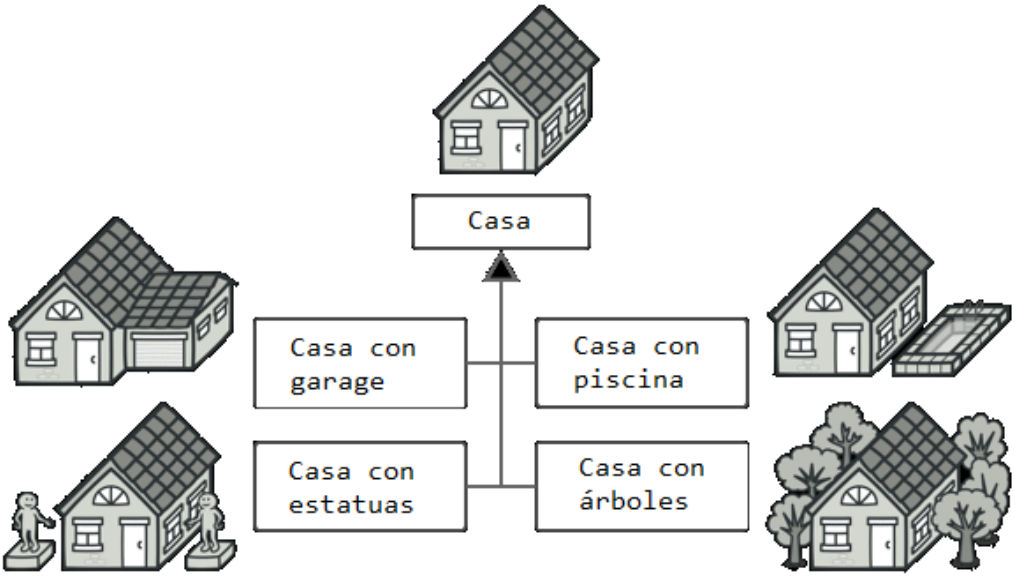
```
class Casa {
	Casa();
	addPuerta(puerta pue);
	addVentana(ventama ven);
	addGarage(Garage gar);
	addPileta(Pileta pil);
	add ...
}
```

El constructor se hace simple y después se agregan objetos de forma externa ya que son parte de Casa, el problema es que se define la casa con todo lo que debe tener, pero la responsabilidad de crear la casa ya es externa y puede que este disperso por todo el código y también puede que se olvide de crear alguna parte de la casa.

```
...
# Main
Casa *miCasa = new Casa();
miCasa.addPuerta(new Puerta());
miCasa.addVentana(new Ventana());
miCasa.addGarage(new Garage());
miCasa.addPileta(new Pileta());
miCasa.add ...
```

La intención es separar la construcción de objetos complejos de su representación. La idea no es que el constructor cree la casa ya que el objeto es grande y complejo como el ejemplo 1. Pero tampoco que este disperso la creación de objetos por todo el código o en subclases como en el ejemplo 2.

Crear una subclase por cada configuración posible de un objeto puede complicar demasiado el programa.

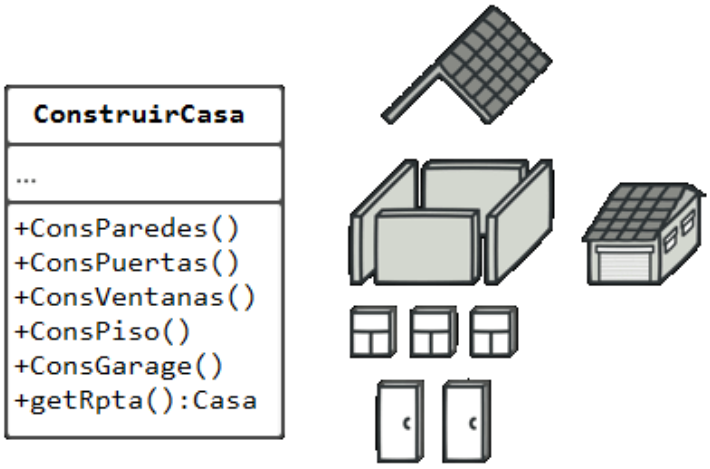


3.2. Patrón Builder

Se debe usar el patrón Builder cuando:

- El algoritmo para crear un objeto complejo debe ser independiente del objeto y de cómo se construye.
- El proceso de construcción debe permitir diferentes representaciones del objeto.

El patrón Builder sugiere que saques el código de construcción del objeto de su propia clase y lo coloques dentro de objetos independientes llamados constructores.



El patrón organiza la construcción de objetos en una serie de pasos (consParedes, consPuertas, etc.). Para crear un objeto, se ejecuta una serie de estos pasos en un objeto constructor. Lo importante es que no necesitas invocar todos los pasos. Puedes invocar sólo aquellos que sean necesarios para producir una configuración particular de un objeto.

Puede ser que algunos pasos de la construcción necesiten una implementación diferente cuando tengamos que construir distintas representaciones del producto. Por ejemplo, las paredes de una cabaña pueden ser de madera, pero las paredes de un castillo tienen que ser de piedra.

En este caso, podemos crear varias clases constructoras distintas que implementen la misma serie de pasos de construcción, pero de forma diferente. Entonces podemos utilizar estos constructores en el proceso de construcción (por ejemplo, una serie ordenada de llamadas a los pasos de construcción) para producir distintos tipos de objetos.

Los distintos constructores ejecutan la misma tarea de formas distintas.

Por ejemplo, imagina un constructor que construye todo de madera y vidrio, otro que construye todo con piedra y hierro y un tercero que utiliza oro y diamantes. Al invocar la misma serie de pasos, obtenemos una casa normal del primer constructor, un pequeño castillo del segundo y un palacio del tercero. Sin embargo, esto sólo funcionaría si el código cliente que invoca los pasos de construcción es capaz de interactuar con los constructores mediante una interfaz común.

Ejemplo 3:

```
class BuilderEspecifico{
public:
	BuilderEspecifico() {
		this->Reset();
	}
	void ProducirPuerta()const override {
		this->product->componentes.puerta("Puerta");
	}
	void ProducirPiso()const override {
		this->product->componentes.piso("Piso");
	}

}
```

Las clases constructoras concretas siguen la interfaz constructora y proporcionan implementaciones específicas de los pasos de construcción. El programa puede tener multitud de variaciones de objetos constructores, cada una de ellas implementada de forma diferente.

Ejemplo del Patrón Builder:

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

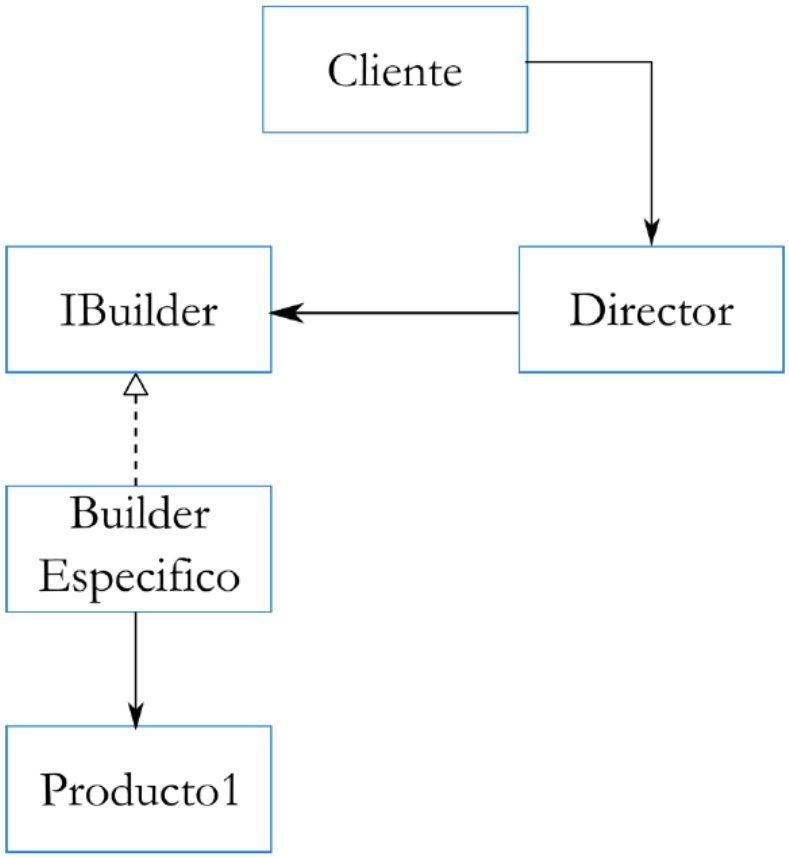
La clase **Producto1**, Es lo que se quiere, es el producto del objeto. El producto terminado depende de BuilderEspecifico, y si lo relacionamos con el ejemplo de la casa, es la construcción una casa de madera o una casa de ladrillos.

La Interfaz IBuilder, permitirá tener la lista de funciones a las características disponibles que los objetos pueden seleccionar e implementar. Es una clase abstracta encargada de construir el producto. Relacionado con el ejemplo de la casa es el obrero general.

La clase BuilderEspecifico permite la selección e implementación de las diferentes características disponibles, estas serán devueltas como un objeto de la clase Producto1 donde se encontrarán las características solicitadas. En la clase Producto1, se mantendrá la lista de características asignadas, así como operaciones para trabajar con ellas. En esta clase se implementan los diferentes productos. Relacionado con el ejemplo de la casa son los obreros que saben hacer la casa.

La clase director se encarga de ofrecer objetos construidos con cantidades diferentes de implementaciones. Ejecuta cada proceso de instanciamiento. Ejecuta el paso a paso la construcción del objeto. Relacionado con el ejemplo de la casa es el arquitecto.

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



Dado el problema de crear un objeto complejo en el cual hay varias características o parámetros que pueden ser inicializadas de diferentes formas. Un Objeto A podrá inicializar solo dos parámetros y los demás vacos. Un objeto B podrá 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.

```cpp
#include "iostream"
#include "vector"
using namespace std;

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";
	}
};

/* La interfaz de Builder especifica métodos para crear las diferentes partes
de los objetos Product. */
class IBuilder {
public:
	virtual ~IBuilder() {}
	virtual void ProducirParteA() const = 0;
	virtual void ProducirParteB() const = 0;
	virtual void ProducirParteC() const = 0;
};

/* Las clases de BuilderEspecifico siguen la interfaz de Builder y proporcionan
implementaciones específicas de los pasos de construcción. El programa puede
tener varias variaciones de Builders, implementadas de manera diferente. */
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;
        }
};

/*El Director solo es responsable de ejecutar los pasos de construcción en una
secuencia particular. Es útil cuando se fabrican productos de acuerdo con un
pedido o configuración específicos. Estrictamente hablando, la clase Director
es opcional, ya que el cliente puede controlar directamente a los
constructores.*/
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();
        }
};

/* El código del cliente crea un objeto constructor, se lo pasa al director y
luego inicia el proceso de construcción. El resultado final se recupera del
objeto constructor. */
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 basico:\n";
        builder->ProducirParteA();
        builder->ProducirParteC();
        p = builder->GetProducto();
        p->ListaComp();
        delete p;
        delete builder;
}
```

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

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 LAB15_GRUPO_A/B/C_CUI_1erNOMBRE_1erAPELLIDO.

   (Ejemplo: LAB15_GRUPO_A _2022123_PEDRO_VASQUEZ).
4. Debe remitir el documento ejecutable con el siguiente formato:

   LAB15_GRUPO_A/B/C_CUI_ EJECUTABLE_1erNOMBRE_1erAPELLIDO

   (Ejemplo: LAB15_GRUPO_A_EJECUTABLE_2022123_PEDRO_VASQUEZ).

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