de Nacho Iborra

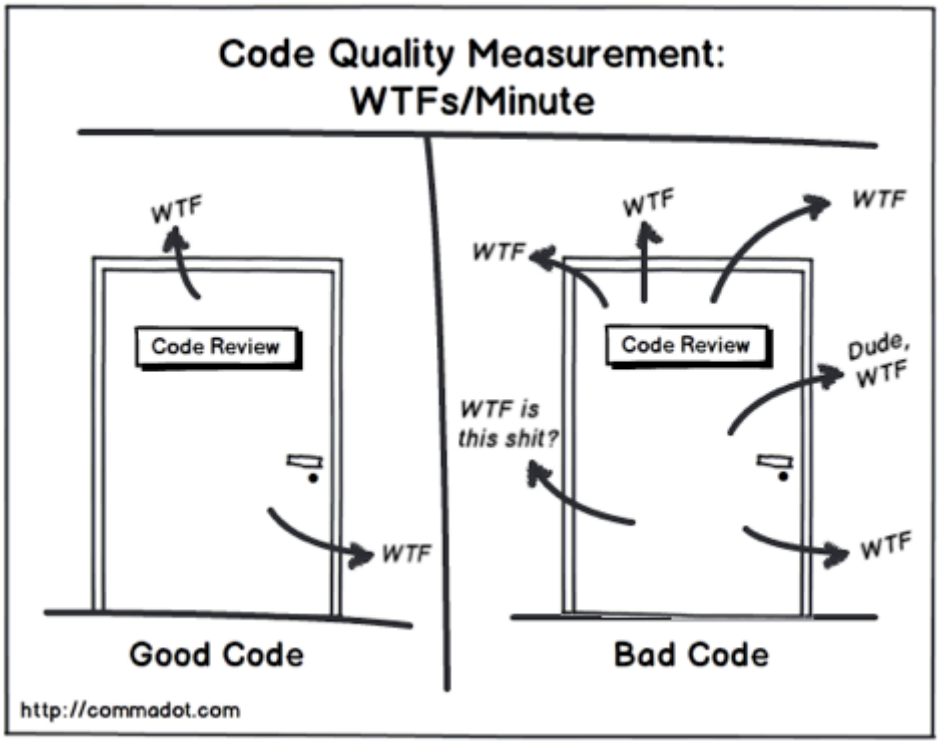
Ambientes de desarrollo

Bloque 1

Unidad 9: Mejores prácticas de programación

1.9.1. Introducción al código limpio

Como introducción a lo que queremos que "domine" como buenas pautas de programación, leamos algunos extractos del libro. *Código limpio,* por Robert C. Martin. Para empezar, aquí puede ver una representación gráfica de cómo medir la calidad del código:



1.9.1.1. La importancia de la práctica

El aprendizaje de la artesanía consta de dos partes: conocimiento y trabajo. Debe adquirir el conocimiento de los principios, patrones, prácticas y heurísticas que un artesano conoce, y también debe moler ese conocimiento en sus dedos, ojos e intestinos trabajando duro y practicando.

Puedo enseñarte la física de andar en bicicleta. De hecho, las matemáticas clásicas son relativamente sencillas. La gravedad, la fricción, el momento angular, el centro de masa, etc., se pueden demostrar con menos de una página llena de ecuaciones. Dadas esas fórmulas, podría probarte que andar en bicicleta es práctico y darte todos los conocimientos que necesitas para que funcione. Y todavía te caerías la primera vez que te subiste a esa bicicleta.

La codificación no es diferente. [...]

Aprender a escribir código limpio es un trabajo duro. Requiere algo más que el conocimiento de principios y patrones. Debes sudar por eso. Debes practicarlo tú mismo y ver cómo fracasas. Debes ver a otros practicarlo y fallar. Debes verlos tropezar y volver sobre sus pasos. Debe verlos agonizar por las decisiones y ver el precio que pagan por tomar esas decisiones de manera incorrecta.

1.9.1.2. Efectos del código incorrecto

Sé de una empresa que, a finales de los 80, escribió una aplicación espectacular. Fue muy popular y muchos profesionales lo compraron y usaron. Pero luego los ciclos de liberación comenzaron a estirarse. Los errores no se repararon de una versión a otra. Los tiempos de carga aumentaron y los accidentes aumentaron. Recuerdo el día en que apagué el producto por la frustración y no lo volví a usar. La empresa cerró poco tiempo después de eso.

Dos décadas después conocí a uno de los primeros empleados de esa empresa y le pregunté qué había sucedido. La respuesta confirmó mis temores. Habían llevado el producto al mercado rápidamente y habían hecho un gran lío en el código. A medida que agregaban más y más funciones, el código empeoraba cada vez más hasta que simplemente ya no podían administrarlo. Fue el código incorrecto lo que derribó a la empresa.

1.9.1.3. La imposibilidad de un gran rediseño

[...] *Finalmente, el equipo se rebela. Informan a la gerencia que no pueden continuar desarrollándose en esta odiosa base de código. Exigen un rediseño. La gerencia no quiere gastar los recursos en un rediseño completamente nuevo del proyecto, pero no puede negar que la productividad es terrible. Finalmente, se someten a las demandas de los desarrolladores y autorizan el gran rediseño en el cielo.*

Se selecciona un nuevo equipo de tigres. Todo el mundo quiere estar en este equipo porque es un proyecto totalmente nuevo. Pueden empezar de nuevo y crear algo realmente hermoso. Pero solo los mejores y más brillantes son elegidos para el equipo tigre. Todos los demás deben continuar manteniendo el sistema actual.

Ahora los dos equipos están en carrera. El equipo tigre debe construir un nuevo sistema que haga todo lo que hace el antiguo. No solo eso, tienen que mantenerse al día con los cambios que se realizan continuamente en el sistema antiguo. La administración no reemplazará el sistema antiguo hasta que el nuevo sistema pueda hacer todo lo que hace el sistema antiguo.

Esta carrera puede durar mucho tiempo. Lo he visto tardar 10 años. Y para cuando está terminado, los miembros originales del equipo Tiger ya no están, y los miembros actuales exigen que el nuevo sistema sea rediseñado porque es un desastre.

1.9.1.4. Algunas razones por las que existe un código incorrecto

¿Alguna vez has atravesado un lío tan grave que te llevó semanas hacer lo que debería haber tomado horas? ¿Ha visto lo que debería haber sido un cambio de una línea, realizado en su lugar en cientos de módulos diferentes? Estos síntomas son demasiado comunes.

¿Por qué le sucede esto al código? ¿Por qué el buen código se descompone tan rápidamente en código incorrecto? Tenemos muchas explicaciones para ello. Nos quejamos de que los requisitos cambiaron de manera que frustran el diseño original. Lamentamos los horarios que eran demasiado ajustados para hacer las cosas bien. Hablamos de gerentes estúpidos y clientes intolerantes y

tipos de marketing inútiles y desinfectantes telefónicos. Pero la culpa, querido Dilbert, no está en nuestras estrellas, sino en nosotros mismos. No somos profesionales.

Esta puede ser una píldora amarga de tragar. ¿Cómo pudo ser culpa nuestra este desastre? ¿Y los requisitos? ¿Y el horario? ¿Qué pasa con los directores estúpidos y los tipos inútiles de marketing? ¿No tienen parte de la culpa?

No. Los gerentes y especialistas en marketing nos buscan la información que necesitan para hacer promesas y compromisos; e incluso cuando no nos miran, no debemos sentir vergüenza de decirles lo que pensamos. Los usuarios nos buscan para validar la forma en que los requisitos encajarán en el sistema. Los jefes de proyecto nos buscan para que les ayudemos a elaborar el cronograma. Somos profundamente cómplices en la planificación del proyecto y compartimos gran parte de la responsabilidad por cualquier falla; ¡especialmente si esos fallos tienen que ver con un código incorrecto! "¡Pero espera!" tu dices. "Si no hago lo que dice mi gerente, me despedirán". Probablemente no. La mayoría de los gerentes quieren la verdad, incluso cuando no actúan como tal. La mayoría de los gerentes quieren un buen código, incluso cuando están obsesionados con la programación. Pueden defender con pasión el horario y los requisitos; pero ese es su trabajo.

Para llevar este punto a casa, ¿qué pasaría si usted fuera médico y tuviera un paciente que le exigiera que detuviera todo el tonto lavado de manos en preparación para la cirugía porque estaba tomando demasiado tiempo? Claramente, el paciente es el jefe; y, sin embargo, el médico debería negarse absolutamente a cumplir. ¿Por qué? Porque el médico sabe más que el paciente sobre los riesgos de enfermedad e infección. Sería poco profesional (mucho menos criminal) que el médico cumpliera con el paciente.

Por lo tanto, tampoco es profesional que los programadores se dobleguen a la voluntad de los gerentes que no comprenden los riesgos de hacer líos.

> **Ejercicios propuestos:**

> **1.9.1.1.** ¿Estás de acuerdo con estas razones? ¿Crees que puede haber otras razones por las que existe un código incorrecto?

1.9.1.5. Algunas definiciones de "buen código"

Bjarne Stroustrup, inventor de C ++ y autor de *El lenguaje de programación C ++*

Me gusta que mi código sea elegante y eficiente. La lógica debe ser sencilla para dificultar la ocultación de los errores, las dependencias mínimas para facilitar el mantenimiento, el manejo de errores completo de acuerdo con una estrategia articulada y el rendimiento cercano al óptimo para no tentar a las personas a ensuciar el código con optimizaciones sin principios. El código limpio hace una cosa bien.

Grady Booch, autor de *Análisis y diseño orientado a objetos con aplicaciones*

El código limpio es simple y directo. El código limpio se lee como una prosa bien escrita. El código limpio nunca oculta la intención del diseñador, sino que está lleno de abstracciones nítidas y líneas directas de control.

Dave Thomas, fundador de OTI, padrino de la *Eclipse* estrategia

El código limpio puede ser leído y mejorado por un desarrollador que no sea su autor original. Tiene pruebas unitarias y de aceptación. Tiene nombres significativos. Proporciona una forma en lugar de muchas formas de hacer una cosa. Tiene dependencias mínimas, que se definen explícitamente, y proporciona una API clara y mínima. El código debe ser alfabetizado ya que, dependiendo del idioma, no toda la información necesaria se puede expresar claramente en código solo.

Michael Feathers, autor de *Trabajar eficazmente con código heredado*

Podría enumerar todas las cualidades que noto en un código limpio, pero hay una cualidad general que conduce a todas ellas. El código limpio siempre parece haber sido escrito por alguien a quien le importa. No hay nada obvio que puedas hacer para mejorarlo. Todas esas cosas fueron pensadas por el autor del código, y si intentas imaginar mejoras, volverás a donde estás, sentándote apreciando el código que alguien te dejó, código dejado por alguien que se preocupa profundamente por el arte.

Ward Cunningham, inventor de Wiki, inventor de Fit, coinventor de eXtreme Programming. Fuerza motriz detrás de los patrones de diseño. Smalltalk y líder intelectual de OO. El padrino de todos los que se preocupan por el código

Sabes que estás trabajando en un código limpio cuando cada rutina que lees resulta ser más o menos lo que esperabas. Puede llamarlo código hermoso cuando el código también hace que parezca que el lenguaje fue creado para el problema.

Y, para terminar con esta parte, al hablar de código limpio, debes tener en cuenta la regla de los Boy Scouts: *Salir el campamento más limpio de lo que lo encontraste*

> **Ejercicios propuestos:**

> **1.9.1.2.** ¿Qué definición te gusta más o crees que es la más adecuada? ¿Por qué?

> **1.9.1.3.** ¿Cómo se puede aplicar la regla de los Boy Scouts a la escritura de códigos?

1.9.2. Tratar con nombres de variables

Una vez que sepa qué es una variable y su propósito principal (almacenar valores que se pueden modificar a lo largo de la ejecución del programa), debe usar nombres significativos para sus variables. En esta sección veremos esta característica y algunas otras reglas que deben seguir los nombres de variables.

Los nombres son fundamentales en la programación, ya que asignaremos un nombre a (casi) todo lo que incluyamos en nuestro programa. En las primeras unidades asignaremos nombres a las variables, pero luego las asignaremos a funciones, parámetros, clases, archivos, espacios de nombres, etc.

1.9.2.1. Los nombres deben ser significativos

Al leer el nombre de una variable (o cualquier otro elemento en el código), debe responder algunas preguntas básicas, como por qué existe, qué hace y cómo se usa. Si un nombre requiere un comentario, entonces no es un nombre adecuado. Por ejemplo, si queremos almacenar en una variable el promedio de edad de una lista de personas, NO deberíamos hacer esto:

```
En t una;                    // Edad media
```

Podríamos hacer esto en su lugar:

```
En t ageAverage;
```

Excepción a la regla: bucles

Si está codificando un bucle, probablemente necesitará una variable entera para almacenar el valor con el número de iteraciones realizadas. Esta variable puede tener un nombre significativo, si puede (o quiere) usarlo más tarde ...

```
En t cuenta = 0 ;
mientras (cuenta < 10 ) {

    . . .
```

o simplemente un nombre corto y típico (por ejemplo, `yo` o `norte`) para usarlo SOLAMENTE en el recuento de bucles:

```
para ( En t yo = 0 ; yo < 10 ; i ++) {

    . . .
```

1.9.2.2. Evite malentendidos, pequeñas variaciones y palabras ruidosas

Debemos evitar:

- Nombres que pueden ser "falsos amigos", es decir, parecen significar algo, mientras que su intención es que signifiquen algo completamente diferente. Por ejemplo, si llamamos a una variable `cuenta` , ¿Para qué se usa esto? ¿Una cuenta bancaria? ¿Una `cuenta de usuario` en un sitio web?
- Pequeñas variaciones en los nombres de las variables. Dos variables diferentes deben tener dos nombres claramente diferentes. Si tenemos una variable llamada `totalRegisteredCustomers` , no es buena idea tener otro llamado `totalUnregisteredCustomers` , ya que podríamos mezclarlos cuando leemos sus nombres y usar el incorrecto. En lugar de esto, podemos usar `registrado` y `anónimo` , por ejemplo.
- Palabras ruidosas, es decir, palabras que forman parte del nombre de la variable pero que no proporcionan información adicional a este nombre. Por ejemplo, si una variable se llama `nameString` , la palabra *Cuerda* no tiene sentido, ya que debemos deducir que el nombre está almacenado en una variable de cadena. De la misma forma, una variable llamada `dinero` proporciona la misma información que otro llamado `moneyAmount` .

1.9.2.3. Agrega un contexto significativo

Hay algunas palabras que no son ruidosas, pero nos ayudan a establecer un nombre de variable en un contexto apropiado. Por ejemplo, si estamos hablando de los datos necesarios para almacenar la dirección de alguien (nombre, apellido, calle, ciudad, código postal ...), pero solo vemos la variable ciudad en el código, ¿podremos deducir que esta variable está almacenando parte de la dirección? Para asegurarnos de que deducimos esto, podemos agregar alguna información al nombre de la variable, como un prefijo: addressCity es más probable que esté asociado con una dirección que solo ciudad .

1.9.2.4. Elija una palabra por concepto

Trate de usar siempre el mismo nombre para expresar el mismo concepto. Por ejemplo, si implementa varias aplicaciones para varios clientes y usa una variable para almacenar el inicio de sesión de las personas que inician sesión, no llame a esta variable usuario en una aplicación, iniciar sesión en otra aplicación y así sucesivamente. Utilice siempre la misma palabra (usuario o iniciar sesión , en este caso).

De la misma forma, no uses la misma palabra para hablar de diferentes conceptos. Por ejemplo, no uses la palabra suma para todos los cálculos finales, a menos que sean realmente adiciones. Es mejor usar total , o resultado en este caso.

1.9.2.5. Otras características deseables

Además de todas las recomendaciones explicadas anteriormente, los nombres deben seguir algunas otras reglas, como:

- Debe nombrar sus variables de una manera que le permita pronunciar este nombre a otras personas, para discutir sobre su valor o errores de código con respecto a esta variable. Si está almacenando la fecha de nacimiento de alguien en una variable, podría llamarla Fecha de nacimiento , pero no deberías llamarlo ddmmyyyy , para Por ejemplo, ya que le resultaría difícil pronunciar este nombre en una discusión.
- Si usa nombres cortos en sus variables (nombres de una sola letra, por ejemplo), será difícil para usted encontrar cada ocurrencia de esta variable en su código, porque se fusionará con otros elementos que contendrán este nombre corto como parte de sus nombres.
- Hace algunos años, era muy habitual encontrar algún tipo de prefijos o sufijos en los nombres de las variables que revelaban alguna información sobre la variable. Por ejemplo, podríamos llamar a una variable edad donde prefijo *yo* mostró que esta variable era un número entero. Este hábito fue forzado por algunos lenguajes de programación antiguos donde el tipo de datos se declaraba como prefijo. Pero hoy en día hay muchos lenguajes de programación nuevos que no necesitan esta regla, y muchos IDE que nos ayudan a descubrir el tipo de cada variable fácilmente, así que no use estos prefijos.

1.9.2.6. ¿Mayúsculas o minúsculas?

El uso de letras mayúsculas y minúsculas en los nombres depende del lenguaje de programación en sí. Existen principalmente cuatro estándares de nomenclatura:

- **El caso de Carmel:** se utiliza en lenguajes como Java o Javascript. Cada palabra en el nombre de la variable comienza con mayúsculas, excepto la primera palabra. Por ejemplo:

```
String personName;
```

- Hay un subconjunto del estándar de caso camel, llamado **Estuche Pascal** en el que la primera palabra del nombre también comienza con mayúsculas. Este subconjunto es empleado por C # para definir elementos públicos (los elementos privados se nombran usando camel case). Por ejemplo:

```
String personName;
público int Personaje;
```

- **Estuche de serpiente:** se utiliza en lenguajes como PHP. Las palabras variables están separadas por guiones bajos:

```
$ person_name = "Nacho" ;
```

- **Estuche de kebab:** las palabras variables están separadas por guiones. No es muy popular entre los lenguajes de programación, ya que muchos de ellos no permiten el guión como parte del nombre de la variable (para no confundirlo con el operador de sustracción). Hay algunos ejemplos, como Lisp o Clojure.

```
( def nombre de persona "Nacho" )
```

- **Mayúsculas:** se utiliza en muchos lenguajes para definir constantes. Las palabras del nombre suelen estar separadas por guiones bajos, como en el estándar de caso de serpiente:

```
const int MAXIMUM_SIZE = 100 ;
```

1.9.3. Comentarios

Los comentarios bien ubicados nos ayudan a comprender el código que los rodea, mientras que los comentarios mal ubicados pueden dañar la comprensión del código. Algunos programadores piensan que los comentarios son fallos y deben evitarse tanto como sea posible. Una de las razones argumentadas es que son difíciles de mantener. Si cambiamos el código después de escribir un comentario, es posible que olvidemos actualizar el comentario y, por lo tanto, hablaría de algo que ya no está presente en el código.

Otra razón para evitar los comentarios es que están estrechamente vinculados a un código incorrecto. Cuando escribimos un código incorrecto, a menudo pensamos que podemos escribir algunos comentarios para hacerlo comprensible, en lugar de limpiar el código en sí.

En esta sección aprenderemos dónde poner comentarios. En primer lugar, veremos qué tipo de comentarios son necesarios (lo que llamamos *buenos comentarios),* y luego veremos qué comentarios son evitables (*malos comentarios).*

1.9.3.1. Buenos comentarios

Los siguientes comentarios se consideran necesarios:

- **Comentarios legales,** como derechos de autor o autoría, según los estándares de la empresa. Este tipo de comentarios se colocan normalmente al principio de cada archivo fuente que pertenece al autor o empresa.

- **Comentarios de introducción,** un breve comentario al principio de cada archivo fuente (normalmente clases) que explica el propósito principal de esta clase o archivo fuente.
- **Explicación de intenciones.** Estos comentarios se utilizan cuando:
    - Intentamos encontrar una mejor solución al problema pero no pudimos, y luego explicamos que una parte del código podría mejorarse.
    - Hay una parte del código que no sigue el mismo patrón que el código que lo rodea (por ejemplo, una variable entera entre un grupo de flotantes), y queremos explicar por qué hemos usado esta instrucción o tipo de datos.

- **TODO comentarios,** que se colocan en partes incompletas. Nos ayudan a recordar todas las tareas pendientes. Este tipo de comentarios se ha vuelto tan popular que muchos IDE los detectan y resaltan automáticamente.

- **Documentación de API.** Algunos lenguajes de programación, como Java o C #, nos permiten agregar algunos comentarios en algunas partes del código para que estos comentarios se exporten a formato HTML o XML, y pasen a formar parte de la documentación.

1.9.3.2. Malos comentarios

Los siguientes son ejemplos de malos comentarios que podemos evitar:

- Algún tipo de **comentarios informativos** pueden evitarse cambiando el nombre del elemento que están explicando. Por ejemplo, si tenemos este comentario con esta variable:

```
// Número total de clientes registrados
En t total;
```

Podemos evitar el comentario cambiando el nombre de la variable de esta manera:

```
En t totalRegisteredCustomers;
```

- **Comentarios redundantes,** es decir, comentarios que son más largos de leer que el código que están tratando de explicar, o que simplemente son innecesarios, porque el código se explica por sí mismo. Por ejemplo, el siguiente comentario es redundante, ya que el código que explica es bastante comprensible:

```
// Comprobamos si la edad es mayor de 18 años, y si es así, // imprimimos un mensaje
diciendo "Tienes la edad suficiente"
Si (edad> 18 )
        Console.WriteLine ( "Eres lo suficientemente mayor" );
```

- **Comentarios sin contexto,** es decir, comentarios que no van seguidos del código correspondiente. Por ejemplo, el siguiente comentario no se completa con el código apropiado. Falta un código. En el comentario decimos que, si el usuario no es un adulto, se cerrará la sesión de la aplicación, pero no hay un código para cerrar la sesión del usuario debajo del comentario. Tal vez este cierre de sesión se realice en otra parte del código, pero luego este comentario debería colocarse allí.

```
Si (edad> 18 )
{
        Console.WriteLine ( "Eres lo suficientemente mayor" );
} más {
        // Si el usuario no es un adulto, se desconecta
}
```

- **Comentarios obligatorios:** algunas personas piensan que cada variable, por ejemplo, debe tener un comentario que explique su propósito. Pero esa no es una buena decisión, ya que podemos evitar la mayoría de estos comentarios usando nombres de variables apropiados.

- **Comentarios de la revista:** a veces, se coloca un registro de edición al principio de un archivo fuente. Contiene todos los cambios realizados en el código, incluida la fecha y el motivo del cambio. Pero hoy en día, podemos usar aplicaciones de control de versiones, como GitHub, para mantener este registro fuera del código.

- **Marcadores de posición y divisores de código:** es muy habitual hacer comentarios para encontrar rápidamente un lugar en el código, o para separar algunos bloques de código que son bastante largos. No se recomiendan ambos tipos de comentarios si el código está formateado correctamente.

```
// ================== VARIABLES ====================
En t años;
cuerda nombre;
...
// ================== PRINCIPAL ========================
vacío estático público Principal () {

        ...
        ////// RESULTADO FINAL
}
```

- **Comentarios de corchete de cierre,** que se colocan en cada llave de cierre para explicar qué elemento es esta llave de cierre. Estos comentarios se pueden evitar, ya que la mayoría de los IDE actuales resaltan cada par de llaves cuando hacemos clic en ellas, para que podamos hacer coincidir cada par automáticamente.

```
mientras (n> 10 )
{
        Si (n> 5 )
        {
                ...
        } // Si
} // mientras
```

- **Advertencias** que se utilizan cuando tenemos algún código que puede causar problemas en determinadas situaciones, porque es necesario revisarlo. Es muy habitual encontrar algunos bloques de código completamente comentados, y un mensaje de advertencia explicando el problema. Estos comentarios deben convertirse en comentarios "TODO", para advertir al programador que este código debe revisarse en el futuro, en lugar de simplemente eliminar los comentarios.

Ejercicios propuestos:

1.9.3.1. Este programa le pide al usuario que introduzca tres números y obtiene el promedio de ellos. Discuta en clase qué partes del código no están limpias o podrían mejorarse, con respecto a los nombres de variables y comentarios.

```
utilizando Sistema;

clase pública AverageNumbers
{
        vacío estático público Principal () {

                // Variables para almacenar los tres números y el promedio
                En t n1, n2, n3;
                En t Resultado;

                // Le pedimos al usuario que ingrese tres números
                Console.WriteLine ( "Introduce tres números:" );
                n1 = Convert.ToInt32 (Console.ReadLine ());
                n2 = Convert.ToInt32 (Console.ReadLine ());
                n3 = Convert.ToInt32 (Console.ReadLine ());
                // El resultado es el promedio de estos números / * Podríamos haber
                usado un número flotante en su lugar,
                        pero decidimos mantener este programa lo más simple
                        posible * /
                Resultado = (n1 + n2 + n3) / 3 ;
                Console.WriteLine ( "El promedio es {0}" , Resultado);
        }
}
```

1.9.3.2. Este programa le pide al usuario que ingrese números hasta que ingrese un número negativo, o un total de 10 números. Cópielo en su IDE e intente mejorarlo con nombres de variables y comentarios adecuados.

```
/ *
  * (C) IES San Vicente 2016
  * /
utilizando Sistema;

clase pública Ingrese Números
{
        vacío estático público Principal () {

                // Números ingresados   por el usuario
                En t numero 1;
                // Recuento de números
                En t numero2 = 0 ;

                hacer
                {
                        // Le pedimos al usuario que escriba un número
                        Console.WriteLine ( "Escriba un número:" );
                        número1 = Convert.ToInt32 (Console.ReadLine ());
                        número2 ++;
                } mientras (número2 < 10 && número1> = 0 ); // haz..mientras tanto

                Si (número1 < 0 ) Console.WriteLine ( "Terminando. Número negativo" );

                más
                        Console.WriteLine ( "Terminando. 10 números" );
        }
}
```

1.9.3.3. Cree un programa llamado RectangleDraw que le pida al usuario que introduzca una base y una altura de rectángulo, y luego imprima un rectángulo de las dimensiones dadas. Por ejemplo, si el usuario establece una base de 5 y una altura de 3, el programa debería imprimir este rectángulo (lleno de '*'):

```
* * * * *
* * * * *
* * * * *
```

Implemente este programa de acuerdo con las reglas explicadas en este documento, en cuanto a nombres de variables y comentarios.

1.9.4. Espaciado y formato de código

El formato y el espaciado del código apropiado le dicen al lector que el programador ha prestado atención a cada detalle del programa. Sin embargo, cuando encontramos un montón de líneas de código con sangría incorrecta y / o

espaciados, podemos pensar que la misma falta de atención puede estar presente en otros aspectos del código.

1.9.4.1. Espaciado vertical

Veamos algunas reglas simples para formatear y espaciar su código verticalmente:

- Como cada grupo de líneas representa una tarea, estos grupos deben separarse entre sí con una línea en blanco. En un programa de C #, por ejemplo, tendríamos algo como esto (preste atención a dónde se agregan las líneas en blanco):

```
utilizando Sistema;

clase pública Programa
{
        vacío estático público Principal () {

                En t personaje;
                cuerda personName;

                Console.WriteLine ( "Dime tu nombre:" ); personName =
                Console.ReadLine ();

                Console.WriteLine ( "Dime tu edad:" ); personAge = Convert.ToInt32
                (Console.ReadLine ());

                Si (personAge> 18 ) Console.WriteLine ( "Eres un adulto, {0}" ,
                        personName);

        }
}
```

- Los conceptos que están estrechamente relacionados deben colocarse juntos verticalmente. Por ejemplo, si declaramos dos variables para almacenar el nombre y la edad de una persona, entonces deberíamos colocar estas declaraciones una tras otra, sin separaciones. Esto quiere decir que no debemos agregar ningún comentario que rompa la unión:

```
cuerda personName;
/ *
  * ¡Este comentario no debe escribirse aquí!
  * /
En t personaje;
```

- Las llaves de apertura se colocan al final de las líneas que las necesitan (típicas en lenguajes de programación como Java o Javascript) o al principio de la siguiente línea, con la misma sangría que la línea anterior (típica en lenguajes de programación como C o C#). En este último caso, pueden actuar como líneas en blanco de separación entre bloques.

```
// Estilo Java (la llave de apertura NO se considera una línea en blanco)
Si (condición) {
        . . .
}

// Estilo C # (la llave de apertura puede considerarse una línea en blanco)
vacío estático público Principal () {

        Si (condición)
        {
                . . .
        }
. . .
```

Con respecto a la apertura de aparatos ortopédicos, puede decidir cuál de estos patrones desea aplicar, pero debe:

- Aplicar siempre el mismo patrón
- Utilice el mismo patrón que todas las personas de su equipo

1.9.4.2. Formato horizontal

Con respecto al espaciado horizontal o al formato, también hay algunas reglas simples que podemos seguir.

- Una línea de código debe ser corta (tal vez 80 o 100 caracteres de longitud, como máximo). Algunos IDE muestran una línea vertical (típicamente roja) que establece el límite "ideal" para la longitud de cada línea. Si va a ser más largo, debemos cortarlo y dividir el código en varias líneas. También puede aplicar otras reglas para determinar el ancho máximo de línea: nunca debería tener que desplazarse hacia la derecha para ver su código, y debería ser imprimible con la misma apariencia en una página vertical.

```
Si ((persona Edad> 18 && personAge <= sesenta y cinco ) || (personName == "Juan" ) ||
        (personName == "María" ))
{
        . . .
}
```

- El espaciado horizontal nos ayuda a asociar cosas que están relacionadas y a disociar las que no lo están. Por ejemplo, los operadores deben separarse con un espacio en blanco de los elementos que están operando:

```
En t promedio = (número1 + número2) / 2 ;
```

- No alinee los nombres de las variables verticalmente. Era muy típico en los lenguajes de programación antiguos, como ensamblador, pero no tiene sentido en los lenguajes de programación modernos, donde hay muchas

tipos de datos. Si hace esto, puede tender a leer los nombres de las variables sin prestar atención a sus tipos de datos:

```
StringBuilder texto largo;
En t tamano del texto;
cuerda textToFindAndReplace;
```

- La sangría es importante, ya que establece una jerarquía. Hay elementos que pertenecen a todo el archivo fuente y otros que forman parte de un bloque de hormigón. La sangría nos ayuda a determinar el alcance de un grupo de instrucciones. De este modo:
  - El nombre de la clase no tiene sangría
  - Las funciones u otros elementos dentro de una clase tienen sangría en un nivel La implementación de estas funciones tiene sangría en dos niveles
  - Implementaciones de bloques dentro del código de función (código de *Si* o *mientras* cláusulas, por ejemplo) tienen sangría en tres niveles
  - … Etc.

```
clase pública Mi clase
{
        vacío estático público Principal () {

                Console.WriteLine ( "Hola" );
                Si (…)
                {
                        Console.WriteLine ( "Dentro de un si" );
                }
        }
}
```

1.9.5. Funciones

Las funciones son una forma de organizar el código en nuestros programas. Nos ayudan a dividir el código en diferentes módulos y a llamar / usar cada módulo cuando lo necesitemos. Sin embargo, si están incorrectamente escritos, pueden ser parte del problema en lugar de ser parte de la solución. En esta sección aprenderemos cómo deben organizarse las funciones y algunas reglas básicas para escribir un buen código con ellas.

1.9.5.1. El tamaño importa

La regla (quizás) más importante que debe cumplir una función es que debe ser pequeña. Pero el adjetivo *pequeño* es bastante difuso ... ¿cuántas líneas se consideran pequeñas? Bueno, hace algunos años los expertos decían que una función no debería tener más de una página. Pero los tipos de fuente solo permitían algunas líneas por página, y

ahora podríamos escribir hasta 100 líneas en una sola página. ¿Significa esto que podríamos tener funciones de 100 líneas? La respuesta es NO ... una función debe contener aproximadamente 20 líneas de código.

Para hacer esto, debemos dividir nuestro código en funciones y realizar llamadas entre ellas correctamente. Esto provocaría que el nivel de sangría de una función no sea mayor que uno o dos (estructuras de control como `Si` , `mientras` o similar, y una función llamada dentro de ellos)

1.9.5.2. Tareas individuales

¿Cuál es la mejor forma de dividir nuestro código para que cada función no tenga más de 20 líneas? Debemos dividir el programa en tareas, y estas tareas en subtareas, si es necesario, y así sucesivamente. Cada función debe implementar una tarea de un nivel.

Por ejemplo, considere el siguiente programa: validamos el inicio de sesión de un usuario y luego imprimimos su perfil en la pantalla. El programa principal estaría compuesto por dos tareas de alto nivel:

1. Validar usuario
2. Imprimir perfil

Entonces debemos definir una función para cada una de estas tareas. Luego, continuamos dividiendo cada función en subtareas. Por ejemplo, para la primera función:

1. Validar usuario
    1. Imprima "Inicio de sesión de usuario:" en la pantalla
    2. Obtener inicio de sesión de usuario desde el teclado
    3. Imprima "Contraseña de usuario:" en la pantalla
    4. Obtener la contraseña de usuario del teclado
    5. Verificar usuario en la base de datos
    6. Si es correcto, finalice esta función
    7. Si no es correcto, continúe con el paso 1.1.

Casi todas estas subtareas son lo suficientemente simples como para hacerlas en solo una o dos líneas de código, por lo que no necesitamos más niveles ... aparte de la subtarea 1.5, que necesitaría un nuevo nivel:

5. Verificar usuario en la base de datos
    1. Conectarse a la base de datos
    2. Busque el nombre de usuario y la contraseña introducidos
    3. Establecer si el usuario existe o no

Si traducimos este esquema en código, obtendríamos algo como esto:

```
vacío estático público Principal () {

        ValidateUser ();
        PrintUserProfile ();
}

vacío estático público ValidateUser () {

        bool resultado = falso ;
        hacer
        {
                Console.WriteLine ( "Inicio de sesión de usuario:" );
                cuerda login = Console.ReadLine (); Console.WriteLine ( "Contraseña
                de usuario:" );
                cuerda contraseña = Console.ReadLine ();
                resultado = CheckUserInDatabase (inicio de sesión, contraseña); } mientras (!resultado);

}

vacío estático público CheckUserInDatabase ( cuerda iniciar sesión, cuerda contraseña ) {

        . . .
}

. . .
```

1.9.5.3. Nombres de funciones

De la misma forma que las variables deben tener nombres apropiados, las funciones también deben tener nombres que expliquen claramente lo que hacen. Intente aplicar el mismo patrón a funciones similares e intente usar un verbo en el nombre, ya que las funciones HACEN algo. Por ejemplo, si tenemos un montón de funciones para validar algunos datos ingresados por el usuario, nosotros podría llamada estas funciones `validateNumber` , `validatePostalCode` , `validateTelephoneNumber` ... En todas estas funciones usamos un verbo (*validar),* y el mismo patrón *validateAAA, validateBBB ...).* Sería un error llamar a una función `validateNumber` , y use `checkPostalCode` para el segundo, por ejemplo.

En cuanto al uso de mayúsculas y minúsculas, las funciones siguen las mismas reglas que las variables explicadas en apartados anteriores. En Java, por ejemplo, todas las funciones (aparte de los constructores) comienzan con minúsculas, y cada palabra nueva dentro del nombre comienza con una letra mayúscula (*el caso de Carmel):*

```
vacío público myFunction () {…}
```

En C #, esta regla se aplica solo a funciones no públicas. Si una función es pública, comienza con una letra mayúscula (*caso pascal):*

```
vacío público MyFunction () {…}
En t otra función () {…}
```

1.9.5.4. Funciones y espaciado

En cuanto a las funciones, debemos prestar atención a estas reglas explicadas anteriormente cuando hablamos de cómo formatear y espaciar nuestro código. Para empezar, debe haber una línea en blanco entre cada par de funciones:

```
vacío público Función1 ()
{
        . . .
}

público int Función2 ()
{
        . . .
}
```

Las funciones que están estrechamente relacionadas deben colocarse cerradas entre sí, de modo que no necesitemos explorar todo el archivo fuente para encontrar una de ellas en la otra. Esta afinidad puede derivarse de la dependencia entre funciones o de una funcionalidad similar.

- Con respecto a la dependencia, si una función A llama a otra B, entonces A debe colocarse antes de B. Con respecto a la funcionalidad,
- considere un montón de funciones con el mismo prefijo o sufijo (`addNames` , `addCustomers` ...). Estas funciones deben estar próximas entre ellas.

1.9.5.5. Otras características

Hay algunas otras características que son deseables en funciones.

- **Evite los efectos secundarios:** el código de una función no debe modificar nada fuera de esta función directamente, ni hacer nada que no esté implícitamente declarado en su nombre. Por ejemplo, si tenemos una función llamada `validateUser` , no debe almacenar nada en un archivo.
- **Siga la regla de programación estructurada:** según la programación estructurada, cada bloque de código (por ejemplo, una función) debe tener una sola entrada y una salida. Esto significa que cada función debe tener solo una `regreso` declaración, y cada bucle un punto final. Entonces debemos evitar `romper` o `Seguir` instrucciones dentro de los bucles.

> **Ejercicios propuestos:**

> **1.9.5.1.** El siguiente ejemplo le pide al usuario que diga cuántos nombres va a escribir. Luego, inicializa una matriz del tamaño especificado y almacena los nombres en ella. Finalmente, ordena los nombres

alfabéticamente y los imprime. Eche un vistazo al código e intente mejorarlo en términos de limpieza del código.

```
utilizando Sistema;

clase pública SortNames
{
        hoyo estatico PrintNames ( cuerda [] nombres ) {

                Console.WriteLine ( "Lista en orden alfabético:" );
                para cada ( cuerda nombre en nombres) {

                        Console.WriteLine (nombre);
                }
        }
        vacío estático público Principal ()
        {
                En t totalNames;
                cuerda [] nombres;
                cuerda aux;
                Console.WriteLine ( "Ingrese el número total de nombres:" ); totalNames =
                Convert.ToInt32 (Console.ReadLine ());
                nombres = nueva cadena [totalNames]; Console.WriteLine ( "Ingrese los {0} nombres:" ,
                totalNames);
                para ( En t yo = 0 ; i <totalNames; i ++) {

                        nombres [i] = Console.ReadLine ();
                }
                para ( En t yo = 0 ; i <totalNames - 1 ; i ++)
                        para ( En t j = i + 1 ; j <totalNames; j ++)
                                Si (nombres [i] .CompareTo (nombres [j])> 0 ) {

                                        aux = nombres [i];
                                        nombres [i] = nombres [j];
                                        nombres [j] = aux;
                                }
                PrintNames (nombres);
        }
}
```

1.9.5.2. El siguiente código muestra algunas funciones que devuelven algunos valores y un *Principal* programa que los usa. Intente mejorar el código en términos de limpieza.

```
utilizando Sistema;

// Devuelve si un número es par o impar
público estático booleano incluso ( En t número ) {

        Si (número% 2 == 0 )
                regreso cierto ;
        más
                regreso falso ;
}

// Busca una palabra en una matriz de cadenas y devuelve la // posición en la que se
encuentra.
// Si no se encuentra la palabra, devuelve -1
public static int buscar ( cuerda [] matriz, cuerda palabra ) {

        para ( En t yo = 0 ; i <array.Length; i ++) {

                Si (matriz [i] == palabra)
                        regreso yo;
        }
        regreso -1 ;
}

vacío estático público Principal () {

        Console.WriteLine ( "Ingrese un numero" );
        En t número = Convert.ToInt32 (Console.ReadLine ());
        bool even = isEven (número);
        Si (incluso)
                Console.WriteLine ( "El número es par" );
        más
                Console.WriteLine ( "El número es impar" );
        cuerda [] palabras = { "Hola" , "adiós" , "uno" , "dos" , "Tres" , "cuatro" , "rojo" , "grito
        Console.WriteLine ( "Ingresa una palabra" );
        cuerda palabra = Console.ReadLine ();
        En t resultado = búsqueda (palabras, palabra);
        Si (resultado> = 0 ) Console.WriteLine ( "Palabra encontrada en" + resultado);

        más
                Console.WriteLine ( "Palabra no encontrada" );
}
```

1.9.6. Otros elementos a considerar

Para terminar con esta unidad, tengamos una descripción general rápida de algunos otros elementos que deben escribirse con cuidado, como las clases y el manejo de errores.

1.9.6.1. Usando clases

Aunque es posible que no sepa qué es una clase en este momento, es posible que deba conocer algunos conceptos útiles para poder escribirlos correctamente:

- **Estructura estándar de una clase:** los estándares dicen que una clase debe comenzar con una lista de variables o atributos de instancia, y luego los constructores y métodos. Con respecto a la lista de variables, normalmente colocamos las constantes antes y luego las variables. Por ejemplo, este podría ser el código de una clase llamada *Persona* que gestiona el nombre y la edad de las personas:

```
clase Persona
{
        cuerda nombre;
        En t años;

        público Persona ( cuerda un nombre, En t Una época ) {

                nombre = unNombre;
                edad = anAge;
        }

        vacío público Di hola ()
        {
                Console.WriteLine ( "Hola {0}, tienes {1} años" , nombre Edad);

        }
}
```

- **Encapsulamiento:** También se recomienda mantener nuestros atributos privados (o protegidos, si vamos a heredar de esa clase). Las clases y los objetos ocultan sus datos detrás de abstracciones y exponen funciones que operan en esos datos. Se llama *encapsulación.*

- **Cohesión y clases pequeñas:** cuantas más variables de instancia manipule un método, más cohesivo será ese método con su clase. Una clase donde cada variable o atributo es manipulado por cada método es máximamente cohesiva. Nuestro objetivo es tener una alta cohesión, ya que eso significaría que los métodos y las variables son codependientes y actúan como un todo lógico. Además, si tratamos de mantener la cohesión (es decir, tener atributos utilizados por (casi) todos los métodos), probablemente obtendremos clases pequeñas, ya que es difícil tener una clase con una gran cantidad de variables o métodos que sean co- dependiente.

- **Denominación de clases:** una clase suele ser un sustantivo en la especificación de requisitos, por lo que el nombre de una clase debe ser normalmente un sustantivo. Para ser más precisos, debería ser un sustantivo singular, como Person, Shape, Animal, StringBuffer ... Además, el nombre de la clase debería dar suficiente información sobre el objetivo de esta clase y las responsabilidades que tiene. Por ejemplo, una clase llamada String puede hacernos pensar que manejará todos los

métodos necesarios para tratar con textos, como tomar subcadenas, buscar un texto parcial, reemplazar texto, etc.

1.9.6.2. Manejo de errores

El manejo de errores es importante en nuestros programas porque el usuario puede cometer errores al introducir datos, o algunos dispositivos pueden fallar cuando, por ejemplo, intentamos leer o escribir datos desde / hacia un archivo. Sin embargo, si no utilizamos correctamente el manejo de errores, puede dañar nuestro código y ocultar su estructura lógica. En esta sección veremos algunas técnicas que nos ayudarán a escribir código limpio y robusto.

1.9.6.2.1. Excepciones y códigos de error

En los lenguajes de programación anteriores, las excepciones no existían, y las únicas formas de detectar un error eran crear una variable booleana para almacenar si alguna operación era correcta o devolver un número entero que representaba un código de error (dependiendo de este valor, podríamos averiguar el error en sí).

Sin embargo, siempre que un lenguaje de programación permita el manejo de excepciones, es mejor usar excepciones en lugar de códigos de error. Siempre que un bloque de código pueda causar un error, debe intentar detectarlo (o lanzar una excepción a otra función). Pero hay algunos aspectos que debes tener en cuenta al tratar con excepciones:

- **Proporcione contexto con excepciones:** Cuando recibimos un mensaje de error de un programa en ejecución, es importante que este mensaje sea comprensible y podamos averiguar qué salió mal. Por lo tanto, cada vez que lanzamos una excepción, debemos proporcionar información sobre la fuente del error y la ubicación. Por ejemplo, podemos mencionar la operación que falló y el tipo de falla: "Error al leer el archivo. El archivo 'data.txt' no se encuentra en / home / user".

- **Escribir *trata de atraparlo* bloques primero:** puede ser una buena idea escribir tu `trata de atraparlo` bloquear antes de comenzar con las líneas de código que estarán en él. El bloque try define un alcance, como `Si` o `mientras` hacer, y `cuando` lo agregamos, debemos ser conscientes de que la ejecución puede ser abortada en cualquier momento, y debemos dejar el programa en un estado consistente en el bloque catch. Además, iniciar una función que puede causar una excepción con un bloque try..catch le dice al usuario de ese código que se puede esperar una excepción, sin importar cuál o cuándo.

- **Diferenciar los tipos de excepción si es necesario:** en muchos programas, puede ser útil mostrar diferentes mensajes de error para diferentes excepciones. Por ejemplo, un mensaje de error si no se pudo encontrar un archivo, y otro mensaje diferente si no pudimos leer un elemento determinado de él.

1.9.6.2.2. Devolviendo nulo

Hay muchos métodos en muchas API oficiales que regresan `nulo` cuando algo `sale mal`. Por ejemplo, existen métodos que, cuando leemos datos de un archivo, devuelven `nulo` cuando no queda más información para leer. Regresando `nulo` `puede ser` una fuente de nuevos errores, ya que obliga al programador a verificar si el valor `de retorno` de la función es nulo:

```
Valor de cadena = myObject.methodThatCanReturnNull ();

Si (valor! = nulo ) {

        // Resto del código que importa
}
```

Tenga en cuenta que, si hacemos esto, estamos sangrando el código un nivel adicional a la derecha, indicando que este no es el primer nivel de código dentro de la función ... pero lo es. Además, hay algunas otras líneas que no están marcadas. ¿Qué pasa si myObject es nulo? De hecho, no deberíamos comprobar el nulo valor en cualquier parte de nuestro código.

Entonces, ¿qué hacer en lugar de devolver un valor nulo? Tenemos algunas opciones:

- Lanzar una excepción con un mensaje apropiado
- Devuelve un valor especial que representa un error. Por ejemplo, un índice negativo cuando una palabra no se encuentra en una matriz de cadenas.

De la misma manera, no es una buena idea pasar un valor nulo como argumento a cualquier método o función, porque el método podría arrojar un Excepción de puntero nulo sin querer.

> **Ejercicios propuestos:**

> **1.9.6.1.** El siguiente ejercicio pide al usuario que introduzca cuántas edades va a escribir y luego calcula el promedio de estas edades. Verifique el código en términos de limpieza y robustez.

```
utilizando Sistema;

clase pública Edad Promedio
{
        vacío estático público Principal () {

                En t numberOfAges;
                En t [] siglos;

                Console.WriteLine ( "Ingrese el número de edades:" ); numberOfAges =
                Convert.ToInt32 (Console.ReadLine ());
                edades = nuevo int [numberOfAges];

                para ( En t yo = 0 ; i <numberOfAges; i ++) {

                        edades [i] = Convert.ToInt32 (Console.ReadLine ());
                }
                En t total = 0 ;
                para ( En t yo = 0 ; i <numberOfAges; i ++)
                        total + = edades [i];

                En t average = total / numberOfAges; Console.WriteLine ( "El promedio
                es" + promedio);
        }
}
```