CEI

Dr. Edye 656 y Rincón, Maldonado

Analista Programador

Programación II

Prof. Gonzalo Duarte

Valentina Benítez

Maldonado, 19 de septiembre de 2023

Resumen

El actual proyecto se desarrolló empleando el lenguaje C# a través del IDE *Visual Studio Community*. En la propuesta se pide representar la construcción de una casa y posibles acciones que se le pueda implementar, para plasmar esto se utilizó el concepto de herencia, polimorfismo y clase abstracta. Mediante métodos se mejoró la funcionalidad de las habitaciones, donde se aplicó luz inteligente en el living y en los dormitorios, alarma de incendio en la cocina, también se renovó el estilo cambiando el color de las paredes y cambiando el tipo de piso, se potenció la estructura del baño con remodelaciones y también se mostró una breve lista de acciones que se pueden realizar en cada ambiente.

Contenido

1. Introducción

1.1 Herencia

La herencia permite crear clases que reutilizan, extienden y modifican el comportamiento definido en otra clase. La clase que se hereda se denomina clase base y la clase que hereda se denomina clase derivada. Cuando se define una clase para que derive de otra, la clase derivada obtiene implícitamente lo creado en la clase base, esto quiere decir que reutiliza el código de la clase base sin tener que volver a implementarlo. La clase derivada puede agregar más clases derivadas lo que amplía la funcionalidad de la clase base. [1]

1.2 Polimorfismo

El polimorfismo se refiere a la posibilidad de definir clases diferentes que tienen métodos o atributos denominados de forma idéntica, pero que se comportan de manera distinta. El concepto de polimorfismo se puede aplicar tanto a funciones como a tipos de datos.

Tipos de polimorfismo

- Polimorfismo de sobrecarga: ocurre cuando las funciones del mismo nombre existen, con funcionalidad similar, en clases que son completamente independientes una de otra.
- Polimorfismo paramétrico: es la capacidad para definir varias funciones utilizando el mismo nombre, pero usando parámetros diferentes (nombre y/o tipo).
- Polimorfismo subtipado: la habilidad para redefinir un método en clases que se hereda de una clase base. [2]

1.3 Clase abstracta

Las clases abstractas son aquellas que no pueden ser instanciadas, y su finalidad es servir como base para otras clases que sí podrán ser instanciadas. En una clase abstracta alguno de sus métodos está declarado pero no está definido, es decir, se especifica su nombre, parámetros y tipo de devolución pero no incluye código. A este tipo de métodos se les conoce como métodos abstractos. Para que una clase sea considerada abstracta al menos uno de sus métodos tiene que ser abstracto. [3]

2. Desarrollo

Se pide realizar la construcción de una casa con herencia, polimorfismo y clase abstracta.
A continuación se agrega la dirección en donde se encuentra el código resultado: https://github.com/ValentiinaBenitez/construccionCasa

2.1 Estructura

Para comenzar con la construcción de una casa primero se definió una clase abstracta padre (base) denominada Estructura que contiene atributos y métodos, los cuales heredan las clases hijas (derivadas) “Living-Comedor”, “Cocina”, “Baño” y “Dormitorio”. Las clases hijas corresponden a los ambientes de la casa y la clase padre como bien se llamó corresponde a la estructura común.
Como se observa en la figura 2.1 lo que se hizo en un principio fue definir los atributos con los que se trabaja y crear un método constructor para instanciar cada clase hija.

```
internal abstract class Estructura
{
    protected int ancho;
    protected int largo;
    protected int cantidadLuces;
    protected string colorPared;
    protected string tipoPiso;
    protected int alto;

    4 referencias
    public Estructura(int ancho, int largo, int cantidadLuces, string colorPared, string tipoPiso, int alto)
    {
        this.ancho = ancho;
        this.largo = largo;
        this.cantidadLuces = cantidadLuces;
        this.colorPared = colorPared;
        this.tipoPiso = tipoPiso;
        this.alto = alto;

    }
```

Figura 2.1: Atributos y método constructor de la clase base.

Para poder trabajar con los atributos se definió el *getter* y *setter* de cada uno de ellos.

```
0 referencias
public string getTipoPiso() => tipoPiso;

1 referencia
public void setAncho(int ancho) => this.ancho = ancho;
```

Figura 2.2: Ejemplo de *getter* y un *setter*.

A fin de ampliar y complejizar la herencia se pensó en dos posibles acciones sobre las habitaciones, representadas en métodos simples de tipo *public void*, en donde se calculó el presupuesto para cambiar el tipo de piso en base al tamaño de la habitación (Figura 2.3) y

el color de la pared (Figura 2.4) en base a las dimensiones. Para calcular la cantidad de tarros por pared se multiplicó por 3 ya que se descontó la puerta y las ventanas.

```
public void cambiarPisos(Estructura estructura)
{
    int metrosCuadradosTotales = estructura.getAncho() * estructura.getLargo();
    int valorBaldosaPorM2 = 359;
    int costoMaterialesExtras = 2500;
    int costoManoDeObra = 4700;
    int costoTotal = (metrosCuadradosTotales * valorBaldosaPorM2) + costoManoDeObra + costoMaterialesExtras;
    Console.WriteLine("Valor por baldosa: $" + valorBaldosaPorM2);
    Console.WriteLine("Costo total de baldosas: $" + (metrosCuadradosTotales * valorBaldosaPorM2));
    Console.WriteLine("Costo de materiales extras: $" + costoMaterialesExtras);
    Console.WriteLine("Costo mano de obra: $" + costoManoDeObra);
    Console.WriteLine("Costo total: $" + costoTotal);
}
```

Figura 2.3: Método que obtiene el costo para cambiar el piso según las dimensiones de la habitación.

```
public void pintarPared(Estructura estructura)
{
    int dimensionPared = estructura.getLargo() * estructura.getAlto();
    int valorTarroPintura = 1200;
    int tarrosPorPared;

    if (dimensionPared <= 9)
    {
        tarrosPorPared = 1;
    }
    else
    {
        tarrosPorPared = 2;
    }

    int cantidadTotalDeTarros = tarrosPorPared * 3;
    int costoTotal = cantidadTotalDeTarros * valorTarroPintura;
    Console.WriteLine("Valor por tarro de pintura: $" + valorTarroPintura);
    Console.WriteLine("Cantidad de tarros por pared: " + tarrosPorPared);
    Console.WriteLine("Cantidad total de tarros: " + cantidadTotalDeTarros);
    Console.WriteLine("Costo total: $" + costoTotal);
}
```

Figura 2.4: Método que obtiene el costo total de la pintura según las dimensiones de las paredes.

Como se explicó, la clase Estructura es abstracta por lo que debe contener al menos un método abstracto (Figura 2.5), para esto se pensó en un método que imprima en pantalla algunas de las acciones que se pueden realizar dentro de cada tipo de habitación.

```
4 referencias
public abstract void acciones();
```

Figura 2.5: Método abstracto.

En última instancia para completar la clase base y en función de implementar el polimorfismo se creó un método *virtual* con una función que agrega luz inteligente a la casa y calcula el costo total con mano de obra y materiales en base a la cantidad de luces que

hay (Figura 2.6), esta se redefinió en algunas clases derivadas dependiendo de su necesidad. Este tipo de polimorfismo se denomina polimorfismo subtipado.

```
public virtual void mejorarEstructura(Estructura estructura)
{
    int cantidadLuces = estructura.getCantidadLuces();
    int valorInterruptor = 689;

    int cantidadInterruptores = cantidadLuces / 3;

    if (cantidadLuces % 3 == 1)
    {
        cantidadInterruptores += 1;
    }

    int valorTotalDeInterruptores = cantidadInterruptores * valorInterruptor;
    int manoDeObra = 3000;
    int costoTotal = valorTotalDeInterruptores + manoDeObra;

    Console.WriteLine("Se añade luz inteligente.");
    Console.WriteLine("- Valor por interruptor: $" + valorInterruptor);
    Console.WriteLine("- Cantidad de interruptores: " + cantidadInterruptores);
    Console.WriteLine("- Costo mano de obra: $" + manoDeObra);
    Console.WriteLine("- Costo total: $" + costoTotal);
}
```

Figura 2.6: Método *virtual* que calcula el presupuesto para implementar luz inteligente.

2.2 Living-Comedor

Se desarrolló la primera clase hija con la definición del método constructor que hereda del padre (Figura 2.7), si bien en la herencia no se vuelve a escribir lo que se hereda en este caso se debe sentenciar la siguiente línea de código para hacer uso del constructor de la clase padre.

```
1 referencia
public LivingComedor(int ancho, int largo, int cantidadLuces, string colorPared, string tipoPiso, int alto) : base(ancho, largo, cantidadLuces, colorPared, tipoPiso, alto) { }
```

Figura 2.7: Línea de código que aparece en todas las clases derivadas que usa el método constructor de la clase base.

Dado que se estableció un método abstracto se tuvo que definir su comportamiento con un *override* (Figura 2.8), a su vez se creó un método propio de la clase que ordena la habitación, en él se aplicó polimorfismo de sobrecarga (Figura 2.9).

```
public override void acciones()
{
    Console.WriteLine("Puedo mirar tele.");
    Console.WriteLine("Puedo comer.");
    Console.WriteLine("Puedo tener reuniones.");
}
```

Figura 2.8: Acciones que se pueden realizar en la clase derivada Living-Comedor.

```
public void ordenar()
{
    Console.WriteLine("Estoy guardando objetos que están fuera de su lugar");
    Console.WriteLine("...");
    Console.WriteLine("Estoy barriendo");
    Console.WriteLine("...");
    Console.WriteLine("Estoy fregando el piso con fabuloso");
}
```

Figura 2.9: Método propio de la clase derivada que representa la acción de ordenar.

2.3 Cocina

En la segunda clase hija se volvió a escribir la línea de código que utiliza el método constructor del padre (Figura 2.7), se definió la clase abstracta con sus respectivas acciones y se estableció el comportamiento del método propio “ordenar" (Figura 2.10).

```
public override void acciones()
{
    Console.WriteLine("Puedo cocinar.");
    Console.WriteLine("Puedo comer.");
    Console.WriteLine("Puedo fregar.");
}

1 referencia
public void ordenar()
{
    Console.WriteLine("Estoy ordenando la alacena");
    Console.WriteLine("...");
    Console.WriteLine("Estoy limpiando la heladera");
    Console.WriteLine("...");
    Console.WriteLine("Estoy desengrasando la cocina y el horno");
}
```

Figura 2.10: Asignación de comportamiento al método abstracto y método que ordena la habitación.

Luego a través del método *override* se utilizó el método *virtual* “mejorarEstructura” de la clase base y se le asignó otro comportamiento adecuado al ambiente, se cambió la adición de luz inteligente por la adición de detector de humo (Figura 2.11), se calculó en base al tamaño de la cocina cuántos detectores de humo se precisan y el costo final con mano de obra y materiales. Este es el ejemplo de polimorfismo subtipado.

```
public override void mejorarEstructura(Estructura estructura)
{
    int valorDetectorDeHumo = 454;
    int largoCocina = estructura.getLargo();
    int anchoCocina = estructura.getAncho();
    int costoManoDeObra = 2900;
    int cantidadDetectorDeHumo;

    if (anchoCocina <= 3)
    {
        cantidadDetectorDeHumo = 1;
    }
    else if (anchoCocina > 3 && anchoCocina <= 6)
    {
        cantidadDetectorDeHumo = 2;
    }
    else  { cantidadDetectorDeHumo = 3; }

    if (largoCocina <= 4)
    { }
    else if (largoCocina > 4 && largoCocina <= 8)
    {
        cantidadDetectorDeHumo = cantidadDetectorDeHumo * 2;
    }
    else { cantidadDetectorDeHumo = cantidadDetectorDeHumo * 3; }

    int costoTotal = ((cantidadDetectorDeHumo * valorDetectorDeHumo) + costoManoDeObra);

    Console.WriteLine("Se agrega detector de humo.");
    Console.WriteLine("- Valor por detector de humo: $" + valorDetectorDeHumo);
    Console.WriteLine("- Cantidad necesaria de detectores: " + cantidadDetectorDeHumo);
    Console.WriteLine("- Costo total de detectores: $" + (cantidadDetectorDeHumo * valorDetectorDeHumo));
    Console.WriteLine("- Costo mano de obra: $" + costoManoDeObra);
    Console.WriteLine("- Costo total: $" + costoTotal);
```

Figura 2.11: Método *override* que calcula cuántos detectores de humo se precisan y el costo para colocarlos.

2.4 Baño

Dentro de la clase baño también se definió la línea de código que utiliza el método constructor del padre y se le asignó el comportamiento a los métodos “acciones” y “ordenar” (Figura 2.12).

```
public override void acciones()
{
    Console.WriteLine("Puedo bañarme.");
    Console.WriteLine("Puedo cepillarme los dientes.");
    Console.WriteLine("Puedo hacer mis necesidades biológicas.");
}

1 referencia
public void ordenar()
{
    Console.WriteLine("Estoy limpiando el lavamanos y el estante con limpiamueble");
    Console.WriteLine("...");
    Console.WriteLine("Estoy limpiando el espejo con limpiavidrio");
    Console.WriteLine("...");
    Console.WriteLine("Estoy pasando la fregona con fabuloso");
```

Figura 2.12: Método abstracto “acciones” y método con polimorfismo de sobrecarga “ordenar”.

Con el fin de aplicar nuevamente el polimorfismo subtipado se redefinió el método “mejorarEstructura” en donde en base al tamaño del baño se le asignó el costo para aplicar mayor presión de agua, agua caliente en el lavamanos y remodelamiento de tuberías (Figura 2.13).

```
public override void mejorarEstructura(Estructura estructura)
{
    int metrosCuadrados = estructura.getAncho() * estructura.getLargo();
    int costoMejora;

    if (metrosCuadrados <= 9)
    {
        costoMejora = 7000;
    }
    else
    {
        costoMejora = 10000;
    }

    Console.WriteLine("Cambios:");
    Console.WriteLine("- Mayor presión de agua.");
    Console.WriteLine("- Agua caliente en el lavamanos.");
    Console.WriteLine("- Remodelamiento de tuberías.");
    Console.WriteLine("El costo es de $" + costoMejora);
}
```

Figura 2.13: Cálculo de presupuesto para mejorar el baño.

2.5 Dormitorio

Por último en la clase “Dormitorio” solamente se realizó lo mismo que se explicó anteriormente con los métodos: constructor, “acciones” y “ordenar” (Figura 2.14).

```
public override void acciones()
{
    Console.WriteLine("Puedo dormir.");
    Console.WriteLine("Puedo estudiar.");
    Console.WriteLine("Puedo vestirme.");
    Console.WriteLine("Puedo leer un libro.");
}

1 referencia
public void ordenar()
{
    Console.WriteLine("Estoy tendiendo la cama");
    Console.WriteLine("...");
    Console.WriteLine("Estoy limpiando la mesa de luz");
    Console.WriteLine("...");
    Console.WriteLine("Estoy pasando la aspiradora");

}
```

Figura 2.14: Método “acciones” que imprime lo que se puede hacer y método “ordenar” que representa dicha acción.

2.6 Construcción y remodelamiento de la casa

Con el uso de las clases, tanto base como derivadas, junto a sus métodos se diseñó en la clase “Program” un código, donde se desarrolló la construcción y el remodelamiento de una casa.

Una vez se inicia el código se observa un menú con la bienvenida y tres opciones: construir, remodelar o salir (Figura 2.15). En caso de querer salir se ingresa por teclado el 3.

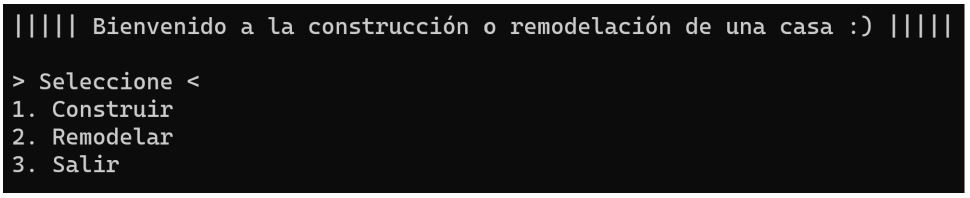

Figura 2.15: Menú principal.

Si se selecciona la opción 1 (construir) a continuación se comienza a pedir valores para los atributos ancho, largo, cantidad de luces, color de pared, tipo de piso y altura de cada habitación como se puede ver en el ejemplo de la Figura 2.16.

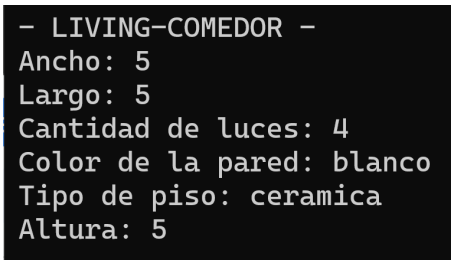

Figura 2.16: Asignación de valores para la construcción del living-comedor.

Enseguida que se eligió los valores se muestra un menú con dos opciones: “ordenar habitaciones” y “ver que puedo hacer en las habitaciones” (Figura 2.17), si se selecciona la primera se observa como se ordena y en caso de elegir la segunda se indica algunas de las actividades que se realiza por habitación (Figura 2.18). En esta porción de código se hizo uso del método abstracto “acciones” y el método “ordenar” al que se le aplicó polimorfismo de sobrecarga.

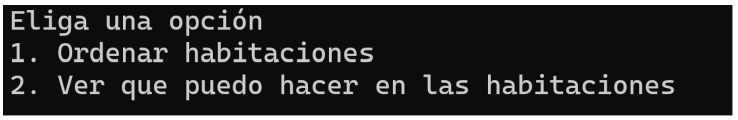

Figura 2.17: Segundo menú.

Primera opción

```
-- LIVING-COMEDOR --
Estoy guardando objetos que están fuera de su lugar
...
Estoy barriendo
...
Estoy fregando el piso con fabuloso
```

Segunda opción

```
-- LIVING-COMEDOR --
Puedo mirar tele.
Puedo comer.
Puedo tener reuniones.
```

Figura 2.18: Ejemplos de las opciones de la figura 2.17.

Luego de que se eligió por primera vez lo que se quiso hacer se muestra el mismo menú con la adición de la opción “siguiente” para avanzar. Una vez se decide seguir con el programa se presenta otro menú en donde se elige una habitación para mejorar (Figura 2.19), dependiendo de la opción que se elija se puede observar la mejora con detalles y su costo (Figura 2.20). En este sector de código se utilizó el método “mejorarEstructura” con el cual se aplicó polimorfismo subtipado.

```
*** Implemente una mejora ***
1. Living-Comedor
2. Cocina
3. Baño
4. Dormitorio
```

Figura 2.19: Tercer menú.

```
Se añade luz inteligente.
- Valor por interruptor: $689
- Cantidad de interruptores: 2
- Costo mano de obra: $3000
- Costo total: $4378
```

Figura 2.20: Ejemplo de mejorar una habitación.

A continuación se muestra nuevamente el menú por si se quiere realizar otra mejora y además se le agrega la opción “siguiente”. En caso de avanzar se devuelve al menú principal con las opciones “construir”, “remodelar” y “salir”. Si se selecciona “remodelar” (opción 2) se puede observar un texto con el tipo de piso y el color de las paredes de las habitaciones, seguido a esto un menú con las modificaciones “cambiar pisos” y “pintar pared” (Figura 2.21). Si no se construyó una casa anteriormente, los valores que se muestran son de la casa por defecto como expone la Figura 2.21.

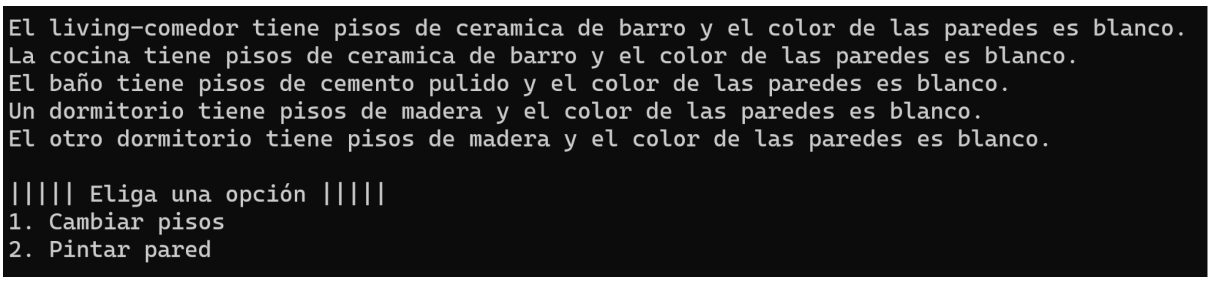

Figura 2.21: Menú de la opción “remodelar”.

Si se elige cambiar pisos (opción 1) se pide un tipo de piso por habitación y a su vez se presenta el costo total con los detalles (Figura 2.22). En esta porción de código se hizo uso del método *void* que aplica herencia. Luego de que se cambia el piso en todas las habitaciones se vuelve al menú principal.

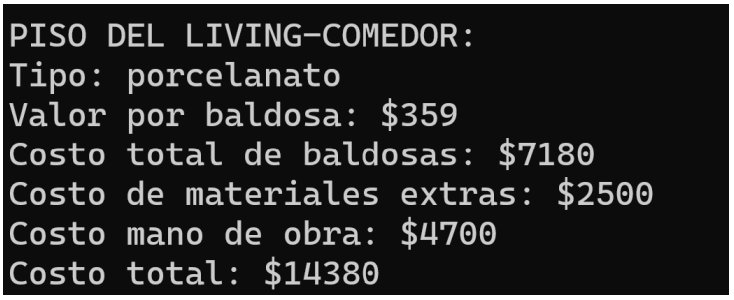

Figura 2.22: Ejemplo de costo total y detalles para cambiar el piso de una habitación.

Y si se elige pintar paredes (opción 2) se pide el color por habitación y a su vez se detalla el costo total (Figura 2.23). También se utilizó el método *void* donde se aplicó herencia.

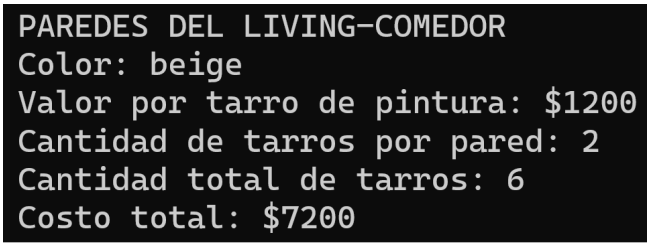

Figura 2.23: Ejemplo del costo total y detalles para pintar las paredes de una habitación.

Por último, después de que se elige el color de las paredes de todas las habitaciones se regresa al menú principal.

3. Conclusiones

En el cumplimiento de realizar la construcción de una casa, se piensa que se tuvo un buen manejo de los conceptos de herencia, polimorfismo y clase abstracta. Se aplicó herencia desde la clase padre a las clases hijas con los atributos, el método constructor, los métodos comunes de pintar las paredes y cambiar los pisos y los métodos *getter* y *setter* de cada atributo. Se implementó polimorfismo de sobrecarga y subtipado, el primero se observa en todas las clases hijas mediante el método “ordenar” y el segundo se incorpora en el método “mejorarEstructura” cuando se sobreescribe en las clases “Cocina” y “Baño”. Se añadió clase abstracta en la clase base por lo que se tuvo que definir un método abstracto “acciones” y crear su comportamiento en cada clase derivada. Un aspecto que se puede mejorar podría ser agregar la posibilidad de crear un dormitorio o baño adicional con una opción en el menú “ampliar construcción”.

4. Referencias

[1] - https://learn.microsoft.com/es-es/dotnet/csharp/fundamentals/object-oriented/inheritance

[2] - https://www.ecured.cu/Polimorfismo_(Inform%C3%A1tica)

[3] - https://www.glosarioit.com/Clase_abstracta