Ciclo Formativo DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Módulo 5 Entornos de desarrollo

Unidad Formativa 2

INICIATIVA Y COORDINACIÓN
IFP Innovación en Formación Profesional

Supervisión editorial y metodológica:
Departamento de Producto de Planeta Formación

Supervisión técnica y pedagógica:
Departamento de Enseñanza de **IFP** Innovación en Formación Profesional

Módulo: Entornos de desarrollo
UF 2 / Desarrollo de Aplicaciones Multiplataforma

MÓDULO 5

Unidad Formativa 2

Entornos de desarrollo

Esquema de contenido

1. REFACTORIZACIÓN

Definición de 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

En el libro Refactoring: Improving the Design of Existing Code Fouler se decía que eran cambios realizados en el software para hacerlo más entendible y modificable, por lo que en esencia no era una optimización de código, ya que ésta en ocasiones lo hace menos comprensible, tampoco se trata de solucionar errores o mejorar algoritmos. Las refactorizaciones suelen efectuarse en la fase de mantenimiento del desarrollo de software, por lo que podría verse como un tipo de mantenimiento preventivo con el propósito de simplificar el código en vista a futuras necesidades y funcionalidades que añadir al software del programa.

La piedra angular de la refactorización se encuentra en una única y sencilla frase: "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".

Analicemos un momento qué es lo que hemos hecho en este sencillo ejemplo. Primeramente tenemos un programa que nos pide un número por teclado, y nos devuelve su cuadrado. Después de refactorizar, tenemos un programa que nos pide un número por teclado y nos devuelve su cuadrado, con la misma interfaz, parece que hemos cumplido con el condicional de la refactorización, pero ¿por qué lo que hemos hecho es una refactorización? A simple vista, simplemente parece que hemos alargado innecesariamente el código del programa, y, sin duda, es verdad para un ejemplo tan sencillo, pero ¿qué ventajas nos aporta el haber extraído una operación en un método? Previsión. Eso es lo que hemos hecho, simplemente nos hemos "preparado" para futuras mejoras o necesidades, es decir, nos hemos anticipado. Imaginad que el programa no solo tiene que calcular el cuadrado de un solo número, sino de muchos números y de distintas fuentes, ¿no resultaría más útil y menos redundante un método que nos calcule el cuadrado de un número dado en vez de repetir la operación en cada salida por pantalla?

Como se puede observar, las refactorizaciones son pequeños cambios que hacen que el código sea más visible, flexible, entendible y modificable. Ahora bien, nos podría surgir la duda de cuándo refactorizar nuestro código, no podemos estar pendientes y tener presentes todas las posibles modificaciones que nuestro programa vaya a tener (más que nada porque no podemos conocer el futuro), ni cuándo voy a tener que reutilizar código anterior. Es por ello que se establece la refactorización como parte del mantenimiento del código, no se trata de planificar nuestras etapas de refactorización, es algo que se tiene que realizar mientras se desarrolla. Muchas veces, mientras estás agregando una nueva funcionalidad al programa, te das cuenta de que el código existente es difícil de entender, o de que cambiar un poco parte de ese código o del diseño .podría facilitar la realización de cambios en un futuro como el que se está llevando a cabo. No tiene que ser exactamente en el mismo instante en que detectemos un problema de ese tipo tampoco, la funcionalidad no debe verse alterada. Por ello siempre hay que realizarlo todo en pasos completos, pasos completos y pequeños, por ello debemos acabar lo que estamos haciendo y, una vez hecho eso, ponernos a refactorizar. Es un proceso que se tiene que intercalar de manera habitual durante el desarrollo. Para comprender bien este proceso y cuándo se recomienda refactorizar, se suele utilizar la metáfora de los dos sombreros.

Cuando uno empieza a programar se pone el sombrero de "hacer código nuevo", una vez que ha terminado una parte del programa, lo ha compilado, lo ha probado y ha comprobado que funciona, deja esa parte del programa funcionando y sigue haciendo su programa. Mientras sigue desarrollando, se da cuenta de que tiene un trozo de código que podría reutilizar en lo que está haciendo (o en algo que pretende hacer) o simplemente ve algo que hecho de otra manera le facilita el trabajo, en ese momento se pone el sombrero de "arreglar código". Ahora no está introduciendo nada nuevo, solo está extrayendo operaciones en métodos, moviendo código de un sitio a otro, dejando lo métodos más pequeños y con una nomenclatura más comprensible; una vez termina, el código funciona exactamente igual que antes, pero está hecho de otra manera que le facilita el trabajo o que evita caer en malas prácticas de código como la redundancia y la duplicación de código.

El concepto es combinar la creación de código nuevo y la mejora y optimización del código de manera alternativa, siempre en pasos pequeños (al menos, lo más pequeños posible, claro), es decir, refactorizar de modo sistemático como parte del desarrollo y del mantenimiento del código, y no como una etapa planificada.

1.1. TABULACIÓN

La tabulación puede que no sea una técnica o una práctica propia de la refactorización en sí misma, ya que no se modifica código. No obstante, con la tabulación, el código queda más claro, con lo que también resulta más fácil de ver y entender, que es parte del objetivo de la refactorización. Definición de tabulación: "La tabulación (también llamada justificación o sangrado) nos permite visualizar el código organizado jerárquicamente sangrando las líneas de código dentro de los bloques de código".

Hoy en día cualquier entorno de desarrollo con el que trabajemos nos maquetará el código del programa con un sangrado y nos coloreará las palabras reservadas, tipos de variables, nombres de clases, namespaces. La tabulación permite ver de una manera rápida y sencilla los niveles y la profundidad de los bloques de código. Podemos ver el código de un programa como si fuera una estructura de árbol, donde cada bloque de código es un nodo del árbol y la anidación de nodos nos aporta una profundidad y una jerarquía entre dichos bloques de código.

En el siguiente ejemplo se puede observar el sangrado de un programa sencillo:

Ejemplo

CON SANGRADO

```
class Proyecto{
    Persona[] participantes;
    bool Participante (Persona X) {
        for (int i = 0; i < participantes.Length; i++) {
            if (participantes[i].id == x.id)
            return true;
        }
        return false;
    }
```

```
    IList<Persona> GetHijosFromParticipantas (Persona persona){
    IList<Persona> Hijos = new List<Persona> ();
    if (participantes.Count () > 0)
    Hijos = participantes
        .Where (p => p.padre == persona.id.).To List();
    return Hijos;
    }
}
```

En el ejemplo observamos que, en una misma clase, los dos métodos se encuentran al mismo nivel, mientras que las instrucciones y estructuras de control que se encuentran dentro de los métodos están en un nivel inferior, y así consecutivamente dentro de cada instrucción y estructura de control. Todo esto está muy bien, pero tampoco parece gran cosa, ¿verdad?, bueno, comprobémoslo, veamos ahora ese mismo código, pero sin sangrado, todo al mismo nivel.

Ejemplo

SIN SANGRADO

```
class Proyecto
Persona[] participantes;
bool Participante (Persona x){
 for (int i = 0; i < participantes.Length; i++) {
 if (participantes[i].id == x.id)
return true;
}
return false;
}
 IList<Persona> GetHijosFromParticipantes (Persona persona)
IList<Persona> Hijos = new List<Persona>();
if (participantes.Count () > 0)
Hijos = participantes
.Where (p => p.padre == persona.id).ToList();
return Hijos;
}
```

Ahora ya no parece tan sencillo el código, hay que estar más pendientes de dónde acaban las instrucciones y bloques, por lo que resulta mucho más complicado y engorroso trabajar sin código tabulado. Aun así, se puede pensar que no es para tanto, que todavía se podría modificar este código sin necesidad de

sangrados, pero imaginaos ahora que tengáis que trabajar con un código de mil líneas sin tabular. No parece una perspectiva muy halagüeña, por lo que si vuestro entorno de desarrollo no os tabula la anidación de bloques de manera automática, o trabajáis sin un entorno de desarrollo, no os olvidéis nunca de tabular, una tabulación por cada nuevo nivel.

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.

1.2. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Los patrones de refactorización, comúnmente llamados catálogos de refactorización o métodos de refactorización, son diversas prácticas concretas para refactorizar nuestro código. Plantean casos o problemas concretos y su resolución refactorizadora, pudiendo ver así un antes y después y sobre todo comprendiendo el porqué. Sin lugar a dudas el catálogo de patrones de refactorización más extendido y aceptado es el de Martin Fowler. En esta sección veremos algunos de los métodos de refactorización más usuales explicados a modo de casos mediante un ejemplo.

- Extraer Método
  - Tienes un fragmento de código que puede agruparse.
  - Conviertes el fragmento en un método cuyo nombre explique el propósito del método.

Ejemplo

EXTRAER MÉTODO

```
void imprimirTodo () {
imprimirbanner ();
//detalles de impresión
Console.WriteLine (“nombre: “+ _nombre);
    ConsoleWriteLine (“cantidad “ + getCargoPendiente());
}
Refactorizamos
void ImprimirTodo () {
    imprimirBanner();
    imprimirDetalles(getCargoPendiente());
}
void imprimirDetalles (double cargoPendiente) {
    Console.WriteLine(“nombre: “ + _nombre);
        Console.WriteLine(“cantidad “ + cargoPendiente);
}
```

- 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.

Ejemplo

SEPARAR VARIABLES TEMPORALES

```
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);
```

- Eliminar Asignaciones a Parámetros
  - Un parámetro es usado para recibir una asignación.
  - Usamos una variable temporal en su lugar.

Ejemplo

ELIMINAR ASIGNACIONES A PARÁMETROS

```
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;
[…]
}
```

- 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.

Ejemplo

MOVER MÉTODO

```
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))[…]
```

- 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

Ejemplo

CONSOLIDAR FRAGMENTOS DUPLICADOS EN CONDICIONALES

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

Refactorizamos

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

- 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

Ejemplo

REEMPLAZAR NÚMERO MÁGICO CON CONSTANT SIMBÓLICA

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

Refactorizamos

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

- Reemplazar Número Mágico com Método Constante
  - Tenemos um literal com um significado particular.
  - Creamos um método que nos devuelve el literal, lo nombramos significativamente y lo sustituimos por el literal.

Ejemplo

REEMPLAZAR NÚMERO MÁGICO CON MÉTODO CONSTANTE

```
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;
}
```

- 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.

Ejemplo

REEMPLAZAR ARRAY CON OBJETO

```
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”);
```

- Encapsular Atributo
  - Tenemos un atributo público.
  - Lo convertimos a privado y le creamos métodos de acceso.

Ejemplo

ENCAPSULAR ATRIBUTO

public String _nombre

Refactorizamos

private String _nombre;

public String getNombre() {return _nombre;}

public void setNombre (String arg) {_nombre = arg;}

- Encapsular Atributo como Propiedad
  - Tenemos un atributo publico
  - Lo convertimos en una propiedad del objeto.

Ejemplo

ENCAPSULAR ATRIBUTO COMO PROPIEDAD

public String _nombre

Refactorizamos

public virtual string _nombre (get; set;)

- 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.
- Extraer SubClase
  - Una clase tiene propiedades que solo son usadas en determinadas instancias.
  - Creamos una subclase para dicho subset de propiedades.
- 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.

1.3. 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.

- **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.
- **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 de datos**. Clases que solo tienen atributos y métodos tipoget 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 auto explicativo. Los comentarios ocultan muchas veces a otro "mal olor".
- **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.

- **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.

1.4. 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.

No se utilizan las pruebas únicamente para refactorizar, como ya hemos visto en capítulos anteriores, pero sí que son conceptos que vienen de la mano. Lo más usual en el desarrollo de un software es que las pruebas se hayan creado antes de siquiera plantearse refactorizar, es decir, es un paso programado, estudiado y planificado que se aprovecha para refactorizar, y no al revés, es decir, no se refactoriza y aprovechamos que vamos a refactorizar y creamos la pruebas, sino que las pruebas son un recurso que aprovechamos a la hora de refactorizar, pero no pertenecen a una relación causa-efecto.

Con ayuda de unas pruebas automatizadas previas, evitamos el riesgo a la hora de refactorizar, ya no nos supone un problema, pues podemos comprobar de una manera sencilla si hemos refactorizado mal, algo que es particularmente útil cuando se necesita refactorizar un código de un proyecto que no hemos creado nosotros, sino que viene desde fuera.

Cabe destacar que la mayor ventaja que nos aportan las pruebas a la hora de refactorizar no se trata de poder comprobar, pues eso ya lo podemos hacer compilando y probando el programa manualmente, sino de una facilidad temporal.

No obstante, las pruebas también podrían llegar a ser un inconveniente a la hora de refactorizar si se emplean de forma inapropiada.

Es muy importante resaltar que las pruebas, ya sean funcionales o unitarias, tienen que comprobar el código de manera independiente a como esté implementado. Cuanto mayor sea el acoplamiento de las pruebas con la implementación, mayores serán los cambios que habrá que realizar en las pruebas cada vez que se modifica parte del código.

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.

1.5. 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.

Hoy en día, la gran mayoría de entornos de desarrollo traen unas refactorizaciones básicas incluidas, unas refactorizaciones automáticas que son sencillas de realizar y que más que ayudarnos a refactorizar cuestiones complicadas nos ahorran tiempo evitando tener que realizar tareas mecánicas y cotidianas de refactorización.

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.

1.5.1. IntelliSense

Es una aplicación de Microsoft integrada en el entorno de desarrollo Visual Studio destinada a autocompletar. Nada más empezar a escribir código IntelliSense detectará qué es lo que queremos escribir, mostrándonos sugerencias alfabéticamente, las cuales además tienen un icono identificador que nos indicará si se trata de una palabra reservada, de una variable, de un método o de una clase.

Además, también nos completará las propiedades y métodos inherentes a cualquier instancia o clase. Esta característica funciona mostrando todas esas propiedades o métodos accesibles mediante un “.”, si seguimos escribiendo nos filtrará entre esas opciones según lo que estemos escribiendo, o podríamos simplemente recorrer el listado que nos ofrece IntelliSense con el fin de encontrar lo que buscamos.

También incluye no solo la nomenclatura y el tipo de las propiedades y métodos, sino que nos especifica qué valores devuelve, cuáles son sus parámetros y cuántas y cuáles son sus sobrecargas.

Por si fuera poco, también nos ofrece la posibilidad de automatizar las estructuras de las funciones, ahorrándonos el tiempo de escribir algo que escribimos muchas veces al día, con paréntesis, corchetes, saltos de línea... lo único que

hay que hacer es escribir el nombre de la función y darle al tabulador. Visual Studio nos creará la estructura necesaria para dicha función. Por ejemplo, si escribimos:

```
if
```

y le damos al tabular obtendremos:

```
if(true)
{
}
```

Dejándonos la estructuralista para ser usada y rellenada. Cabe destacar que el tabulador (5) también lo podemos usar cuando estamos explorando un catálogo de métodos de un objeto, así conseguimos escribir una larga línea en un período extremadamente corto de tiempo, de una manera muy fácil, sencilla e intuitiva.

1.5.2. Refactorizaciones

No todas las herramientas que nos ofrece un entorno de desarrollo se basan únicamente en completar lo que escribimos, sino que también incluyen patrones básicos de refactorización para su uso.

Los patrones más habituales que uno se puede encontrar en un entorno de desarrollo serían Renombrar, Extraer

método, Encapsular campo (atributo), Extraer interfaz, Promocionar variable local a parámetro, Quitar parámetros y Reordenar parámetros.

Para acceder a estas funcionalidades, simplemente hay que seleccionar el código que se quiera refactorizar, y elegir la refactorización deseada, el propio entorno de desarrollo se encarga de realizarlo.

1.5.3. Renombrar

Queremos cambiar el nombre de una variable o método para hacerlo más significativo.

1.5.4. Otras Herramientas

Las herramientas de refactorización que nos ofrece Visual Studio son muy sencillas, rápidas y eficaces, pero no son nuestro único recurso para agilizar el proceso de codificación de software ni para refactorizarlo.

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 tal y como hicimos con el NUnit en el capítulo anterior.

Para aprender sobre su uso y manejo, disponemos de excelentes documentaciones en la página oficial de cada extensión.

ReSharper:

http://www.jetbrains.com/resharper/documentation/index.jsp. i CodeRush:

http://documentation.devexpress.com/#CodeRushpress/Custom Document:8099 Como comentario adicional, el enlace que se muestra de CodeRush se corresponde con su versiónXpress y contiene una serie de ejemplos realizadas a modo de animaciones, con lo que se puede apreciar perfectamente el funcionamiento y uso de cada funcionalidad de esta herramienta.

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, también 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, siendo utilizado tanto como copia de respaldo como para poder volver a una versión anterior o incluso porque por necesidades específicas necesitemos utilizar el código existente en una versión determinada. Por ejemplo, podríamos necesitar crear un parche que solucione un problema con una versión en concreto sin que el usuario final tuviese que actualizar todo el producto a la versión actual.

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.

2.1. 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. Realmente, al ser un servidor de archivos (especial, pero en esencia es lo mismo), siempre se podría acceder directamente a los archivos almacenados en el repositorio y obtener el código sin mayor problema, pero no tendremos un control de versiones sobre dicho código hasta que lo asociemos a dicho repositorio con el control de versiones adecuado.

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; no obstante, si intentásemos realizar esa operación con repositorios de controles de versiones diferentes, es impredecible cómo responderá el sistema ante ello.

2.2. HERRAMIENTAS DE CONTROL DE VERSIONES

Para poder asociar nuestro código a un repositorio del control de versiones, es necesaria una herramienta que se encargue de ir observando los cambios realizados para luego actualizarlo de manera que el repositorio los entienda y pueda llevar el control de versiones de manera apropiada.

2.2.1. 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.

Por esos motivos, y a pesar de haber elegido Visual Studio como IDE principal en el contenido de este libro, no se va a estudiar este control de versiones más allá de su mera mención.

A pesar de ello, 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.

En caso de que se quiera investigar sobre el uso y funcionamiento de TFS, se propone un tutorial sobre su instalación y configuración muy detallado, accesible desde http://mismUps.com/blogs/fagas/archive/2009/08/31/ usts-2010-instalaci-243-n-del-team-foundation-server-2010.aspx.

2.2.2. Cliente de control de versiones: ANKHSVN

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

El primer paso para aprender a utilizarlo es descargar e instalar la extensión AnkhSVN. Podemos descargar el cliente desde la siguiente dirección: http://ankhsvn.open.colab.net/dounloads.

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.

Actividad

2.1. Prueba el funcionamiento de la refactorización automática de Visual Studio Reordenar parámetros en cualquiera de los ejemplos o proyectos vistos hasta ahora.

2.2. Prueba el formatea automático de documentos de Visual Studio con diferentes configuraciones de formato en las opciones del editor de texto.

2.3. Revisa los ejemplos y códigos vistos hasta ahora. ¿Detecta "malos lores"? ¿Hay algún patrón de refactorización que los pueda solucionar?

2.4. Probar la verdadera utilidad de ItelliSense. Realiza un programa sencillo escribiendo el código en el bloc de notas y luego comprueba si ese código tiene errores de compilación que podría haber evitado si hubiese sido asistido mediante la herramienta IntelliSense de Visual Studio.

2.5. Modifica el código de tu proyecto sin guardar los cambios. ¿Observas algún cambio en el explorador de soluciones?, ¿y si guarda o compila?

2.6. Realiza cambios en el código que tiene vinculado el control de versiones y actualiza el repositorio. ¿Qué ocurre si actualizamos a una versión anterior?, ¿podemos volver a la última versión que hemos actualizado? Reflexiona sobre las posibilidades de los resultados obtenidos.

Índice