ANALISTA PROGRAMADOR

Programación II

Profesor Gonzalo Duarte

Práctico N° 1

Programación orientada a objetos
Clases - Herencia - Polimorfismo

Realizado por:
Valentina Benitez y Karen Fernández
Setiembre 2023

Resumen

El actual proyecto se desarrolló empleando el lenguaje C# a través del IDE *Visual Studio Community*. La propuesta busca integrar en código el Parlamento, quiénes lo integran y algunas de las actividades que se realizan allí. Para llevarlo a cabo se utilizó el paradigma de programación orientada a objetos y, por lo tanto, el concepto de herencia, polimorfismo y clase abstracta.

Se crearon las siguientes clases: Legislador como padre, Senador y Diputado como hijas y Parlamento independiente a estas, aunque relacionada. En Legislador se definieron los atributos junto a sus *getter* y *setter*, el método constructor, dos métodos *virtual,* uno para presentar una propuesta y otro para participar en un debate y un método abstracto para votar. En Senador y Diputado se definió como atributo el número de asiento de la cámara correspondiente (alta o baja), su *getter* y *setter*, una lista con propuestas y un método que la retorna, el funcionamiento del método abstracto y particularmente en Diputado se modificó el comportamiento de los métodos *virtual* heredados. Por último, en Parlamento, se utilizó una lista como atributo para almacenar Legisladores, se le creó su método constructor y un método que devuelve la información contenida, también se desarrolló un método para agregar Legisladores, otro para eliminar, para listar cámaras y para filtrar la cantidad por tipo (Senador o Diputado).

Para relacionar estas clases y sus métodos en un programa congruente y funcional, se realizó un menú con varias opciones simulando un sistema de gestión parlamentaria que recibe y procesa información por medio de la interacción con el usuario.

Contenido

1. Introducción..... 1

1.1 Programación Orientada a Objetos..... 1

1.2 Clases y objetos..... 1

1.3 Herencia..... 1

1.4 Polimorfismo..... 1

1.5 Clase abstracta..... 2

2. Desarrollo..... 3

2.1 Legislador..... 3

2.2 Senador..... 5

2.3 Diputado..... 6

2.4 Parlamento..... 7

2.5 Integración..... 10

3. Resultados..... 13

4. Conclusiones..... 13

5. Referencias..... 14

1. Introducción

1.1 Programación Orientada a Objetos

Existen varios paradigmas de programación, que buscan solucionar los problemas mediante diferentes métodos, uno de ellos es Programación Orientada a Objetos. El objetivo es trasladar al programa los objetos de la vida real. Se organiza el código en unidades denominadas clases, de las cuales se crean objetos que se relacionan entre sí para conseguir los objetivos de las aplicaciones [1].

1.2 Clases y objetos

La clase contiene la definición de las características de un modelo, a las que llamamos atributos o propiedades, junto con la implantación de sus funcionalidades particulares, llamadas métodos. Se puede definir como la abstracción de un objeto. A partir de una clase podemos crear cualquier número de objetos de esa clase determinando así sus características. Podemos pensarlo como si la clase fuera un molde o plantilla y a partir de ese molde creemos distintos ejemplares, aunque no tienen por qué ser todos iguales sino que pueden variar sus atributos. La acción de crear un objeto a partir de una clase se llama instanciar [1].

1.3 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. [2]

1.4 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. [3]

1.5 Clases abstractas

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. [4]

2. Desarrollo

Para comenzar se describe punto por punto lo que se solicitó realizar:

- **Punto uno:** se pide crear una clase llamada Legislador con los atributos: partido político, departamento que representa, número de despacho, nombre, apellido, edad, si es casado y la cámara a la que pertenece. También se debe agregar el método “GetCámara” el cual devuelve el valor del atributo cámara.
- **Punto dos:** se pide crear las clases Senador y Diputado que hereden de Legislador y que contengan los atributos número de asiento de cámara alta y número de asiento de cámara baja respectivamente.
- **Punto tres:** se pide crear una clase llamada Parlamento que contenga como propiedad una lista de Legisladores. Se debe definir un constructor que comience la lista y agregar los siguientes métodos: “RegistrarLegislador” en donde se pueda agregar o eliminar un Legislador, “ListarCámaras” que imprima en la consola el número de despacho, nombre, apellido y la cámara en la que trabajan y “CantidadDeLegisladoresPorTipo” en donde se liste la cantidad de Senadores y Diputados.
- **Punto cuatro:** se solicita agregar los siguientes métodos polimórficos: “PresentarPropuestaLegislativa” que permita a un legislador exponer una propuesta en la cámara correspondiente. “Votar” para emitir un voto en una sesión y “ParticiparDebate” en donde se permita a un legislador participar en un debate.
- **Punto cinco:** se pide crear un menú para administrar las posibles operaciones que se consideren convenientes.
- **Punto seis:** se pide realizar un diagrama UML.

A continuación se agrega la dirección en donde se encuentra el código resultado: https://github.com/KarenFernandez13/poo-practico1.git - (Carpeta Practico1 - copia).

2.1 Legislador

Se inició el proyecto con la creación de la clase padre Legislador de tipo *abstract*.
Se definieron los siguientes atributos: *string* PartidoPolitico, *string* Departamento, *int* Despacho, *string* Nombre, *string* Apellido, *int* Edad, *string* Cámara y *bool* Casado (Se inicializa en valor false. Ver funcionamiento en 2.5 - Integración) todos ellos junto a sus correspondientes *getter y setter* (Figura 2.2). También se creó el método constructor Legislador, el cual recibe como parámetros a todos los atributos antes mencionados, ya que serán ingresados por el usuario del sistema (Figura 2.1).

```
abstract class Legislador
{
    protected string partidoPolitico;
    protected string departamento;
    protected int despacho;
    protected string nombre;
    protected string apellido;
    protected int edad;
    protected bool casado;
    protected string camara;

    public Legislador(string partidoPolitico, string departamento, int despacho, string nombre, string apellido,
        int edad, bool casado, string camara)
    {
        this.camara = camara;
        this.casado = casado;
        this.edad = edad;
        this.nombre = nombre;
        this.apellido = apellido;
        this.despacho = despacho;
        this.departamento = departamento;
        this.partidoPolitico = partidoPolitico;

    }
```

Figura 2.1: Clase Legislador con sus atributos y su método constructor.

```
public string GetCamara() => camara;
public bool GetCasado() => casado;
public int GetEdad() => edad;
public string GetNombre() => nombre;
public string GetApellido() => apellido;
public int GetDespacho() => despacho;
public string GetDepartamento() => departamento;
public string GetPartidoPolitico() => partidoPolitico;


public void SetCamara(string camara) => this.camara = camara;
public void SetCasado(bool casado) => this.casado = casado;
public void SetEdad (int edad) => this.edad = edad;
public void SetNombre(string nombre) => this.nombre = nombre;
public void SetApellido(string apellido) => this.apellido = apellido;
public void SetDespacho(int despacho) => this.despacho = despacho;
public void SetDepartamento(string departamento) => this.departamento = departamento;
public void SetPartido(string partido) => partido = partidoPolitico;
```

Figura 2.2: Getters y Setters correspondientes

Debido a que la clase se declaró abstracta, se definió un método del mismo tipo para la función “Votar”, que se utiliza cuando se ingresan votos de los legisladores. Éste será llamado desde el menú principal, pidiendo al usuario número de despacho, el cual será tomado como referencia y utilizado para retornar nombre y apellido del legislador asociado, concatenado con un mensaje acorde (*string).*

Con el fin de incorporar otros tipos de polimorfismo, se crearon los métodos *virtual* (Polimorfismo subtipado) “ParticiparDebate” y “PresentarPropuesta” que, al ser llamados

desde el menú, piden al usuario número de despacho como referencia y retorna nombre y apellido del legislador asociado, concatenado a un *string* que informa que dicho legislador participará en el próximo debate o que presentará una propuesta. Éstos métodos serán modificados mediante *override* en sus clases hijas, adaptándose al tipo de legislador que corresponda (Figura 2.3).

```
public abstract string Votar();


public virtual string ParticiparDebate()
{
    return "Senador " + nombre + " " + apellido + " participa en el próximo debate";
}

public virtual string PresentarPropuesta()
{
    return "Senador " + nombre + " " + apellido + " presenta una propuesta";
}
```

Figura 2.3: Método abstracto y métodos que aplican polimorfismo subtipado.

2.2 Senador

Luego se creó la clase hija Senador que hereda los atributos y métodos de clase “Legislador”, se añadió un atributo propio denominado “numAsientoCamaraAlta” y se definió su *getter y setter*.

Para poder hacer uso del constructor Legislador se agregó el método *base* constructor que envía los datos de su clase padre para crear el objeto con todos sus atributos, también se ingresa por parámetro el atributo propio y, todo esto, conforma el constructor Senador (Figura 2.4).

```
class Senador : Legislador
{
    private int NumAsientoAlta;

    2 referencias
    public Senador(string partidoPolitico, string departamento, int despacho, string nombre, string apellido,
        int edad, bool casado, string camara, int NumAsientoAlta) : base(partidoPolitico, departamento,
        despacho, nombre, apellido, edad, casado, camara)
    {
        this.NumAsientoAlta = NumAsientoAlta;
    }

    0 referencias
    public int GetAsientoAlta() { return NumAsientoAlta;}

    1 referencia
    public void SetAsientoAlta(int asientoAlta) { asientoAlta = NumAsientoAlta; }
```

Figura 2.4: Clase Senador con su atributo particular, su *getter, setter* y método constructor.

El método abstracto “Votar” de la clase padre, aquí se sobreescribe realizando una pequeña modificación (*string* “Senador”) en el mensaje que devuelve, mientras que el método “ParticiparDebate” se hereda de la clase Legislador sin modificaciones en este caso (Figura 2.5).

```
public override string Votar()
{
  return "Senador" +nombre + " " + apellido + " votó";
}
```

Figura 2.5: Método abstracto “Votar” sobreescrito en clase Senador.

Se creó una lista de tipo *string* llamada “PropuestasSenadores” que contiene diferentes propuestas precargadas, las cuales luego podrán ser consultadas en el menú mediante el método “ObtenerPropuestas”, creado polimórfico de sobrecarga para ser aplicado también en otras clases (Figura 2.6).

```
List<string> propuestasSenadores = new List<string>
{
    "P. N°1: Animales", "P. N°2: Medio ambiente", "P. N°3: Agua"
};

public List<string> obtenerPropuestas()
{
    return propuestasSenadores;
}
```

Figura 2.6: Lista de propuestas y método para obtener la información que contiene.

2.3 Diputado

En la creación de la clase hija Diputado se definió el atributo particular "numAsientoCamaraBaja" con su *getter y setter y* el método *base* constructor, recibiendo como parámetros a todos los atributos heredados y al atributo particular de la clase (Figura 2.7).

Los métodos definidos en esta clase son: el abstracto "Votar" con una modificación en el *string* que retorna, "ParticiparDebate" al cual, en este caso, también se le realiza una modificación en el resultado que devuelve (Figura 2.8).

Al igual que en la clase Senador antes descrita, se creó una lista de propuestas legislativas y un método para obtenerlas y así poder mostrarlas en pantalla al usuario que las solicite (Figura 2.9).

```
class Diputado : Legislador
{
    private int NumAsientoBaja;
    2 referencias
    public Diputado(string partidoPolitico, string departamento, int despacho, string nombre, string apellido,
        int edad, bool casado, string camara, int NumAsientoBaja) : base(partidoPolitico, departamento,
        despacho, nombre, apellido, edad, casado, camara)
    {
        this.NumAsientoBaja = NumAsientoBaja;
    }

    0 referencias
    public int GetAsientoBaja() { return  NumAsientoBaja;}
    1 referencia
    public void SetAsientoBaja(int asientoBaja) { this.NumAsientoBaja = asientoBaja; }
```

Figura 2.7: Método para crear objetos de tipo Diputado, atributo propio, *getter y setter*.

```
public override string Votar()
{
    return nombre + " " + apellido + " votó";

}

public override string ParticiparDebate()
{
    return "Diputado " + nombre + " " + apellido + " participa en el próximo debate";
}

public override string PresentarPropuesta()
{
    return "Diputado " + nombre + " " + apellido + " presenta una propuesta";
}
```

Figura 2.8: Métodos sobreescritos: virtual "ParticiparDebate" y "PresentarPropuesta". Abstracto "Votar".

```
List<string> propuestasDiputados = new List<string>
{
    "P. N°1: Carreteras", "P. N°2: Estructuras abandonadas", "P. N°3: Iluminación"
};

public List<string> obtenerPropuestas()
{
    return propuestasDiputados;
}
```

Figura 2.9: Lista de propuestas de diputados y método de obtención de las mismas.

2.4 Parlamento

La clase Parlamento funciona independiente de las demás, no recibe datos heredados. Contiene una lista de tipo Legislador, la cual se compone de los diferentes legisladores que el usuario ingrese y se inicia con su propio método constructor. Se incluyó un método "ObtenerLegisladores", de tipo *List*, que permite utilizar la información de la lista en otras clases (Figura 2.10).

```
class Parlamento
{
    List <Legislador> ListaLegisladores;

    public Parlamento()
    {
        ListaLegisladores = new List<Legislador>();
    }


    public List<Legislador> obtenerLegisladores()
    {
        return ListaLegisladores;
    }
```

Figura 2.10: Lista de legisladores, su constructor y método para obtener la información contenida.

Se creó el método "IngresarLegislador" que permite a un legislador (previamente creado con su constructor correspondiente) ser enviado por parámetro, y así colocarlo en la lista especificada, en este caso en ListaLegisladores mediante el método *List<T>.Add, perteneciente a la clase System.Collections.Generic* propia de C#.

De la misma clase se utilizó el método *List<T>.Remove,* dentro del método creado "EliminarLegislador", éste recibe por parámetro el número de despacho del legislador a quitar, lo busca dentro de la lista (*List<T>.Find* , método de la clase) y lo elimina.

El método “ListarCamaras” se utiliza para recorrer la lista y devolver la información que allí se encuentra, logrado mediante un bucle *foreach*, el cual genera una acción para cada elemento allí encontrado. En este caso se extraen datos con métodos *Getter*, como nombre, apellido, cámara a la que pertenece y número de despacho y se concatenan con *strings* para generar un listado coherente para mostrar en pantalla (Figura 2.11).

```
public void IngresarLegislador(Legislador Legislador)
{
    ListaLegisladores.Add(Legislador);
}


public void EliminarLegislador(int numero)
{
    Legislador eliminarLegislador = ListaLegisladores.Find(x => x.GetDespacho() == numero);
    ListaLegisladores.Remove(eliminarLegislador);
}


public void ListarCamaras()
{
    foreach (var x in ListaLegisladores)
    {
        Console.WriteLine("Legislador: " + x.GetNombre() + " " + x.GetApellido() + " - " +"Cámara: " +
         x.GetCamara() + " - " + "Numero de despacho: " + x.GetDespacho());
    }
}
```

Figura 2.11: Métodos para ingresar y eliminar legisladores a la lista y método para generar el listado de dicha lista completa.

Para poder contabilizar los legisladores según su tipo o cámara, se inició una variable de tipo *int* llamada contador y luego un método “CantidadPorTipo” que recibe un parámetro *int,* éste parámetro viene dado por la elección del usuario durante la ejecución. Deberá ingresar “0” para contar Diputados ó “1” para contar Senadores. En cualquiera de los dos casos se aplica un bucle foreach que recorre ListaLegisladores, busca la cámara ingresada en cada elemento y dada la coincidencia, suma 1 al contador. Así con todos los elementos de la lista. Al final retorna el valor de la variable contador, el cual será mostrado en pantalla a modo de resultado (Figura 2.12).

```
public int CantidadPorTipo(int numero)
{
    int contador = 0;

    if (numero == 0)
    {
        foreach (var y in ListaLegisladores)
        {
            if ((y.GetCamara() == "Diputado"))
            {
                contador++;
            }
        }
    }
    else if (numero == 1)
    {
        foreach (var y in ListaLegisladores)
        {
            if ((y.GetCamara() == "Senador"))
            {
                contador++;
            }
        }
    }
    return contador;
}
```

Figura 2.12: Métodos para contabilizar Senadores o Diputados según elección.

2.5 Integración

Para integrar todos los elementos y crear una aplicación funcional, se creó un Menú que presenta las opciones y permite seleccionar al usuario la que desea implementar. Para inicializar los objetos necesarios para el funcionamiento, se declara un objeto de cada clase. Un “Parlamento” denominado “Legisladores” de tipo la lista, un Diputado1 y un Senador1. Con estos 2 últimos se podrán ingresar todos los legisladores que sean necesarios, ya que cuando se crea un legislador, sus datos se guardan en la lista y ese mismo objeto se reutiliza para generar los siguientes y repetir el proceso. También al inicio se declaran todas las listas a utilizar y las variables necesarias para completar los atributos. Menú: Mediante la variable “seleccion” de tipo *int*, se ingresa a las diferentes opciones, son 6 en total (incluida la opción de salida) (Figura 2.13).

```
//MENU
Console.WriteLine("SISTEMA DE INFORMACIÓN PARLAMENTARIA");
Console.WriteLine("");
Console.WriteLine("Menú Principal: ");
Console.WriteLine("");
Console.WriteLine("Elige una opción: ");
Console.WriteLine("1 - Ingresar / Eliminar Legislador ");
Console.WriteLine("2 - Ver Listados de Cámaras");
Console.WriteLine("3 - Propuestas Legislativas ");
Console.WriteLine("4 - Registrar participación en debate");
Console.WriteLine("5 - Registrar Voto");
Console.WriteLine("0 - Salir");
```

Figura 2.13: Menú principal.

Comienza dentro de un bucle *While* con la condición: selección distinto de “0”, por lo que si se ingresa el valor 0, se finaliza el programa.

La opción 1 (if (seleccion == 1)) permite ingresar o eliminar legisladores, se pregunta si es senador o diputado, cada opción dentro de su *if* correspondiente. En ambos casos, se pedirán los datos correspondientes a los atributos de los legisladores que serán cargados a variables antes declaradas, se realizan validaciones de tipo de dato y de rangos de valores, según necesidad particular. Éstos datos son los que se envían por parámetro al método constructor que corresponde para el objeto que se está creando.

En el caso específico del atributo “Casado” (*bool*), se incluyó de por medio una variable “EstadoCivil” de tipo *int* para que el usuario ingrese 1 o 0, *true* o *false* respectivamente (Figura 2.14).

```
int EstadoCivil = Convert.ToInt32(Console.ReadLine());
if (EstadoCivil == 1)
{ casado = true; }
else { casado = false; }
```

Figura 2.14: Condición para agregar *true* o *false* en el atributo casado.

A continuación se utiliza el método IngresarLegislador para cargar el objeto en la lista Legisladores. Se da un mensaje en pantalla de ingreso exitoso y se vuelve a mostrar el menú principal. Cuando se elige la opción “Eliminar Legislador”, se debe ingresar el número de despacho para identificar al legislador a remover, se muestra la lista de legisladores disponible, se ingresa dicho número, se valida que exista en la lista y que sea efectivamente un número y se aplica el método “EliminarLegislador”. Luego de realizado, se vuelve a mostrar el menú principal.

En la opción 2 (if (seleccion == 2)), se presentan dos posibles acciones, mostrar el listado de legisladores existentes (opción 1) o mostrar la cantidad de cada tipo (opción 2). En ambos casos, se implementan los métodos creados para cada acción referenciando a la lista de legisladores y se muestra un correspondiente mensaje informativo en pantalla, logrados por medio de *Console.Writeline*, concatenando un *string* con el resultado del método aplicado. En el caso de mostrar la cantidad por tipo, se aplica dos veces el método asociado, una para diputados y otra para senadores y se proveen ambos resultados a la vez como se observa en la Figura 2.15, y en el caso para listar los legisladores el resultado se imprime en el método como se explicó en el punto 2.4 Parlamento.

```
Console.WriteLine("Cantidad de Diputados: " + Legisladores.CantidadPorTipo(0));
Console.WriteLine("Cantidad de Senadores: " + Legisladores.CantidadPorTipo(1));
```

Figura 2.15: Reclutamiento y exposición de la cantidad de legisladores por tipo.

En la opción 3 (if (seleccion == 3)), se encuentra el listado de propuestas existentes, de senadores y diputados por separado, más el método “IngresarPropuesta” (opción 1 y 2 respectivamente). Para mostrar propuestas, se recorre las listas “PropuestasSenadores” y “PropuestasDiputados” con un bucle *foreach* y se muestra cada una de las disponibles en pantalla, separadas por su cualidad de pertenecer a la cámara de diputados, o bien de senadores. Éstas listas contienen información previamente cargada por el programador.

Para el ingreso de una nueva propuesta por parte del usuario del sistema, se muestra el listado de legisladores y se solicita el número de despacho, de nuevo se recorre la lista Legisladores con *foreach* y mediante el método “GetDespacho” igualado al valor ingresado se debe encontrar el legislador asociado y atribuirle la presentación de la nueva propuesta. Se valida existencia y tipo de dato ingresado.

De esta misma manera se implementan los métodos “ParticiparDebate” y “Votar”, opción 4 y 5 del menú respectivamente. Éstos no ameritan listados de información, ya que solo apuntan a una función particular que no depende de otros datos.

Todos ellos al finalizar la ejecución emiten un mensaje relativo a la acción que demuestra haber sido realizada con éxito.

3. Resultados

Como resultado de la creación del Sistema de Información Parlamentaria se obtuvo un diagrama UML que describe de forma concisa las clases, las relaciones entre ellas, sus atributos y sus métodos (Figura 3.1)

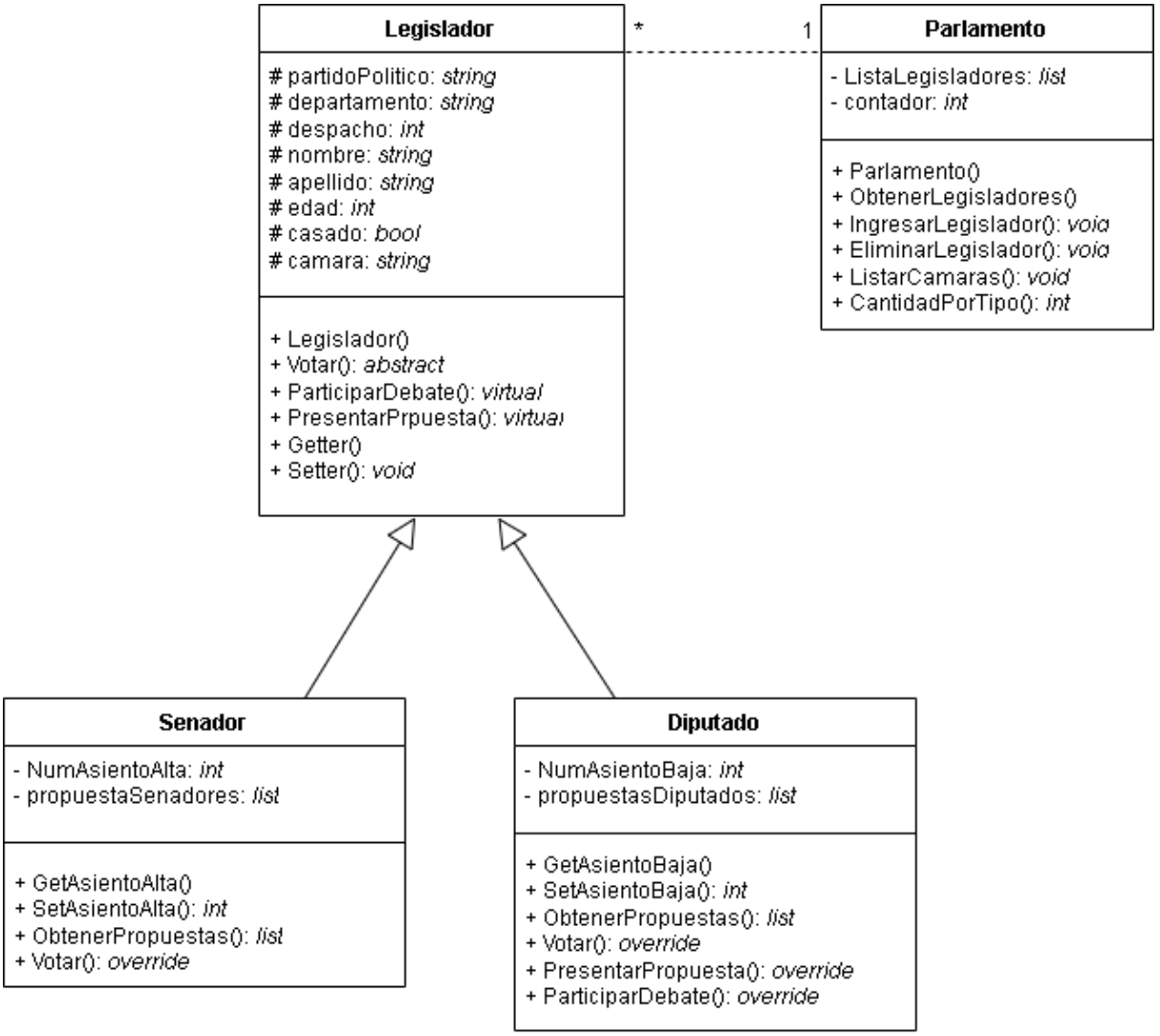

Figura 3.1: Diagrama UML.

A continuación se expone un resumen del funcionamiento y uso visto desde la aplicación de consola:

- Procedimiento para ingreso y eliminación de legisladores (Figura 3.2).

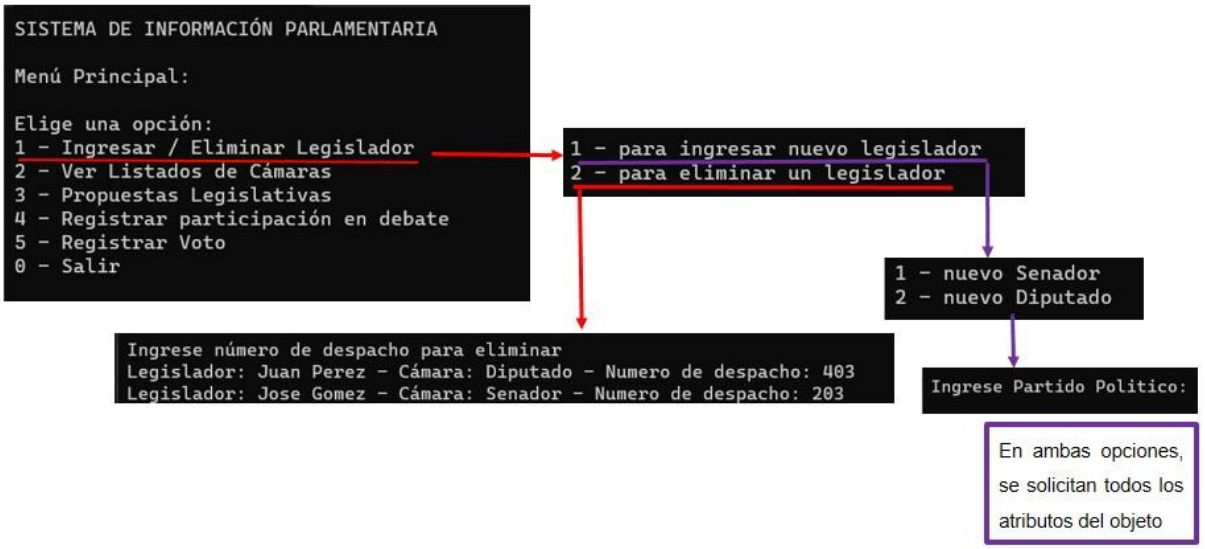

Figura 3.2

- Obtención de listados de cámaras (Figura 3.3).

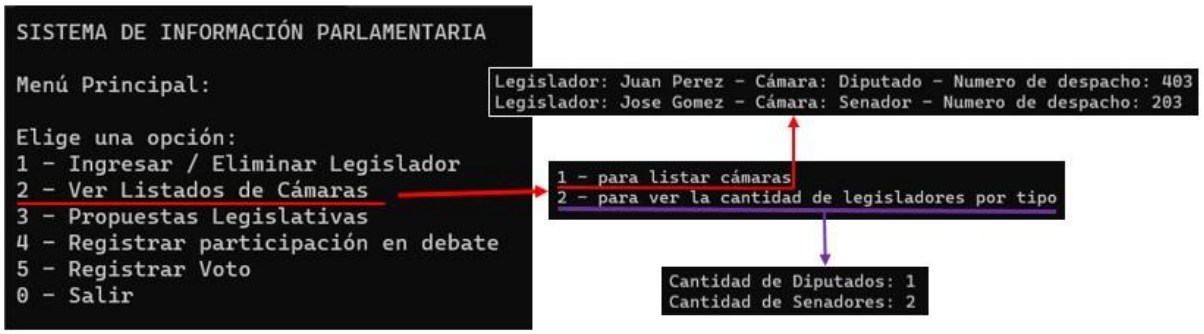

Figura 3.3

- Acceso y registro de propuestas legislativas (Figura 3.4)

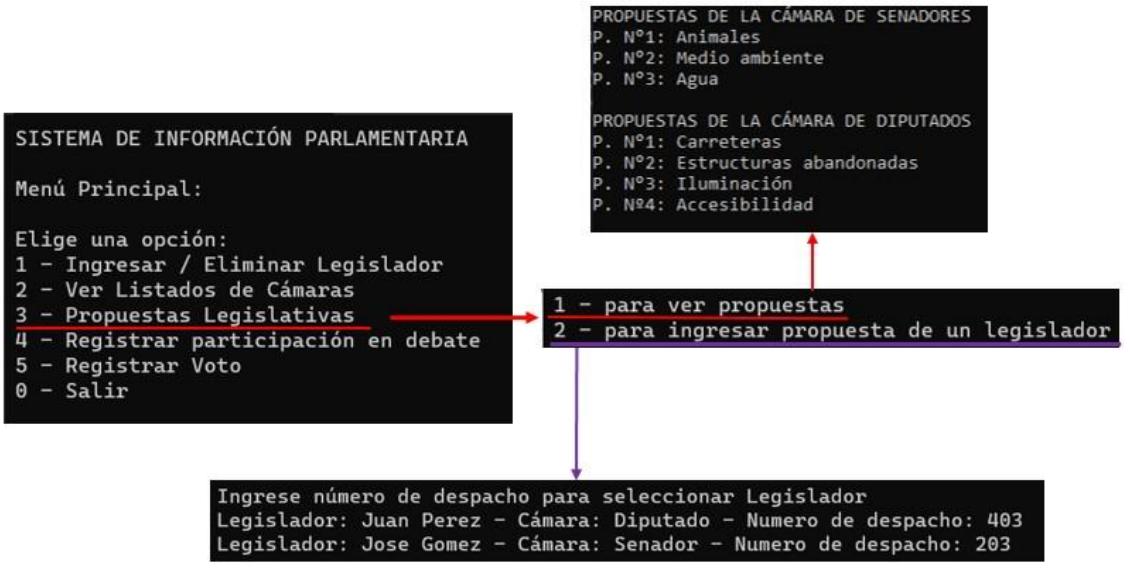

Figura 3.4

- Registro de participación en debates (Figura 3.5).

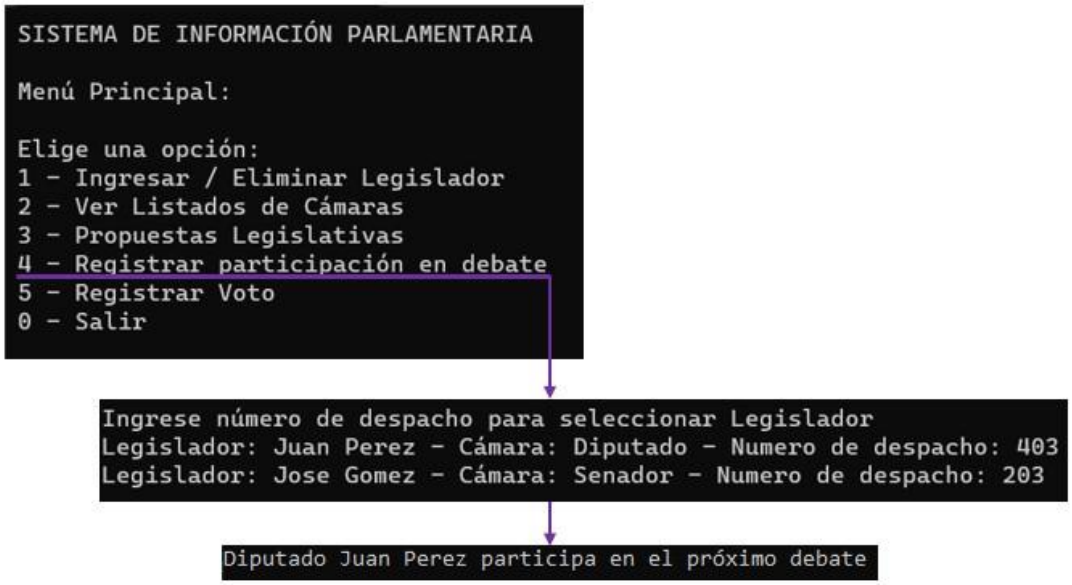

Figura 3.5

- Ingreso de votos (Figura 3.6).

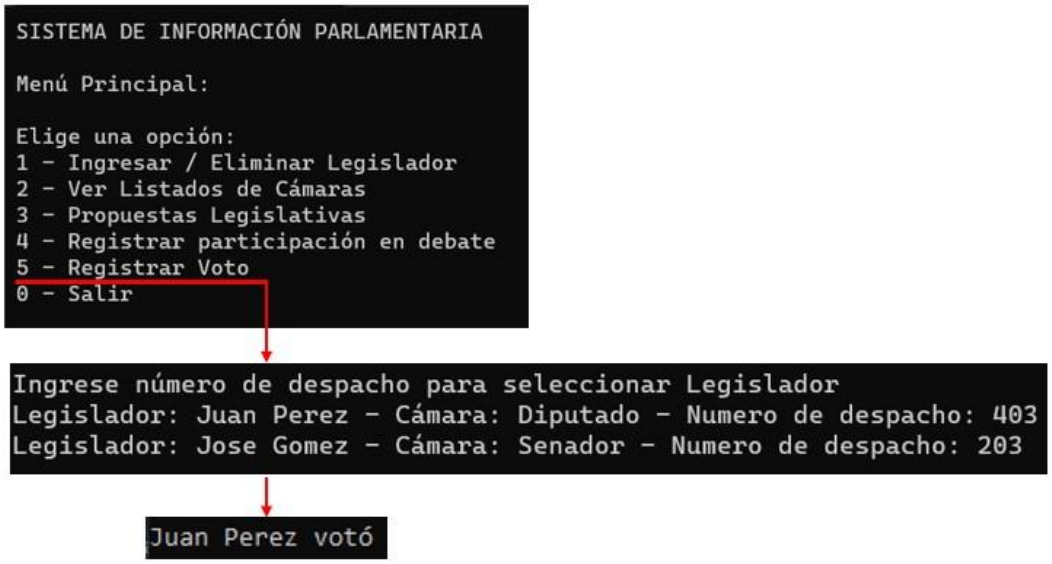

Figura 3.6

4. Conclusiones

El presente proyecto demuestra una solución efectiva al problema planteado, habiendo logrado cumplir con todos los requisitos solicitados, además de tener en cuenta ciertos aspectos no especificados, como las validaciones de datos ingresados por los usuarios, la disponibilidad de algunas porciones de información precargada en el sistema, etc.

Consideramos haber logrado un buen desarrollo de acuerdo a las herramientas con las que disponemos y respetando el paradigma de programación orientada a objetos. Pudimos poner en práctica los conceptos abordados, como herencia, polimorfismo y clases abstractas y tener una mejor comprensión de todos ellos funcionando en conjunto.

Aplicamos la herramienta “Listas”, un concepto nuevo para nosotros, el cual logramos adecuar al proyecto. También en este orden, incorporamos el uso de GitHub para compartir nuestro trabajo ordenadamente y tener a disposición los cambios y crecimiento del mismo.

Como aspecto a mejorar mencionaremos las validaciones de datos que, aunque funcionan correctamente, nos hubiese gustado crearlas dentro de métodos y así, lograr un desarrollo de código más reducido y ordenado.

En términos generales estamos conformes con el resultado, hemos aprendido algunas herramientas nuevas que esperamos poder seguir descubriendo y profundizando en el futuro.

5. Referencias

[1] - https://desarrolloweb.com/articulos/499.php

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

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

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