ENTORNOS DE DESARROLLO

ciclos formativos de grado superior de
desarrollo de aplicaciones multiplataforma

CAPÍTULO 4

Carlos Casado Iglesias

Ra-Ma®

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

"La refactorización consiste en realizar una transformación al software preservando su comportamiento, modificando su estructura interna para mejorarlo" (Opdyke, 1992).

En múltiples ocasiones durante el transcurso de un proyecto de grandes proporciones o de larga duración es frecuente encontrarse con que necesitamos revaluar y modificar código creado anteriormente, o ponerse a trabajar con un proyecto de otra persona, para lo cual antes se deberá estudiar y comprender. A pesar de los comentarios o la documentación del proyecto en cuestión, la refactorización (informalmente llamada limpieza de código) ayuda a tener un código que es más sencillo de comprender, más compacto, más limpio y, por supuesto, más fácil de modificar.

Refactorizar no cambia la funcionalidad del código ni el comportamiento del programa, el programa deberá comportarse de la misma forma antes y después de efectuar las refactorizaciones.

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

Las refactorizaciones son pequeños cambios que hacen que el código sea más visible, flexible, entendible y modificable.

No existe ninguna etapa de desarrollo especifica para la refactorización, se puede **refactorizar** siempre que se quiera ya que la funcionalidad (si lo hemos hecho bien) no va a sufrir cambio alguno.

Aunque no exista ninguna etapa para la refactorización, debería ser una tarea recurrente a la hora de codificar o mantener una aplicación. El modo más eficiente de tener un código **refactorizado** es intercalar la creación de código nuevo con la refactorización de código.

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

Las **tabulaciones** e **indentados** son también un excelente recurso para obtener una mayor visibilidad del código, con lo que conseguir que el código sea mas entendible y fácilmente modificable.

Aunque la tabulación no sea una técnica o una práctica propia de la refactorización en sí misma, ya que no se modifica código, cumple con al menos parte de los objetivos de la refactorización.

Visual Studio nos ofrece una herramienta que permite formatear el código de manera automática utilizando la entrada de menú **Editar > Avanzadas > Dar formato al documento**. Recuerde que puede personalizar el formato del código mediante el menú **Herramientas > Opciones > Editor de texto** de Visual Studio.

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Para refactorizar nuestro código, podemos encontrar útil disponer de una serie de reglas o casos de uso para aplicar en nuestro código, eso es justamente lo que hacen los patrones de refactorización. Aportan una base de modificaciones en el código para sustituir elementos de la manera más apropiada dependiendo de lo que quieras refactorizar y más importante aún, de como lo quieras refactorizar.

También tendríamos que tener en cuenta que no siempre se pueden aplicar estos patrones al código sobre el que estamos trabajando, depende en gran medida del lenguaje que estemos utilizando. En este caso concreto estamos hablando de patrones de refactorización para ser aplicados a lenguajes del paradigma de programación orientada a objetos.

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Extraer Método

✓Tienes un fragmento de código que puede agruparse.

✓Conviertes el fragmento en un método cuyo nombre explique el propósito del método.

```
void imprimirTodo() {
     imprimirBanner();
     //detalles de impresión
     Console.WriteLine (“nombre: “ + _nombre);
     Console.WriteLine(“cantidad “ + getCargoPendiente());
}
```

Refactorizamos

```
void ImprimirTodo() {
     imprimirBanner();
     imprimirDetalles(getCargoPendiente());
}
void imprimirDetalles(double cargoPendiente) {
     Console.WriteLine (“nombre: “ + _nombre);
     Console.WriteLine(“cantidad “ + cargoPendiente);
}
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Separar Variables Temporales

✓Tienes una variable temporal que usas más de una vez, pero no es una variable de bucle ni una variable temporal de colección.

✓Creamos una variable temporal diferente para cada asignación.

```
double temp = 2 * (_alto + _ancho);
Console.WriteLine (temp);
temp = _alto * _ancho;
Console.WriteLine (temp);

Refactorizamos

final double perimetro = 2 * (_alto + _ancho);
Console.WriteLine (perimeter);
final double area = _alto * _ancho;
Console.WriteLine (area);
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Eliminar asignaciones a parámetros

✓Un parámetro es usado para recibir una asignación.

✓Usamos una variable temporal en su lugar.

```
int descuento (int entradaValor, int cantidad, int año) {
    if (entradaValor > 50)
        entradaValor -= 2;
    […]
}
```

Refactorizamos

```
int descuento (int entradaValor, int cantidad, int año) {
    int resultado = entradaValor;
    if (entradaValor > 50)
        resultado -= 2;
    […]
}
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Mover Método

✓Un método es, o será, usado por más características de otra clase que en aquella donde está definido.

✓Crearemos un nuevo método con un cuerpo similar en la clase que se use más. Convertiremos el cuerpo del método antiguo en una delegación simple o lo removeremos por completo.

```
class Proyecto {
      Persona[] participantes;
}
class Persona {
int id;
boolean participante(Proyecto p) {
      for(int i=0; i<p.participantes.length; i++)
            if (p.participantes[i].id == id)
                  return(true);
      return(false);
      }
[…] if (x.participante(p)) […]
```

Refactorizamos

```
class Proyecto {
      Persona[] participantes;
      boolean participante(Persona x) {
            for(int i=0; i<participantes.length; i++)
            if (participantes[i].id == x.id)
                  return(true);
            return(false);
      }
}
class Persona {
      int id;
}
[…] if (p.participante(x)) […]
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Consolidar fragmentos duplicados en condicionales

✓El mismo fragmento de código está en todas las ramas de una expresión condicional.

✓Sacamos dicho fragmento fuera de la expresión.

```
if (esAcuerdoEspecial()) {
    total = precio * 0.95;
    enviar();
}else {
    total = precio * 0.98;
    enviar();
}
```

Refactorizamos

```
if (esAcuerdoEspecial())
    total = precio * 0.95;
else
    total = precio * 0.98;
enviar();
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Descomponer un Condicional

✓Tenemos una complicada declaración en el condicional.

✓Extraemos métodos de la condición y del cuerpo del condicional.

```
if (fecha.antes (EMPIEZA_VERANO) || fecha.despues(FIN_VERANO))
     cargo = cantidad * _tasaInvierno + _cargoServicioInvierno;
else
     cargo = cantidad * _tasaVerano;
```

Refactorizamos

```
if (noEsVerano(fecha))
     cargo = cargoInvierno(cantidad);
else
     charge = cargoVerano (cantidad);
double cargoInvierno(int cantidad) {
     return cantidad * _tasaInvierno + _cargoServicioInvierno;
}
double cargoVerano(int cantidad) {
     return cantidad * _tasaVerano;
}
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Consolidar Expresiones Condicionales

✓Tenemos una secuencia de condicionales con el mismo resultado.

✓Los combinamos en una sola expresión y lo extraemos.

```
double cuantiaPorDiscapacidad() {
     if (_antiguedad < 2)
          return 0;
     if (_mesesDiscapacitado > 12)
          return 0;
     if (_esTiempoParcial)
          return 0;
     // calculamos la cantidad por discapacidad
}
```

Refactorizamos

```
double cuantiaPorDiscapacidad () {
     if (esNoElegibleParaDiscapacidad())
          return 0;
     // calculamos la cantidad por discapacidad
}
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

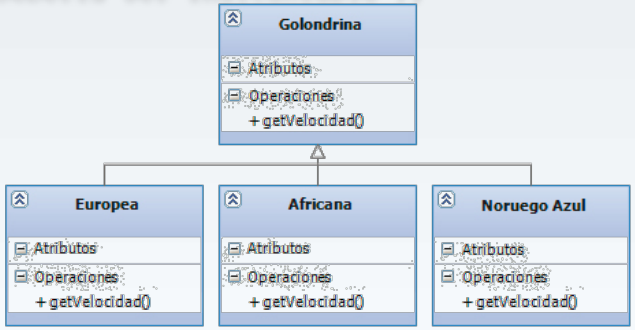
Reemplazar Condicional por Polimorfismo

✓Tenemos un condicional que elige diferentes comportamientos dependiendo del tipo de un objeto.

✓Movemos cada caso del condicional en un método sobrecargado en una subclase. Hacemos el método original abstracto.

```
double getVelocidad(){
      switch (_tipo){
            case EUROPEA:
                  return getBVelocidadBase();
            case AFRICANA:
                  return getVelocidadBase() - getFactorCarga() * _numeroCocos;
            case NORUEGO_AZUL:
                  return (_esMoteado) ? 0 : getVelocidadBase(_voltaje);
      }
      throw new RuntimeException("Debería ser inalcanzable");
}
```

Refactorizamos



OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Reemplazar Número Mágico con Constante Simbólica

✓Tenemos un literal con un significado particular.

✓Creamos una constante, la nombramos significativamente y la sustituimos por el literal.

```
double energiaPotencial(double masa, double altura) {
    return masa * altura * 9.81;
}

Refactorizamos

double energiaPotencial(double masa, double altura) {
    return masa * CONSTANTE_GRAVITACIONAL * altura;
}
static final double CONSTANTE_GRAVITACIONAL = 9.81;
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Reemplazar Número Mágico con Método Constante

✓Tenemos un literal con un significado particular.

✓Creamos un método que nos devuelve el literal, lo nombramos significativamente y lo sustituimos por el literal.

```
double energiaPotencial(double masa, double altura) {
    return masa * altura * 9.81;
}
```

Refactorizamos

```
double energiaPotencial(double masa, double altura) {
    return masa * constanteGravitacional() * altura;
}
public static double constanteGravitacional(){
    return 9.81;
}
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

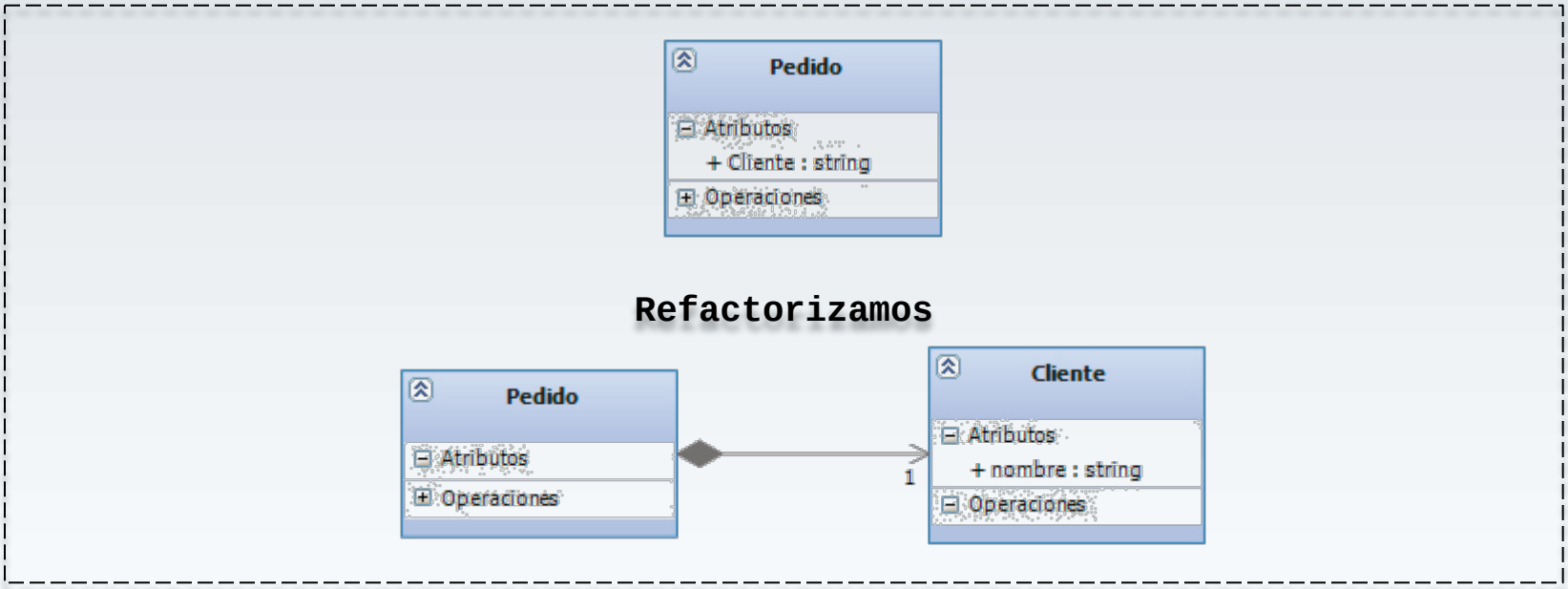
1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Reemplazar Datos y Valores por Objetos

✓Tenemos un atributo que necesita información o comportamiento adicional.

✓Convertimos el atributo en un objeto.



OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Reemplazar Array con Objeto

✓Tenemos un array en el que ciertos elementos tienen un significado diferente.

✓Reemplazamos el array con un objeto que tenga un atributo para cada elemento.

```
String[] fila = new String[3];
fila[0] = “San Martín de la Arena C.D.”;
fila[1] = “15”;

Refactorizamos

Rendimiento fila = new Rendimiento();
fila.setNombre(“San Martín de la Arena C.D.”);
fila.setGanados(“1337”);
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Encapsular Atributo

✓Tenemos un atributo público.

✓Lo convertimos a privado y le creamos métodos de acceso.

```
public String _nombre;

Refactorizamos

private String _nombre;
public String getNombre() {return _nombre;}
public void setNombre(String arg) {_nombre = arg;}
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Encapsular Atributo como Propiedad

✓Tenemos un atributo público.

✓Lo convertimos en una propiedad del objeto.

```
public String _nombre;

Refactorizamos

public virtual string Nombre {get; set;}
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

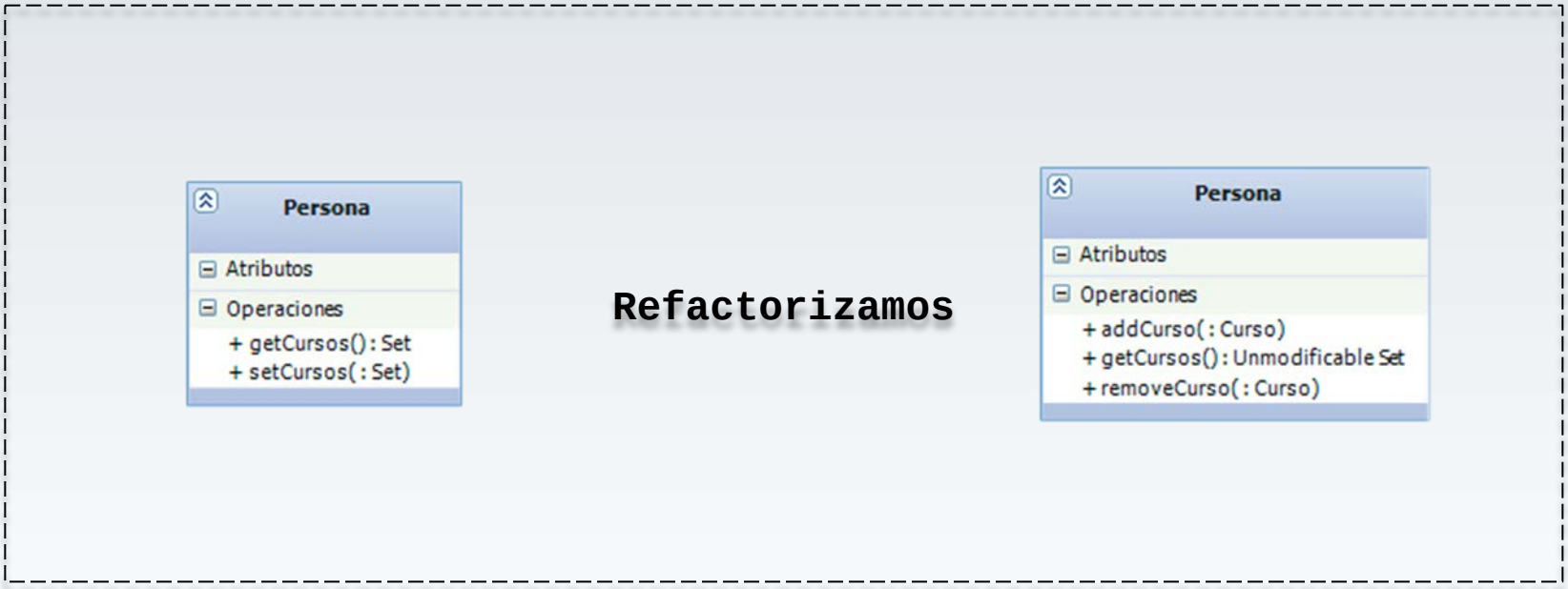
1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Encapsular Colección

✓Un método devuelve una colección.

✓Hacemos que devuelva una colección de solo lectura y le facilitamos métodos de adición y eliminación de elementos.



OPTIMIZACIÓN Y DOCUMENTACIÓN

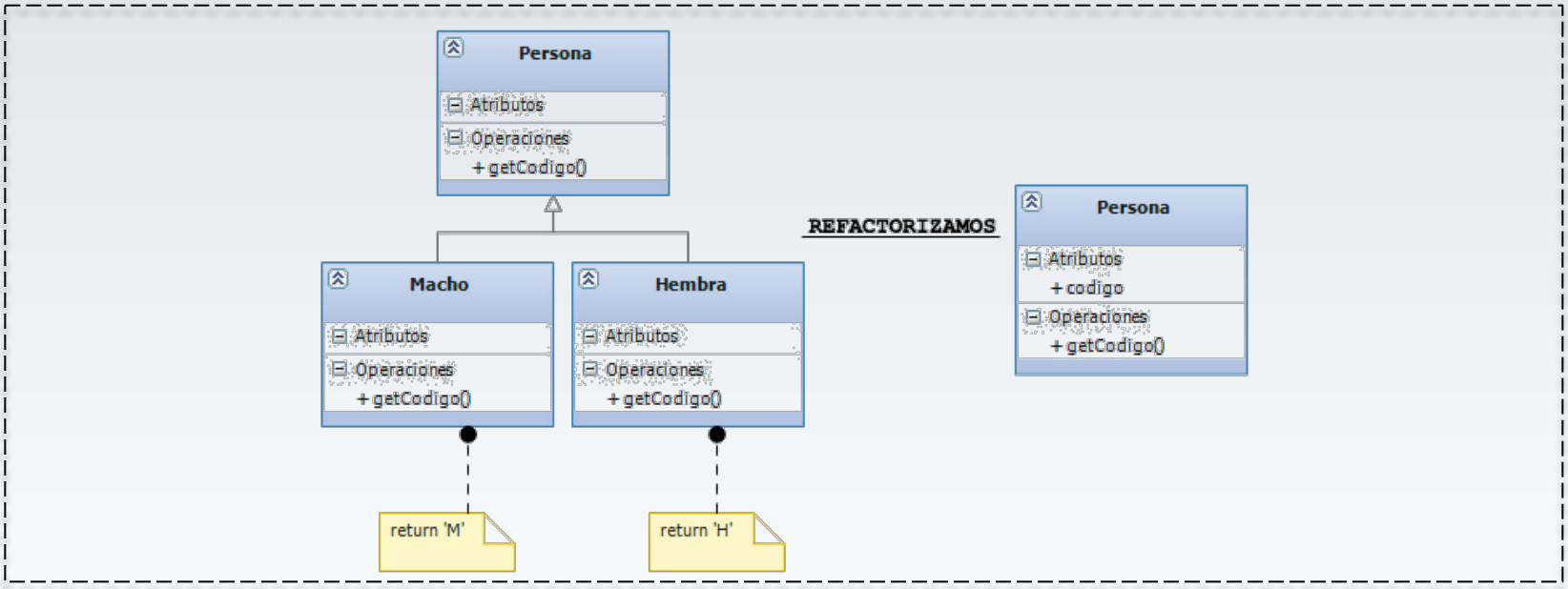
1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Reemplazar SubClases por Atributos

✓Tenemos subclases que solo varían en métodos que devuelven información constante.

✓Cambiamos los métodos por atributos de la superclase y eliminamos las subclases.



OPTIMIZACIÓN Y DOCUMENTACIÓN

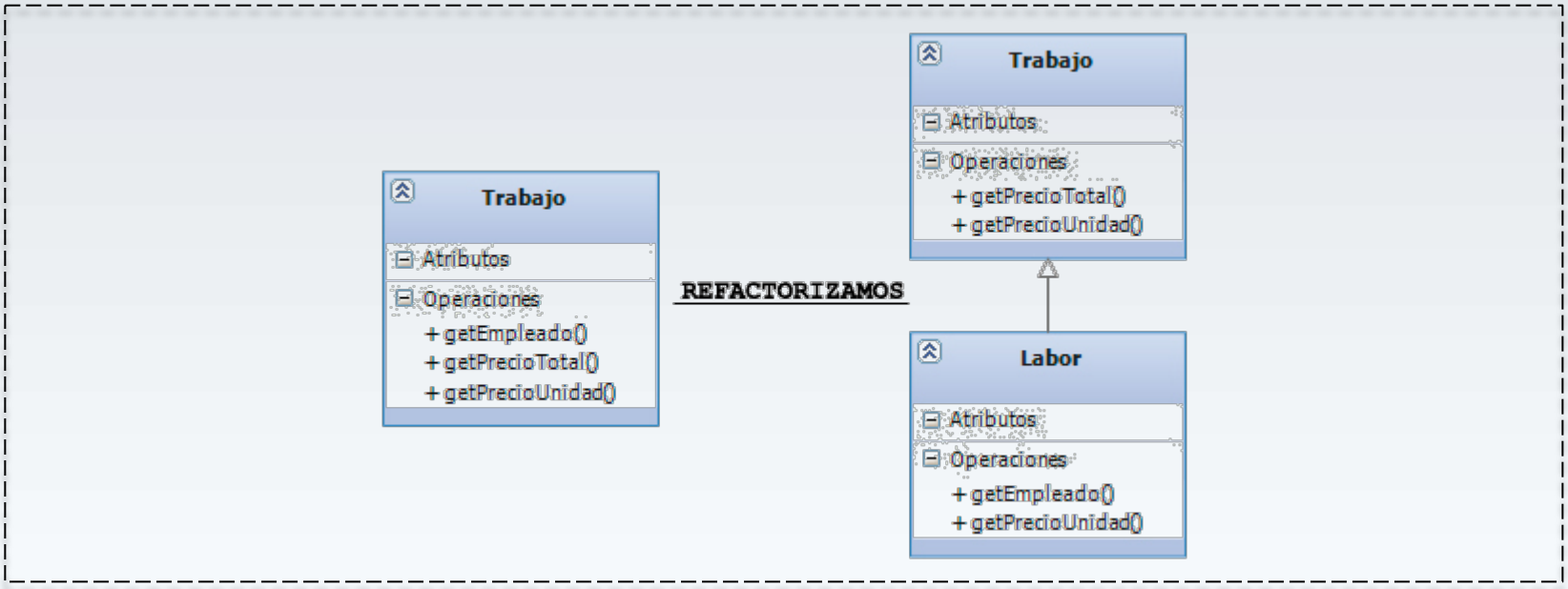
1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Extraer SubClase

✓Una clase tiene propiedades que solo son usadas en determinadas instancias.

✓Creamos una subclase para dicho subset de propiedades.



OPTIMIZACIÓN Y DOCUMENTACIÓN

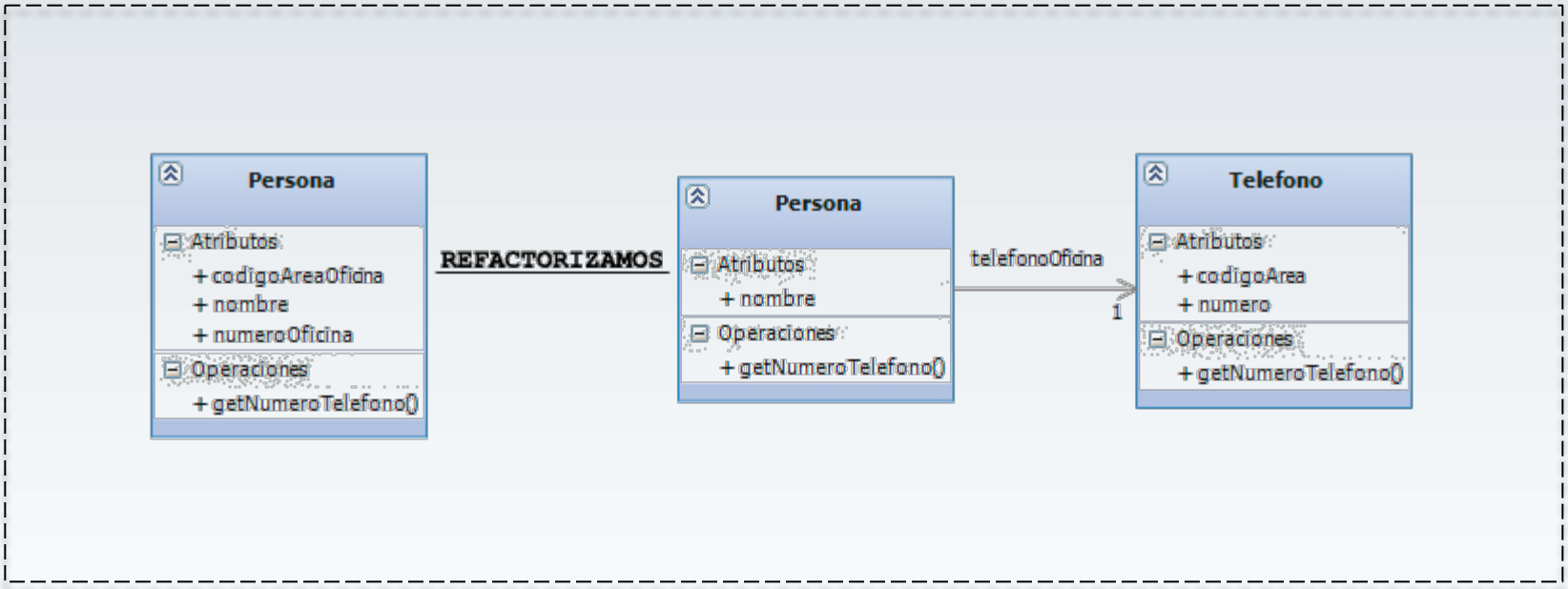
1. REFACTORIZACIÓN

➢ PATRONES DE REFACTORIZACIÓN MÁS USUALES

Extraer Clase

✓Tenemos una clase que hace el trabajo que debería ser hecho por dos.

✓Creamos una nueva clase y movemos los atributos y métodos relevantes de la vieja a la nueva clase.



OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ MALOS OLORES

Los "**malos olores**" son una relación de malas prácticas de desarrollo, indicadores de que nuestro código podría necesitar ser **refactorizado**. No siempre que detectemos un posible "**mal olor**" es un fallo de diseño en nuestro código y deberemos **refactorizarlo**, pero nos ayudará saber reconocer los indicadores y valorar si ese indicador es válido y tendremos que **refactorizar**.

Los "**malos olores**" no son necesariamente un problema en sí mismos, pero nos indican que hay un problema cerca. Varias de las malas prácticas de código reconocidas como antipatrones suelen corresponderse con "**malos olores**".

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ MALOS OLORES

Método largo. Los programas que viven más y mejor son aquellos con métodos cortos, que son más reutilizables y aportan mayor semántica.

Clase grande. Clases que hacen demasiado y por lo general con una baja cohesión, siendo muy vulnerables al cambio.

Lista de parámetros larga. Los métodos con muchos parámetros elevan el acoplamiento, son difíciles de comprender y cambian con frecuencia.

Clase de datos. Clases que solo tienen atributos y métodos tipo get y set. Las clases siempre deben disponer de algún comportamiento no trivial.

Estructuras de agrupación condicional. Lo que comentamos en un case o switch con muchas cláusulas, o muchos ifs anidados, tampoco es una buena idea.

Comentarios. No son estrictamente "malos olores", más bien "desodorantes". Al encontrar un gran comentario, se debería reflexionar sobre por qué algo necesita ser tan explicado y no es autoexplicativo. Los comentarios ocultan muchas veces a otro "mal olor".

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ MALOS OLORES

Atributo temporal. Algunos objetos tienen atributos que se usan solo en ciertas circunstancias. Tal código es difícil de comprender, ya que lo esperado es que un objeto use todas sus variables.

Generalidad especulativa. Jerarquías con clases sin utilidad actual, pero que se introducen por si en un futuro fuesen necesarias. El resultado son jerarquías difíciles de mantener y comprender, con clases que pudieran no ser nunca de utilidad.

Jerarquías paralelas. Cada vez que se añade una subclase a una jerarquía hay que añadir otra nueva clase en otra jerarquía distinta.

Intermediario. Clases cuyo único trabajo es la delegación y ser intermediarias.

Legado rechazado. Subclases que usan solo un poco de lo que sus padres les dan. Si las clases hijas no necesitan lo que heredan, generalmente la herencia está mal aplicada.

Intimidad inadecuada. Clases que tratan con la parte privada de otras. Se debe restringir el acceso al conocimiento interno de una clase.

Cadena de mensajes. Un cliente pide algo a un objeto que a su vez lo pide a otro y éste a otro, etc.

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ MALOS OLORES

Obsesión primitiva. Uso excesivo de tipos primitivos. Existen grupos de tipos primitivos (enteros, caracteres, reales, etc.) que deberían modelarse como objetos. Debe eliminarse la reticencia a usar pequeños objetos para pequeñas tareas, como dinero, rangos o números de teléfono que debieran muchas veces ser objetos.

Clase perezosa. Una clase que no está haciendo nada o casi nada debería eliminarse.

Cambios en cadena. Un cambio en una clase implica cambiar otras muchas. En estas circunstancias es muy difícil afrontar un proceso de cambio.

Envidia de características. Un método que utiliza más cantidad de cosas de otro objeto que de sí mismo.

Duplicación de código. Como comentábamos en el capítulo 1, duplicar, o copiar y pegar, código no es una buena idea.

Grupos de datos. Manojos de datos que se arrastran juntos (se ven juntos en los atributos de clases, en parámetros, etc.) debieran situarse en una clase. Tiene beneficios inmediatos como son la reducción de las listas de parámetros y de llamadas a métodos.

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ REFACTORIZACIÓN Y PRUEBAS

Las pruebas son una parte fundamental en el proceso de refactorización, tenemos que recordar que la piedra angular de la refactorización es no agregar ni modificar la funcionalidad del programa mientras se **refactoriza**, por lo que la creación de pruebas de código nos permite corroborar si antes y después de una refactorización los resultados son los mismos o si, por el contrario, hemos hecho algo mal a la hora de **refactorizar**.

Es probable que nos encontremos con casos en los que la refactorización necesaria que hay que realizar es tan grande que afecta al diseño del código de manera muy significativa, lo cual nos obligará a modificar las pruebas. Como se ha comentado con anterioridad, la mejor opción es ir **refactorizando** sobre la marcha, pero si nos encontramos en algún momento con un problema como éste, lo más aconsejable es modificar las pruebas antes o a la vez que el código, de modo que las pruebas y la refactorización se guíen de manera **reflexiva y recíproca**.

En este sentido, también resultaría útil **reflexionar** sobre los cambios que traerá una refactorización concreta, tanto al diseño como a las pruebas, y cómo poder aprovechar esa relación y esos cambios en nuestro propio beneficio.

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ HERRAMIENTAS DE VISUAL STUDIO

Una refactorización manual larga y completa puede convertirse fácilmente en una tarea extremadamente pesada y complicada. Un trabajo tan tedioso parece pedir a gritos herramientas que automaticen el proceso, las cuales, sin duda, existen y son efectivas en gran medida, a pesar de ello, el problema de automatizar la refactorización sigue siendo algo complejo, ya que la mayoría de las refactorizaciones necesitan analizar la estructura del software que se quiere refactorizar.

Además de aplicar patrones determinados de refactorización, con el fin de conseguir un código limpio y optimizado en un tiempo récord, existen características dentro de los entornos de desarrollo y herramientas que nos facilitan el trabajo en tiempo real de programación, en el caso concreto de nuestro entorno de desarrollo escogido, tendríamos una herramienta propia llamada **IntelliSense** y un ser de herramientas propias de refactorización.

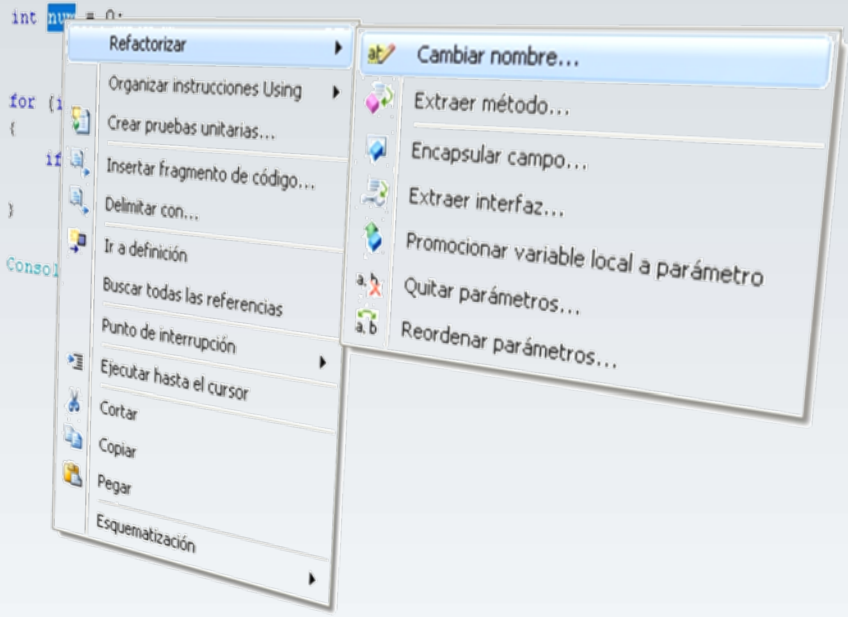
IntelliSense es una poderosa herramienta dentro de Visual Studio que nos permite realizar inspecciones a las clases y objetos, analiza nuestro código, incluso el JavaScript de nuestras páginas web y nos ofrece unas excelentes funciones de autocompletado.

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ HERRAMIENTAS DE VISUAL STUDIO

En Visual Studio disponemos de una serie de refactorizaciones automáticas que nos permiten realizar varias de las refactorizaciones vistas con anterioridad de manera rápida y sencilla.



Para acceder a éstas refactorizaciones automáticas basta con hacer click con el botón derecho y elegir la operación que queremos realizar. La operación seleccionada se realizará sobre la zona que hayamos previamente seleccionado.

OPTIMIZACIÓN Y DOCUMENTACIÓN

1. REFACTORIZACIÓN

➢ OTRAS HERRAMIENTAS

Disponemos de una amplia galería de extensiones y añadidos a Visual Studio que pueden sernos de suma utilidad. Además, muchas de ellas pueden venir dadas por un lenguaje concreto, por lo que también tendríamos que tener presente ese parámetro o restricción, teniendo en cuenta que en este libro nos enfocamos en el lenguaje de programación **C#**, recomendamos dos extensiones adicionales para optimizar el proceso de creación de código y su refactorización, **ReSharper y CodeRush**, ambas están disponibles para su descarga desde el repositorio de extensiones de Visual Studio.

ReSharper

✓http://www.jetbrains.com/resharper/documentation/index.jsp

CodeRush

✓http://documentation.devexpress.com/#CodeRushXpress/CustomDocument8099

OPTIMIZACIÓN Y DOCUMENTACIÓN

2. CONTROL DE VERSIONES

En el Capítulo 2, realizamos una primera toma de contacto con el desarrollo **colaborativo**, su funcionalidad y su relación directa con el control de versiones.

Como comentamos, no es el único uso que nos ofrece el control de versiones, ya que, aunque desde luego su funcionalidad parece destinada a un desarrollo colaborativo completo, en donde muchos programadores trabajan de manera simultánea en un proyecto.

Se suele utilizar de manera muy habitual para llevar un control de las versiones o **revisiones** de un determinado programa y poder tener un **repositorio** accesible con las diferentes versiones creadas.

También se utiliza el código del control de versiones de modo **exclusivo**, donde para poder realizar cambios se debe marcar previamente cuál es el elemento sobre el que se va a trabajar, de este modo, el **control de versiones** impedirá que otro usuario pueda modificar ese elemento hasta que se elimine la restricción. De este modo, se evita la aparición de **conflictos** o inconsistencias que suelen aparecen en el desarrollo colaborativo al poder desarrollar de modo asíncrono y simultáneo sobre el código del repositorio en detrimento de la libertad que ofrece el desarrollo colaborativo completo.

OPTIMIZACIÓN Y DOCUMENTACIÓN

2. CONTROL DE VERSIONES

➢ REPOSITORIOS

Un **repositorio** es básicamente un **servidor de archivos** típico, con una gran diferencia: lo que hace a los repositorios especiales en comparación con esos servidores de archivos es que recuerdan todos los cambios que alguna vez se hayan escrito en ellos, de este modo, cada vez que actualizamos el repositorio, éste recuerda cada cambio realizado en el archivo o estructura de directorios. Además, permite establecer información adicional por cada actualización, pudiendo tener por ejemplo un **changelog** de las versiones en el propio repositorio.

Cada herramienta de control de versiones tiene su propio repositorio, y, por desgracia, no son **interoperables**, es decir, no puedes obtener los datos del repositorio o actualizarlo si el repositorio y el control de versiones no coinciden.

También podríamos asociar una **copia de trabajo** (la que se encuentra en nuestro disco duro) a varios repositorios del mismo control de versiones siempre y cuando éste soporte replicación de repositorios.

OPTIMIZACIÓN Y DOCUMENTACIÓN

2. CONTROL DE VERSIONES

➢ VISUAL SVN SERVER

Para poder hacer uso del control de versiones, antes necesitamos de un **repositorio** con el que sincronizar nuestra copia de trabajo. Para ello, vamos a montar uno de los repositorios más populares y utilizados, el conocido como **SubVersion (SVN)**.

El proceso de instalación es muy sencillo, el único aspecto clave que tenemos que tener en cuenta es marcar la opción de instalación **Visual SVN Server and Management Console**.

Una vez instalado, sólo necesitamos crear un **repositorio** (un directorio) y un **usuario** para acceder a él, una vez realizados esos sencillos pasos ya tendríamos al repositorio listo y dispuesto para ser usado.

OPTIMIZACIÓN Y DOCUMENTACIÓN

2. CONTROL DE VERSIONES

➢ TEAM FOUNDATION SERVER

Microsoft tiene a su disposición un sistema de control de versiones llamado **Team Foundation Server**, que utiliza una serie de bases de datos **SQL Server** para almacenar los datos y el **IIS** para publicarlos.

Este control de versiones es exclusivo de productos de Microsoft como Visual Studio, lo cual nos ofrece varias ventajas, como la perfecta integración con plataformas de trabajo como **Sharepoint**, pero también muchas restricciones, sobre todo si se trata de trabajo colaborativo, y cuestiones de licencias.

El propio Visual Studio tiene integrado el **cliente de TFS**, por lo que en caso de que se disponga de un **repositorio de TFS** se podría utilizar de un modo similar al de cualquier cliente de control de versiones.

OPTIMIZACIÓN Y DOCUMENTACIÓN

2. CONTROL DE VERSIONES

➢ ANKH-SVN

AnkhSVN es un cliente disponible como extensión para Visual Studio que nos permite trabajar con los repositorios de **SubVersion**.

Actualizar ficheros en el repositorio: Como aún no hemos utilizado el control de versiones, vamos a actualizar el repositorio con los ficheros de una solución cualquiera. Por lo tanto, abriremos cualquier proyecto o solución en la que hayamos trabajado para subirla al repositorio. Una vez abierta, haremos clic con el botón derecho en la solución desde el explorador de proyectos y seleccionaremos la opción **Add Solution to Subversion.**

En la ventana emergente que nos aparecerá, escribiremos el nombre del repositorio que creamos previamente con el Visual SVN e introducimos nuestro usuario y contraseña, con éste sencillo paso ya tendríamos nuestra solución vinculada al repositorio.

Cada vez que queramos actualizar nuestra copia de trabajo en el repositorio deberemos hacer clic con el botón derecho en la solución desde el explorador de soluciones y seleccionaremos la opción **Commit solution changes**.

OPTIMIZACIÓN Y DOCUMENTACIÓN

2. CONTROL DE VERSIONES

- **ANKH-SVN**

Actualizar ficheros locales a una versión específica: Si queremos actualizar nuestra copia de trabajo con una versión concreta existente en nuestro repositorio, tendríamos que efectuar una operación de actualización. Para ello, haremos clic con el botón derecho en nuestra solución y elegiremos el submenú Subversión y, dentro de él, la opción **Update to specific version**.

En la ventana emergente que nos aparecerá, elegiremos la opción Revision para actualizarlo a una versión concreta.

Veremos que nos sale un campo para escribir la versión que queremos y al lado un botón marcado con "…", haremos clic en dicho botón y podremos ver el historial de versiones de nuestro repositorio.

Luego sólo tendríamos que elegir la versión que queremos del historial de versiones para que nos actualice la copia de trabajo con la versión elegida.

OPTIMIZACIÓN Y DOCUMENTACIÓN

3. DOCUMENTACIÓN

En todo proyecto de software, es muy importante la **documentación**, no solo para el usuario final, sino para los propios desarrolladores, tanto para uno mismo como para otros desarrolladores que tengan que trabajar en el presente o futuro con el proyecto.

Ya hemos comentado cómo ayuda la **refactorización** en el entendimiento del código, pero siempre nos será más comprensible una buena documentación, no solo porque está escrito en nuestro propio lenguaje, sino porque puede contener notas aclaratorias que el código no nos podría decir directamente, como por ejemplo el patrón utilizado o documentación y referencia sobre las librerías ajenas utilizadas en el proyecto.

En los proyectos web, la documentación técnica tiene un objetivo añadido, ya que es habitual que necesitemos de configuraciones específicas a la hora de desplegar el proyecto, como podría ocurrir con un enrutamiento específico o un sistema de autentificación específico.

OPTIMIZACIÓN Y DOCUMENTACIÓN

3. DOCUMENTACIÓN

El primer paso en una buena documentación es el uso apropiado de comentarios. En este mismo capítulo hemos hablado sobre los comentarios, señalando que si un código necesita ser comentado es porque no es suficientemente descriptivo por sí solo y necesita ser refactorizado. Aunque esto sea efectivamente así, sigue siendo una buena ayuda al entendimiento del código y sobre todo muy utilizado a la hora de crear notas dentro del código que no necesariamente tienen que ser una explicación del funcionamiento del código, además se utiliza de manera recurrente con el fin de omitir algunas instrucciones para que no se ejecuten, muy importante en los procesos de depuración.

Cada lenguaje suele tener sus propios modos de establecer comentarios, en **Visual Basic**, utilizaríamos el doble ‘ para crear una línea de comentario, mientras que en **C#**, y muchos otros lenguajes, se utiliza la doble **/** para crear ese comentario. Tenemos también dos modos de crear comentarios: comentarios de línea o comentarios de bloque.

OPTIMIZACIÓN Y DOCUMENTACIÓN

3. DOCUMENTACIÓN

Además de los comentarios simples de una línea o de bloque, tendríamos un tercer tipo de comentario: el comentario de **documentación**. Este tipo de comentarios aportan información sobre las clases y métodos de nuestro código, ofreciendo información específica y estructurada sobre su función y sus datos de entrada y salida.

La información adicional surge fruto de una inspección en el código y se encuentra en formato estructurado a modo de etiquetas donde podremos añadir la información que nos muestra en la plantilla y añadir nosotros las etiquetas que creamos convenientes. En C# para Visual Studio se utiliza la triple barra (///) para marcar el comentario de documentación.

```
/// <summary>
/// Método que crea una nueva entrada en los movimientos de la cuenta
/// </summary>
/// <param name=”fecha”>Fecha del movimiento</param>
/// <param name=”descripcion”>Descripción del movimiento</param>
/// <param name=”tipo”>Tipo: Ingreso o Gasto</param>
/// <param name=”importe”>Cantidad del importe</param>
/// <returns>Devuelve el mensaje resultante de la operación</returns>
public string nuevoMovimiento(DateTime fecha, string descripcion, int tipo, double
importe) {
[…]
}
```

OPTIMIZACIÓN Y DOCUMENTACIÓN

3. DOCUMENTACIÓN

Estos comentarios tienen además un objetivo añadido, ya que, si lo especificamos en las propiedades del proyecto, serán utilizados por Visual Studio para crear un archivo **XML** de documentación con esa información. Para activar esa funcionalidad, nos vamos a la pestaña **Generar** de las propiedades del proyecto y activamos la casilla Archivo de documentación **XML**; una vez hecho, si **compilamos** nuestro proyecto, podemos ver que, en el directorio raíz del proyecto, se ha creado un archivo **XML** que contiene una información básica sobre el ensamblado y las clases pregenerado por Visual Studio, y también los comentarios de documentación que hemos incluido en nuestro proyecto.

Si realizásemos esa operación en todos los métodos y clases de nuestro proyecto, podremos tener de un modo sencillo y rápido una documentación básica sobre nuestro proyecto con un peso ligero y sumamente portable y modificable.

OPTIMIZACIÓN Y DOCUMENTACIÓN

3. DOCUMENTACIÓN

➢ VSDOCMAN

Seguramente, la **documentación XML** que nos genera el entorno de desarrollo de manera automática no sea suficiente para nuestros propósitos, deberíamos poder crear también archivos de ayuda que vincular a nuestro proyecto.

Es por ello que utilizaremos una herramienta disponible como extensión para Visual Studio: **VSDocMan**. **VSDocMan** es una herramienta ligera y muy sencilla de utilizar. Nos permite crear archivos de ayuda en base a la documentación que hemos creado con los comentarios de documentación.

Podemos establecer una serie de configuraciones básicas para el formateado de los documentos de ayuda e incluso podremos configurarlo para que nos interprete las etiquetas personalizadas que hayamos podido incluir en nuestros comentarios simplemente con añadirlas en el apartado **User-defined Tags** de la configuración de **VSDocMan**.