

Reportes para los 7 archivos

Archivo:		BosqueEncantado.cs	
	Método	# Línea(s)	Tipo de error y descripción
1.	BosqueEncantado (Constructor)	18	Naming: El nombre "int_decision" se consideraría un error de notación húngara , ya que contiene el tipo de datos int como prefijo en el nombre. En su lugar, se debería utilizar un nombre más descriptivo que refleje el propósito de la variable, como decisión o algo similar.
2.	BosqueEncantado (class)	8-10	Comments: Los comentarios en el código como "//LAGO ENCANTADO: 1", "//ARBOL: 2", "//VACIO: 3" son innecesarios y no aportan información adicional que no sea obvia a partir del contexto.
3.	cambiarnumerosdelbosque	85-124	Violación del principio de responsabilidad única (SOLID): El método cambiarnumerosdelbosque tiene múltiples responsabilidades, como calcular los contadores, realizar cambios en la matriz crear y una instancia de la clase Pantalla. Sería mejor dividir este método en funciones más pequeñas y enfocarse en una única responsabilidad para cada método.
4.	BosqueEncantado (Constructor)	17,18	El uso inconsistente de mayúsculas y minúsculas en los nombres de variables (doubleAleatorio, int_decision) viola las convenciones de capitalización establecidas. Se recomienda seguir una convención de capitalización consistente, como camel case, para mejorar la legibilidad y la coherencia del código.
5.	cambiarnumerosdelbosque	120,121	Rompe el principio Dependency Inversion Principle: en la clase Pantalla para crear una instancia y llamar al método crear(). Esto acopla la clase BosqueEncantado a la clase Pantalla, lo que dificulta la reutilización y la prueba. Sería mejor utilizar una abstracción o una interfaz y aplicar la inversión de dependencia para desacoplar las clases y permitir una mayor flexibilidad y extensibilidad.
6.	num_lag_vec	23,57	Falta de manejo de excepciones: El código no incluye ningún manejo de excepciones para capturar errores posibles durante la generación de números aleatorios (Math.random()) o el acceso a elementos de la matriz (matriz[pf,pc]). Es importante agregar bloques de manejo de excepciones adecuadas para garantizar la robustez y el manejo de errores adecuados del código.
7.		87-119	Falta de indentación adecuada: El código no sigue una

	cambiarnumerosdelbosque		sangría y espacio consistente, lo cual dificulta la lectura y comprensión del código. Un formato de código consistente con sangría y espacio adecuado mejora significativamente la legibilidad.
8.	BosqueEncantado (Constructor)	25-41	Mala encapsulación de condicionales: Hay redundancia en las condiciones ,el bloque if que verifica <code>int_decision != 3 && int_decision != 2 && int_decision != 1</code> es redundante, ya que se ejecuta después de las condiciones anteriores y no hace nada. No tiene un efecto real en la lógica del programa.
9.	num_lag_vec	47	Function arguments (2 or fewer ideally): El metodo <code>num_lag_vec</code> tiene 3 parametros que podrian ser menos ,de acuerdo con el principio de "Argumentos de función" de Clean Code, se sugiere tener idealmente dos o menos argumentos en una función.
10.	num_lag_vec	47-83	Hay duplicación innecesaria en los métodos <code>num_lag_vec</code> y <code>num_arb_vec</code>: Ambos métodos tienen una estructura muy similar y realizan operaciones casi idénticas. Puede refactorizar el código para eliminar la duplicación y hacerlo más legible y mantenible.

Archivo:		Hangman.cs	
Método		# Línea(s)	Tipo de error y descripción
11.	PlayHangman	15	Nombres poco descriptivos: En el método <code>PlayHangman()</code> , se usa la variable <code>idx</code> para representar el índice generado aleatoriamente, lo cual no brinda información clara sobre su propósito.
12.	PlayHangman	49	Comentarios necesarios: Hay algunos comentarios necesarios y redundantes que no aportan información adicional útil. Por ejemplo, el comentario en el bucle for dentro del método <code>PlayHangman()</code> que dice <code>"//Sigue jugando"</code> . Este tipo de comentarios obvios no son necesarios y solo agregan ruido al código.
13.	perdio	61-74	Bloques de código comentados: Existen bloques de código comentados que no se utilizan actualmente y no deberían estar presentes en el código. Por ejemplo, el método <code>perdio()</code> está completamente comentado y no se llama desde ningún otro lugar del código.
14.	readdata	89-103	Método no utilizado: El método <code>readdata()</code> no se utiliza en ninguna parte del código, pero no se declara como estático. Si un método no se utiliza y no requiere acceder a instancias específicas de la clase, se recomienda declarar como <code>static</code> para indicar que pertenece a la clase en lugar de las instancias.
15.	PlayHangman()	13,14,15	Métodos y variables no privados: Algunos métodos y variables en la

			clase Hangman no están declarados como privados, lo que podría exponer necesariamente su acceso desde fuera de la clase. Es una buena práctica encapsular adecuadamente los miembros de una clase y limitar su acceso según sea necesario. Por ejemplo la variable file.
16.	PlayHangman()	37	Uso de excepciones inapropiadas: En el manejo de la entrada del jugador, se utiliza una excepción FormatException para capturar cualquier error de formato al convertir la entrada a un personaje. Sin embargo, en lugar de capturar la excepción, sería mejor validar y manejar adecuadamente la entrada del jugador sin recurrir a excepciones.
17.	PlayHangman()	8-58	Rompe el principio de responsabilidad unica: En lugar de tener toda la lógica del juego dentro de un único método estático PlayHangman(), sería más limpio y mantenible dividir la funcionalidad en métodos más pequeños y bien definidos. Esto permitiría separar las diferentes etapas del juego (inicialización, adivinanza del jugador, validación del resultado) en métodos separados, lo que facilita su comprensión, prueba y posible reutilización en el futuro.

Archivo: Letters.cs			
	Método	# Línea(s)	Tipo de error y descripción
18.	Count	10	Naming: El nombre "strCaptured" se consideraría un error de notación húngara , ya que contiene el tipo de datos str como prefijo en el nombre. En su lugar, se debería utilizar un nombre más descriptivo que refleje el propósito de la variable, como decisión o algo similar.
19.	Count	7-40	Función con múltiples responsabilidades: El método Count() realiza varias tareas, como capturar la entrada del usuario, realizar el recuento de vocales y consonantes, y mostrar los resultados en la consola. Es recomendable que cada método se enfoque en una única responsabilidad para mejorar la legibilidad y facilitar el mantenimiento.
20.	Count	7-40	Falta de comentarios y documentación: El código carece de comentarios que explican su funcionamiento, lo que dificulta la comprensión para otros desarrolladores. Es importante agregar comentarios que expliquen el propósito de las secciones de código y cualquier lógica compleja.
21.	Count	10	Falta de manejo de errores: El código asume que la entrada del usuario es válida y no maneja casos de entrada incorrectos. Sería apropiado agregar validaciones y manejo de errores para garantizar una ejecución segura y predecible del programa. Una parte específica donde se podría aplicar el manejo de errores es en la lectura de la entrada del usuario. Actualmente, el código asume que la entrada del usuario será válida y no realiza ninguna validez.

22.	Count	17,21	Código duplicado: El código para verificar si un carácter es una voz se repite dos veces, una vez para minúsculas y otra vez para mayúsculas. Sería más eficiente y limpio tener una única implementación para verificar tanto las vocales minúsculas como las mayúsculas.
23.	Count	17,23	Uso de valores mágicos: El código utiliza valores numéricos directamente en varias partes, como en el bucle "for" para y en las comparaciones de caracteres. Estos valores numéricos sin explicación se conocen como "valores mágicos" y dificultan la comprensión y el mantenimiento del código. Es preferible utilizar constantes con nombres descriptivos o variables declaradas con nombres significativos en lugar de valores numéricos directos.
24.	Count	11	Uno de los errores de estructura de datos en el código es el uso de un arreglo de caracteres (char[]) "sentence" para almacenar la oración ingresada por el usuario. Esto limita la flexibilidad y funcionalidad del programa, ya que un arreglo de caracteres no proporciona métodos y operaciones convenientes para trabajar con texto. En lugar de utilizar un arreglo de caracteres, sería más apropiado utilizar una estructura de datos más robusta y adecuada para el manejo de texto, como una cadena (string). La utilización de cadenas permitiría aprovechar los métodos y propiedades disponibles para manipular y analizar el texto de manera más eficiente y legible.

Archivo:		NumberGuessing.cs	
Método		# Línea(s)	Tipo de error y descripción
25.	NumberGuessing (class)	44,45,46	Comentarios inútiles: El código tiene comentarios que no agregan información útil y solo repiten lo que el código ya indica claramente. Los comentarios deben utilizarse para explicar el por qué y no el qué, ya que el qué debe ser evidente a través del código en sí mismo.
26.	Play	11,12,16	Uso de nombres de variables poco descriptivos: El código utiliza nombres de variables como "randno", "input", "count", etc., que no son lo suficientemente descriptivos. Sería preferible utilizar nombres más significativos que reflejen el propósito o el contenido de las variables.
27.	Play	7-42	Falta de comentarios explicativos: El código carece de comentarios que expliquen el propósito o la lógica detrás de ciertas secciones, lo cual dificulta la comprensión y mantenibilidad del código. En la función Play(): Sería útil agregar comentarios que expliquen la estructura del juego, cómo se generan los números aleatorios, cómo se solicita la entrada del usuario y cómo se determina si el número adivinado es correcto.
28.	Play	16	Falta de manejo de errores: El código asume que la entrada del usuario es válida y no maneja casos de entrada incorrectos. Sería apropiado agregar validaciones y manejo de errores para garantizar una ejecución segura y predecible del programa. En la línea 16 sería útil un manejo de errores que capte números que no cumplen con el intervalo o caracteres

			inválidos.
29.	Play	7-42	Falta de separación de responsabilidades: El método Play() realiza demasiadas tareas en un solo bloque de código. Sería preferible separar las responsabilidades en métodos más pequeños y bien definidos.
30.	Play	7-42	Error relacionado con la violación del principio de "Objects and Data Structures": En el código dado, se utiliza una clase llamada "NumberGuessing" que tiene un método estático "Play" para jugar al juego de adivinar números. Sin embargo, esta implementación está más orientada a la estructura de datos en lugar de a los objetos. La clase no encapsula adecuadamente la lógica del y, en cambio, exponen detalles internos, como la generación de números aleatorios y el bucle principal, directamente en el método estático.
31.	Play	7-42	Falta de pruebas unitarias: No se incluyen pruebas unitarias en el código proporcionado. Las pruebas unitarias son esenciales para garantizar que el código funcione correctamente y para facilitar su mantenimiento y refactorización. Sería recomendable agregar pruebas unitarias para cubrir diferentes casos de uso y escenarios.

	Archivo:	Roman.cs	
	Método	# Línea(s)	Tipo de error y descripción
32.	From	73	Nombres de variables poco descriptivos: Algunos nombres de variables como "total" no es descriptiva y no transmite su propósito o significado.
33.	Combine	55	Método Combine mal diseñado: El método Combine utiliza el tipo dinámico para los parámetros val1 y val2, lo cual hace que el código sea menos seguro y menos legible. Sería mejor especificar los tipos esperados de manera limpia.
34.	Combine	48	Lanzamiento genérico de excepciones: El código lanza una excepción genérica Exception("Must be of type Number") en caso de que los valores val1 y val2 no se puedan convertir a enteros. Sería preferible utilizar una excepción más específica y proporcionar un mensaje de error más descriptivo.
35.	From	71-99	Lógica confusa en el método From: El método From tiene una lógica complicada y poco clara para convertir números romanos a enteros. La lógica puede ser difícil de seguir y entender, lo que dificulta la mantenibilidad del código.
36.	Roman	14-23	Uso inconsistente de formato y estilo: El código tiene inconsistencias en la sangría y el espacio, lo que afecta la legibilidad y la coherencia del código.
37.	From	71-99	Falta de comentarios explicativos: El código carece de comentarios que explican el propósito, la lógica o las limitaciones de ciertas secciones o

			métodos. En el método From(string roman), se podría agregar comentarios para explicar cómo se realiza la conversión de un número romano a un número entero.
38.	Roman (class)	12-41	Violación del principio Responsabilidad única: La clase Roman tiene múltiples responsabilidades, como el manejo de diccionarios y la conversión de números romanos. Sería recomendable dividir la funcionalidad en clases más pequeñas y cohesivas.
39.	Combine	43	No se evita el uso de condicional excesivo: se utiliza un condicional negativo con el operador de negación "!" para verificar si no se puede convertir val1 o val2 a tipo entero. Además también lo hace en otros lugares del código , lo cual es mala práctica y un error de clean code
40.	Roman (class)	9,10	Make objects have private/protected members: Hay dos variables globales public por lo que es inseguro y cualquiera puede obtenerlas desde afuera , es mejor hacerlas privadas y crear getter y setter si es el caso de si se quiere crear un acceso. En este caso se ve que es necesario acceder a esas variables solamente en la clase donde se crean , por lo que deberían ser privadas

Archivo: Pantalla.cs			
	Método	# Línea(s)	Tipo de error y descripción
41.	Crear	6	Convención de nombres de metodos: El nombre de la clase, Pantalla, no sigue las convenciones de nombres de C#. Debería seguir el formato CamelCase, donde cada palabra comienza con mayúsculas en nombres de métodos.
42.	Crear	6-16	Falta de comentarios explicativos: El código carece de comentarios que explican el propósito, la lógica o las limitaciones de ciertas secciones o métodos. Los comentarios descriptivos son importantes para comprender rápidamente el funcionamiento del código. Por ejemplo se podría agregar uno que explique el sentido del método crear() y dentro de este comentarios que explique su lógica.
43.	Pantalla (class)	4	Falta de encapsulación: Los miembros de la clase, como matrizEnteros, se definen como públicos, lo que viola el principio de encapsulación. Sería más adecuado utilizar propiedades y definir los miembros como privados para controlar el acceso desde fuera de la clase.
44.	Pantalla (class)	4	Nombres de variables poco significativos: Los nombres de variables como "matrizEnteros" no es descriptivo y no distingue el propósito de las variables.
45.	Pantalla (class)	4	Uso de Notación Húngara - Tipo: Declaraciones y Nomenclatura. El nombre de la variable "matrizEnteros" utiliza notación húngara, una convención de nomenclatura desaconsejada en el paradigma de programación moderna. Se recomienda utilizar nombres más descriptivos y legibles sin incluir información redundante sobre el tipo de la variable.

46.	Pantalla (constructor)	3	Uso incorrecto de la palabra clave 'var' en el constructor de la clase: Este error pertenece al tipo de Naming, ya que se recomienda especificar un tipo concreto en lugar de utilizar 'var' en los parámetros de los métodos o constructores.
47.	Pantalla (constructor)	3-5	Falta de manejo de errores: No se realiza ninguna validación en el constructor para verificar si la matriz proporcionada es nula o tiene dimensiones válidas. Sería bueno agregar una validación para garantizar que se proporcione una matriz válida.

	Archivo:	Program.cs	
	Método	# Línea(s)	Tipo de error y descripción
48.	Program (class)	5-10	Comentarios necesarios: Los comentarios con fechas y nombres de juegos anteriores no son relevantes para el funcionamiento del programa y solo agregan ruido necesario al código. Es recomendable eliminar comentarios obsoletos o irrelevantes.
49.	Main	14-17	Código muerto: código comentado que ya no se utiliza, si el código ya no está en uso es mejor eliminarlo

Soluciones a 5 problemas encontrados durante la revisión de código. En las soluciones se debe poder identificar:

1. (2pt) Problema que está corrigiendo (de los identificados en las tablas del punto anterior).
2. (2pt) Segmento de código o screenshot que muestre la versión con el problema.
3. (2pt) Breve explicación sobre a qué se debe el error y por qué es importante de corregir.
4. (2pt) Segmento de código o screenshot que muestre la versión corregida.
5. (2pt) Breve explicación de su solución y justificación de por qué dicha solución es adecuada.

A.

1. Se está corrigiendo que en el método Combine del archivo **Roman.cs** se utiliza el tipo **dinámico** para los parámetros val1 y val2, lo cual hace que el código sea menos seguro y menos legible

2.

```

0 referencias
43 public int Combine(dynamic val1, dynamic val2)
44 {
45     int value;
46     if (!int.TryParse(val1, out value) || !int.TryParse(val2, out value))
47     {
48         throw new Exception("Must be of type Number");
49     }
50
51     return val1 + val2;
52 }
53

```

3. El uso de `dynamic` en el método `Combine` rompe los principios de código limpio por varias razones:

- Falta de claridad en los tipos de datos: Al utilizar `dynamic`, no queda claro qué tipo de datos se esperan como entrada para `val1` y `val2`. El código se vuelve menos legible y dificulta la comprensión del propósito y comportamiento del método.
- Pérdida de verificación estática de tipos: Al usar `dynamic`, se pierde la verificación estática de tipos en tiempo de compilación. Esto significa que los errores relacionados con el tipo de datos no se detectarán hasta que se ejecute el programa, lo que puede llevar a errores difíciles de depurar.
- Mayor riesgo de excepciones en el tiempo de ejecución: Al realizar el `TryParse` en los valores `val1` y `val2`, existe el riesgo de que no sean convertibles a `int` y se genere una excepción en el tiempo de ejecución. Esto podría evitarse utilizando tipos específicos en lugar de dinámica y realizando una conversión segura en tiempo de compilación.

4.

```
43 public int Combine(int val1, int val2)
44 {
45     return val1 + val2;
46 }
```

5. Una posible solución sería cambiar el tipo de los parámetros `val1` y `val2` a `int` en lugar de `dynamic`. Esto permitiría tener una mayor claridad en los tipos de datos esperados y aprovechar la verificación estática de tipos en el tiempo de compilación. Además, eliminaría el riesgo de excepciones en el tiempo de ejecución al no depender del `TryParse`. Al utilizar `int` como tipo de los parámetros, se asegura que solo se pueden pasar valores enteros al método. Esto facilita la comprensión del código y evita posibles errores relacionados con los tipos de datos.

B.

1. **Se está arreglando el problema de nombres poco descriptivos en el archivo `Hangman.cs`.** En el método `PlayHangman()` línea 15, se usa la variable `idx` para representar el índice generado aleatoriamente, lo cual no brinda información clara sobre su propósito.

2.

```
8 public static void PlayHangman()
9 {
10
11     Console.WriteLine("Welcome to Hangman!!!!!!!!!!!!");
12
13     StreamReader file = new StreamReader("usa.txt");
14     Random randGen = new Random();
15     var idx = randGen.Next(0, 61333);
16     //var idx = randGen.Next(0, 10);
```


3. El nombre de la variable `idx` en el código proporcionado es un ejemplo de nombre no descriptivo, lo cual es considerado una mala práctica en Clean Code. El uso de nombres no descriptivos dificulta la comprensión del código y puede llevar a confusiones o errores al leer o dar mantenimiento al código en el futuro. En este caso, `idx` no comunica claramente el propósito de la variable ni qué representa. Para mejorar la legibilidad y comprensión del código, es recomendable utilizar un nombre más descriptivo que indique claramente el propósito de la variable.

4.

```
0 referencias
8 public static void PlayHangman()
9 {
10
11     Console.WriteLine("Welcome to Hangman!!!!!!!!!!!!");
12
13     StreamReader file = new StreamReader("usa.txt");
14     Random randGen = new Random();
15     int randomIndex = randGen.Next(0, 61333);
```

5. En esta versión, se ha cambiado el nombre de la variable `idx` a “**randomIndex**”, que es más descriptivo y comunica claramente que se trata de un índice aleatorio utilizado para seleccionar una palabra del archivo "usa.txt". Esta modificación mejora la legibilidad del código y facilita la comprensión de su funcionalidad.

C.

1. **Hay duplicación innecesaria en los métodos `num_lag_vec` y `num_arb_vec`:** Ambos métodos tienen una estructura muy similar y realizan operaciones casi idénticas. Esto ocurre en el archivo **BosqueEncantado.cs** en las líneas desde la 47 hasta la 83.

2.

```
1 referencia
public int num_lag_vec(int pf, int pc, int pIntCont, var matriz)
{
    for (int estaFila = pf - 1; estaFila <= pf + 1; ++estaFila)
    {
        for (int estaColumna = pc - 1; estaColumna <= pc + 1; ++estaColumna)
        {
            if (!(estaFila == pf && estaColumna == pc))
            {
                if (0 <= estaFila && 0 <= estaColumna && pf < cantidadFilas && pc < cantidadColumnas)
                {
                    if (matriz[pf,pc] == 1)
                    {
                        ++pIntCont;
                    }
                }
            }
        }
    }
    return pIntCont;
}

1 referencia
public int num_arb_vec(int pf, int pc, int pIntCont, var pmatriz)
{
    for (int estaFila = pf - 1; estaFila <= pf + 1; ++estaFila)
    {
        for (int estaColumna = pc - 1; estaColumna <= pc + 1; ++estaColumna)
        {
            if (!(estaFila == pf && estaColumna == pc))
            {
                if (0 <= estaFila && 0 <= estaColumna && pf < cantidadFilas && pc < cantidadColumnas)
                {
                    if (pmatriz[pf,pc] == 2)
                    {
                        ++pIntCont;
                    }
                }
            }
        }
    }
    return pIntCont;
}
```

3. La redundancia de este código puede afectar la calidad del código de las siguientes maneras:

- Mantenimiento complicado: Si hay un error o se requiere una modificación en la lógica compartida entre los dos métodos, es necesario realizar los cambios en ambos lugares, lo que aumenta la posibilidad de cometer errores y hace que el mantenimiento sea más complicado.
- Duplicación innecesaria: La duplicación de código va en contra de los principios del código limpio, ya que aumenta la cantidad de código que se debe escribir, leer y mantener. Además, cualquier modificación futura prevalecerá en múltiples lugares, lo que aumenta la posibilidad de errores.
- Dificultad para escalar y agregar nuevas funcionalidades: Si se desea agregar más métodos similares en el futuro, se repetiría la misma estructura y lógica duplicada, lo que dificulta la escalabilidad y la introducción de nuevas funcionalidades de manera eficiente.

4.

```
2 referencias
47 public int ContarCeldasAdyacentes(int pf, int pc, int targetValue)
48 {
49     int pIntCont = 0;
50
51     for (int estaFila = pf - 1; estaFila <= pf + 1; ++estaFila)
52     {
53         for (int estaColumna = pc - 1; estaColumna <= pc + 1; ++estaColumna)
54         {
55             if (!(estaFila == pf && estaColumna == pc))
56             {
57                 if (0 <= estaFila && estaFila < cantidadFilas && 0 <= estaColumna && estaColumna < cantidadColumnas)
58                 {
59                     if (matrizBosque[estaFila, estaColumna] == targetValue)
60                         ++pIntCont;
61                 }
62             }
63         }
64     }
65
66     return pIntCont;
67 }
68
```

5. En este nuevo método **ContarCeldasAdyacentes**, hemos combinado las funcionalidades de los métodos `num_lag_vec` y `num_arb_vec` en una única función. Esta función toma un parámetro adicional `targetValue` para indicar el valor objetivo que se desea contar en las celdas adyacentes. Pero se elimina el parámetro de matriz, ya que esta matriz es una variable global y se puede usar en todos los métodos de la clase. Al combinar ambos métodos, evitamos la duplicación de código y reducimos la redundancia. Además, ahora podemos utilizar este método genérico para contar celdas adyacentes con diferentes valores de objetivo, lo que aumenta la flexibilidad y la reutilización del código.

D.

1. **Falta de manejo de errores:** El código asume que la entrada del usuario es válida y no maneja casos de entrada incorrectos. Esto ocurre en la línea 16 del método `Play()` en el archivo de "NumberGuessing.cs".

2.

```
0 referencias
7 public static void Play()
8 {
9     while (true)
10     {
11         int randno = Newnum(1, 101);
12         int count = 1;
13         while (true)
14         {
15             Console.WriteLine("Enter a number between 1 and 100(0 to quit):");
16             int input = Convert.ToInt32(Console.ReadLine());
17             if (input == 0)
18                 return;
19             else if (input < randno)
20             {
21                 Console.WriteLine("Low, try again.");
22                 ++count;
23                 continue;
24             }
25         }
26     }
27 }
```

3. El código tiene un problema potencial en la línea donde se lee la entrada del usuario. La falta de manejo de errores al convertir la entrada del usuario a un número entero puede causar una excepción si el usuario ingresa un valor que no se puede convertir. Por ejemplo, si el usuario ingresa un texto en lugar de un número, se producirá una excepción "FormatException". Rompiendo la eficiencia del código.

4.

```
0 referencias
public static void Play()
{
    while (true)
    {
        int randno = Newnum(1, 101);
        int count = 1;
        while (true)
        {
            Console.Write("Enter a number between 1 and 100 (0 to quit): ");
            string inputString = Console.ReadLine();

            if (inputString == "0")
                return;

            if (int.TryParse(inputString, out int input))
            {
                if (input < randno)
                {
                    Console.WriteLine("Low, try again.");
                    ++count;
                    continue;
                }
                else if (input > randno)
                {
                    Console.WriteLine("High, try again.");
                    ++count;
                    continue;
                }
                else
                {
                    Console.WriteLine("You guessed it! The number was {0}!", randno);
                    Console.WriteLine("It took you {0} {1}.\n", count, count == 1 ? "try" : "tries");
                    break;
                }
            }
            else
            {
                Console.WriteLine("Invalid input. Please enter a valid number.");
            }
        }
    }
}
```

5. En este código, después de leer la entrada del usuario como una cadena de texto, se utiliza el método `int.TryParse` para intentar convertirlo a un número entero. Si la conversión es exitosa, el número convertido se guarda en la entrada variable y se continúa con la lógica del juego. Si la conversión falla, se muestra un mensaje de error indicando que se debe ingresar un número válido. Este manejo sencillo de errores asegura que el programa no se rompe si el usuario ingresa una entrada no válida y proporciona una mejor experiencia al usuario al solicitarle que ingrese un número válido.

E.

1. **Bloques de código comentados:** Existen bloques de código comentados que no se utilizan actualmente y no deberían estar presentes en el código. Esto está en el método `perdio` desde la línea 61 hasta la 74, en el archivo `Hangman.cs`

2.

```
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
1 referencia

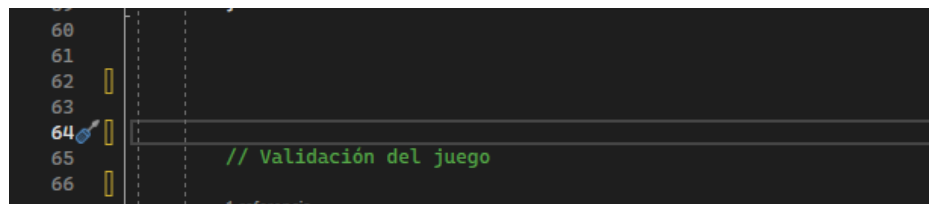
/*public static bool perdio(string word1, string word2, int counter)
{
    if (word1.Equals(word2))
    {
        Console.Write("Congratulations the word was: " + word2);
        return true;
    }
    if (counter > 10)
    {
        Console.Write("You lose, the word was: " + word2);
        counter++;
    }
    return false;
}*/

// Validación del juego
```

3. Un método completamente comentado puede romper el principio de Clean Code porque se considera "código muerto" o código que no se ejecuta y que puede confundir a los lectores. Además, los comentarios deben ser utilizados para explicar el por qué y el cómo del código, no para desactivar o eliminar partes del código.

En el código proporcionado, el método periodo() está completamente comentado, lo que implica que no se está utilizando en el programa actual. Si ese método ya no se necesita y no se va a utilizar en el futuro, es una buena práctica eliminarlo completamente en lugar de dejarlo como un código comentado. En este código no se llama en ninguna parte, además de que no hay pruebas de depuración, por lo que se puede considerar código muerto.

4.

A screenshot of a code editor with a dark background. On the left, a vertical line of line numbers is visible, with 64 highlighted in blue. The code on line 64 is a single line of text: `// Validación del juego`. The line is commented out, indicated by a green double-slash at the beginning. The rest of the code on the line is in a light green color. The editor has a light gray border on the right side.

5. Se eliminó el código del método comentado con lo cual pudimos alcanzar los siguientes beneficios:

- Mejoró la claridad y legibilidad del código.
- Evita confusiones y malentendidos.
- Facilita el mantenimiento y refactorización del software.
- Reduce el tamaño del archivo y mejora el rendimiento.

En resumen, al eliminar este código comentado, mejora la legibilidad, claridad y mantenibilidad del código, evita confusiones y facilita futuras modificaciones. También puede tener un impacto positivo en el rendimiento al reducir el tamaño del archivo fuente.

Enlace del repositorio de github:

https://github.com/AndresMatarritaC04668/c04668_ci0126_23a.git

Agregue un pequeño resumen, no más de 200 palabras indicando claramente

- 1. Dos cosas que no sabía y aprendió en el laboratorio**
- 2. Una cosa que se le hizo difícil de realizar y explique por qué fue difícil.**
- 3. Una cosa que se le hizo fácil de realizar y explique por qué fue fácil.**
- 4. Indique cuánto tiempo tardó en realizar el laboratorio.**

En el laboratorio de Clean Code, aprendí dos cosas importantes:

1. Uso adecuado de excepciones: Aprendí que es importante utilizar excepciones más específicas en lugar de la clase base `Exception`. Esto proporciona información más precisa sobre el tipo de error que ocurrió y ayuda a mejorar la legibilidad y mantenibilidad del código.
2. Evitar el uso necesario de tipos dinámicos: Descubrí que es preferible especificar directamente el tipo esperado para los parámetros en lugar de utilizar `dynamic`. Esto mejora la claridad y permite realizar verificaciones estáticas de tipos en tiempo de compilación.

En cuanto a las dificultades, encontrar problemas específicos en el código fue un desafío. Requería un análisis minucioso y conocimiento de los principios de código limpio. Sin embargo, con la guía de identificación de problemas y las pautas de código limpio, pude identificar los problemas y proponer soluciones adecuadas.

La parte más fácil del laboratorio fue corregir el uso necesario de `dynamic` en el método `Combine`. Fue fácil porque la solución era cambiar los parámetros a `int` y realizar la suma correctamente.

En total, tardé aproximadamente 6 horas en buscar y corregir los errores encontrados en el código proporcionado.