CF GRADO SUPERIOR

CICLOS FORMATIVOS

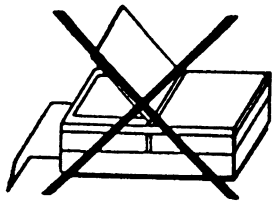
R.D. 1538/2006

Entornos de Desarrollo

Ra-Ma®

www.ra-ma.es/cf

CARLOS CASADO IGLESIAS



La ley prohíbe
Copiar o Imprimir este libro

ENTORNOS DE DESARROLLO
© Carlos Casado Iglesias

© De la Edición Original en papel publicada por Editorial RA-MA
ISBN de Edición en Papel: 978-84-9964-169-0
Todos los derechos reservados © RA-MA, S.A. Editorial y Publicaciones, Madrid, España.

MARCAS COMERCIALES. Las designaciones utilizadas por las empresas para distinguir sus productos (hardware, software, sistemas operativos, etc.) suelen ser marcas registradas. RA-MA ha intentado a lo largo de este libro distinguir las marcas comerciales de los términos descriptivos, siguiendo el estilo que utiliza el fabricante, sin intención de infringir la marca y solo en beneficio del propietario de la misma. Los datos de los ejemplos y pantallas son ficticios a no ser que se especifique lo contrario.

RA-MA es una marca comercial registrada.

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa. Sin embargo, RA-MA Editorial no asume ninguna responsabilidad derivada de su uso ni tampoco de cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene por objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa o de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.

Reservados todos los derechos de publicación en cualquier idioma.

Según lo dispuesto en el Código Penal vigente ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro sin autorización previa y por escrito de RA-MA; su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes, intencionadamente, reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica.

Editado por:
RA-MA, S.A. Editorial y Publicaciones
Calle Jarama, 33, Polígono Industrial IGARSA
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: editorial@ra-ma.com
Internet: www.ra-ma.es y www.ra-ma.com

Maquetación: Gustavo San Román Borrueco
Diseño Portada: Antonio García Tomé

ISBN: 978-84-9964-373-1

E-Book desarrollado en España en Septiembre de 2014

Entornos de Desarrollo

CARLOS CASADO IGLESIAS

Ra-Ma®

Descarga de Material Adicional

Este E-book tiene disponible un material adicional que complementa el contenido del mismo.

Este material se encuentra disponible en nuestra página Web www.ra-ma.com.

Para descargarlo debe dirigirse a la ficha del libro de papel que se corresponde con el libro electrónico que Ud. ha adquirido. Para localizar la ficha del libro de papel puede utilizar el buscador de la Web.

Una vez en la ficha del libro encontrará un enlace con un texto similar a este:

"Descarga del material adicional del libro"

Pulsando sobre este enlace, el fichero comenzará a descargarse.

Una vez concluida la descarga dispondrá de un archivo comprimido. Debe utilizar un software descompresor adecuado para completar la operación. En el proceso de descompresión se le solicitará una contraseña, dicha contraseña coincide con los 13 dígitos del ISBN del libro de papel (incluidos los guiones).

Encontrará este dato en la misma ficha del libro donde descargó el material adicional.

Si tiene cualquier pregunta no dude en ponerse en contacto con nosotros en la siguiente dirección de correo: ebooks@ra-ma.com

Dedicado especialmente a todos los amigos y familiares que me apoyaron y confiaron en mí para llevar este libro a buen puerto, en especial a Idoia Mugartegui, sin la que nada de esto habría sido posible

Índice

INTRODUCCIÓN **9**

CAPÍTULO 1. DESARROLLO DE SOFTWARE **11**
1.1 EL PROGRAMA INFORMÁTICO 12
1.1.1 Interacción con el sistema 12
1.2 LENGUAJES DE PROGRAMACIÓN 14
1.2.1 Clasificación y características 14
1.3 OBTENCIÓN DE CÓDIGO EJECUTABLE 17
1.3.1 Tipos de código (fuente, objeto y ejecutable) 17
1.3.2 Compilación 17
1.4 PROCESOS DE DESARROLLO 19
1.4.1 Análisis 19
1.4.2 Diseño 19
1.4.3 Codificación 20
1.4.4 Pruebas 20
1.4.5 Documentación 20
1.4.6 Explotación 21
1.4.7 Mantenimiento 21
1.5 ROLES QUE INTERACTÚAN EN EL DESARROLLO 21
1.6 ARQUITECTURA DE SOFTWARE 22
1.6.1 Patrones de desarrollo 22
1.6.2 Desarrollo en tres capas 38

RESUMEN DEL CAPÍTULO 41
EJERCICIOS PROPUESTOS 41
TEST DE CONOCIMIENTOS 44

CAPÍTULO 2. INSTALACIÓN Y USO DE ENTORNOS DE DESARROLLO **45**
2.1 CARACTERÍSTICAS 46
2.1.1 Extensiones y herramientas 46
2.1.2 Personalización y configuración 48
2.2 CRITERIOS DE ELECCIÓN DE UN IDE 48
2.2.1 Sistema operativo 49
2.2.2 Lenguaje de programación y *framework* 49
2.2.3 Herramientas y disponibilidad 49
2.3 USO BÁSICO DE UN IDE 51
2.3.1 Edición de programas y generación de ejecutables 51
2.3.2 Desarrollo colaborativo 51
2.4 NUESTRA ELECCIÓN VISUAL STUDIO 53
2.4.1 Instalación 54

2.4.2 Recorrido por las ventanas y paletas principales 55
2.4.3 Personalización y configuración 59

RESUMEN DEL CAPÍTULO 63
TEST DE CONOCIMIENTOS 64

CAPÍTULO 3. DEPURACIÓN Y REALIZACIÓN DE PRUEBAS 65
3.1 HERRAMIENTAS DE DEPURACIÓN 66
3.1.1 Puntos de ruptura 66
3.1.2 Puntos de seguimiento 67
3.1.3 Inspecciones 67
3.2 ANÁLISIS DE CÓDIGO 68
3.2.1 Analizador estático de código 68
3.3 CASOS DE PRUEBA 71
3.3.1 Caja blanca 72
3.3.2 Caja negra 73
3.3.3 Rendimiento 75
3.3.4 Coherencia 77
3.4 PRUEBAS UNITARIAS 78
3.4.1 Metodología 78
3.4.2 NUnit 79

RESUMEN DEL CAPÍTULO 83
EJERCICIOS PROPUESTOS 84
TEST DE CONOCIMIENTOS 85

CAPÍTULO 4. OPTIMIZACIÓN Y DOCUMENTACIÓN 87
4.1 REFACTORIZACIÓN 88
4.1.1 Tabulación 90
4.1.2 Patrones de refactorización más usuales 91
4.1.3 Malos olores 101
4.1.4 Refactorización y pruebas 102
4.1.5 Herramientas de Visual Studio 103
4.2 CONTROL DE VERSIONES 118
4.2.1 Repositorios 118
4.2.2 Herramientas de control de versiones 121
4.3 DOCUMENTACIÓN 124
4.3.1 Uso de comentarios 124
4.3.2 Herramientas 126

RESUMEN DEL CAPÍTULO 128
TEST DE CONOCIMIENTOS 129

CAPÍTULO 5. DISEÑO ORIENTADO A OBJETOS. DIAGRAMAS DE CLASE 131
5.1 INTRODUCCIÓN A UML 132
5.2 DISEÑO DE CLASES EN UML 133
5.2.1 Clases, atributos y métodos 133
5.2.2 Relaciones 135

5.3 HERRAMIENTAS 139
5.3.1 Herramienta de modelado de VS 139
5.3.2 UMLPad 144
RESUMEN DEL CAPÍTULO 146
EJERCICIOS PROPUESTOS 147
TEST DE CONOCIMIENTOS 147
CAPÍTULO 6. DISEÑO ORIENTADO A OBJETOS. DIAGRAMAS DE COMPORTAMIENTO 149
6.1 TIPOS Y CAMPO DE APLICACIÓN 150
6.2 DIAGRAMAS DE ACTIVIDAD 150
6.3 DIAGRAMAS DE CASOS DE USO 153
6.4 DIAGRAMAS DE SECUENCIA 155
6.4.1 Ingeniería inversa 157
RESUMEN DEL CAPÍTULO 163
EJERCICIOS PROPUESTOS 164
TEST DE CONOCIMIENTOS 165
CAPÍTULO 7. ¡PONLO EN PRÁCTICA! 167
7.1 NUESTRO PROYECTO 168
7.2 PLANTEAMIENTO 168
7.2.1 Diseño conceptual 168
7.2.2 Modelado completo 168
7.3 ¿QUÉ TIPO DE PROYECTO ES? 169
7.3.1 Tipos de proyecto 169
7.4 DOCUMENTACIÓN 170
7.5 OPCIONAL: INSTALACIÓN Y DISTRIBUCIÓN 170
7.6 NOTAS 170
7.7 PROYECTO PROPUESTO 171
CAPÍTULO 8. COMENTARIOS Y CONCLUSIONES 175
ÍNDICE ALFABÉTICO 179

Introducción

Este libro surge con el propósito de acercar al lector a los aspectos más importantes que encierran los entornos de desarrollo, aprovechando todas las facilidades y funcionalidades que ofrecen al programador. Con tal propósito, puede servir de apoyo también para estudiantes del Ciclo Formativo de Grado Superior de Informática y de Ingenierías Técnicas.

Hoy en día, existen muchos usuarios y profesionales de la Informática que discuten las ventajas e inconvenientes de algunos entornos de desarrollo y prefieren limitarse al uso exclusivo de uno de ellos. Se planteó que el contenido del libro fuese lo más genérico posible pero, teniendo en cuenta la necesidad de explicar el funcionamiento y uso concreto de determinadas herramientas CASE integradas o no en el entorno de desarrollo, se ha tenido que especificar un IDE como nuestra elección para seleccionar las diferentes herramientas y su manejo. Por ello, para ofrecer un conocimiento completo de su funcionamiento, el libro se centrará en el uso y estudio del Visual Studio 2010, utlizando en su mayoría conocimientos y aplicaciones del lenguaje de programación C#.

Para todo aquel que use este libro en el entorno de la enseñanza (Ciclos Formativos o Universidad), se ofrecen varias posibilidades: utilizar los conocimientos aquí expuestos para inculcar aspectos genéricos de los entornos de desarrollo o simplemente centrarse en preparar a fondo alguno de ellos. La extensión de los contenidos aquí incluidos hace imposible su desarrollo completo en la mayoría de los casos.

El objetivo principal del libro no es proporcionar conocimientos de programación o diseño, aunque se incluyan en su contenido, la información de este texto pretende enseñar el uso avanzado de los entornos de desarrollo para facilitar el desarrollo de software, aprovechando las extensiones y funcionalidades del mismo a la hora de codificar o completar una aplicación mientras se desarrolla.

Ra-Ma pone a disposición de los profesores una guía didáctica para el desarrollo del tema, que incluye las soluciones a los ejercicios expuestos en el texto. Puede solicitarla a *editorial@ra-ma.com*, acreditándose como docente y siempre que el libro sea utilizado como texto base para impartir las clases.

1 Desarrollo de software

OBJETIVOS DEL CAPÍTULO

- ✓ Reconocer los diferentes tipos de lenguajes de programación y las necesidades que cubren cada uno de ellos.
- ✓ Comprender el proceso de desarrollo de un software y las diferentes tareas que se deben realizar en las diferentes fases.
- ✓ Conocer las diferentes técnicas de arquitecturas de software, sus utilidades y las ventajas que ofrece cada una.

1.1 EL PROGRAMA INFORMÁTICO

Definición de programa informático: "*Un programa informático es un conjunto de instrucciones que se ejecutan de manera secuencial con el objetivo de realizar una o varias tareas en un sistema*".

Un programa informático es creado por un programador en un lenguaje determinado, que será compilado y ejecutado por un sistema. Cuando un programa es llamado para ser ejecutado, el procesador ejecuta el código compilado del programa instrucción por instrucción.

Se podría llegar a decir también que un programa informático es software, pero no sería una definición muy acertada, pues un software comprende un conjunto de programas.

1.1.1 INTERACCIÓN CON EL SISTEMA

El mejor modo para comprender cómo un programa interactúa con un sistema es revisando la funcionalidad básica y el programa básico, irnos directamente al concepto de programa y de sistema en su mínima expresión. Para ello, vamos a revisar el funcionamiento de una única instrucción dentro del conocido simulador von Neumann.

Como hemos comentado brevemente, el procesador ejecutará las instrucciones una a una, pero no solo eso, para cada instrucción realizará una serie de microinstrucciones para llevarla a cabo.

Vamos a realizar el recorrido que efectúa una instrucción de un modo conceptual, nada técnico, profundizando más y más dentro de la interpretación que hace el sistema de nuestro programa. Imaginemos que tenemos un programa extremadamente sencillo que pide por teclado dos números y los suma. La instrucción que realizará la operación de nuestro programa se podría corresponder con la siguiente línea:

```
c = a + b;
```

El ordenador tendrá reservada una cantidad de posiciones de memoria definidas por el tipo de variable que se corresponden con las variables de nuestra instrucción. Es decir, nuestras variables "a", "b" y "c" tendrán unas posiciones de memoria definidas que es donde el sistema almacenará los valores de las variables.

No obstante, el procesador no puede ejecutar esa instrucción por sencilla que sea de un solo golpe, la ALU (*Aritmetic Logic Unit*) tiene un número muy limitado de operaciones que puede realizar, usualmente SUMAR y RESTAR.

Para esta demostración en concreto, vamos a definir nuestra siguiente máquina de von Neumann:

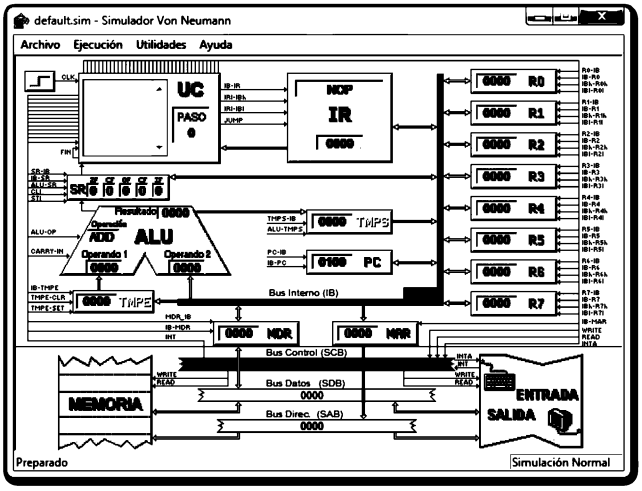


Figura 1.1. *Máquina de von Neumann*

Además, para nuestra demostración deberemos tener en cuenta una serie de reglas intrínsecas del sistema:

- La ALU solo puede realizar una operación a la vez.
- El registro temporal de la ALU, el *bus* y los registros solo pueden almacenar un dato a la vez.

Si definimos (para simplificar el modelo) que las posiciones de memoria de "a", "b" y "c" se corresponden con los registros R1, R2 y R3, las microinstrucciones que tendría que realizar nuestra máquina serían las siguientes:

```
R1 - Bus ; Bus - ALU-Temp ; R2 - Bus ; ALU_SUMAR ; ALU - Bus ; Bus - R3
```

Como es lógico, hoy en día, con los microprocesadores modernos, el funcionamiento, aunque muy similar en esencia, de cómo son interpretadas las instrucciones de nuestro programa por el sistema puede variar, sobre todo en lo que las reglas se refiere.

Sin embargo, hay cosas que no cambian, el programa se sigue almacenando en una memoria no volátil y se sigue ejecutando en la memoria de acceso aleatorio, al igual que todas las variables utilizadas.

La interacción con el sistema no es siempre una relación directa, no todos los programas tienen acceso libre y directo al hardware, es por ello que se definen los programas en dos clasificaciones generales: software de sistema y software de aplicación. Es el software de sistema el que se encarga de controlar y gestionar el hardware, así como de gestionar el software de aplicación, de hecho, en un software de sistema, como el sistema operativo, es en donde se ejecuta realmente el software de aplicación. Será el software de aplicación el que incorpore (gracias al compilador) las librerías necesarias para entenderse con el sistema operativo, y éste a su vez sería el que se comunicase con el hardware.

ACTIVIDADES 1.1

- Suponiendo la existencia de una operación en la ALU llamada DECREMENTAR, que utilizase el valor de la ALU-TEMP y le restase 1, especifique cuáles serían las microinstrucciones que se realizarían para multiplicar el contenido de R1 y R2 guardando el resultado en R3.

1.2 LENGUAJES DE PROGRAMACIÓN

Definición de lenguaje de programación:

"Un lenguaje de programación es un conjunto de instrucciones, operadores y reglas de sintaxis y semánticas, que se ponen a disposición del programador para que éste pueda comunicarse con los dispositivos de hardware y software existentes".

El idioma artificial que constituyen los operadores, instrucciones y reglas tiene el objetivo de facilitar la tarea de crear programas, permitiendo con un mayor nivel de abstracción realizar las mismas operaciones que se podrían realizar utilizando código máquina.

En un principio, todos los programas eran creados por el único código que el ordenador era capaz de entender: el código máquina, un conjunto de 0s y 1s de grandes proporciones. Este método de programación, aunque absolutamente efectivo y sin restricciones, convertía la tarea de programación en una labor sumamente tediosa, hasta que se tomó la solución de establecer un nombre a las secuencias de programación más frecuentes, estableciéndolas en posiciones de memoria concretas, a cada una de estas secuencias nominadas se las llamó instrucciones, y al conjunto de dichas instrucciones, lenguaje ensamblador.

Más adelante, empezaron a usar los ordenadores científicos de otras ramas, con muchos conocimientos de física o química, pero sin nociones de informática, por lo que les era sumamente complicado el uso del lenguaje ensamblador; como un modo de facilitar la tarea de programar, y no como un modo de facilitar el trabajo al programador informático, nace el concepto de lenguaje de alto nivel con FORTRAN (*FORmula TRANslation*) como primer debutante.

Los lenguajes de alto nivel son aquellos que elevan la abstracción del código máquina lo más posible, para que programar sea una tarea más liviana, entendible e intuitiva. No obstante, nunca hay que olvidar que, usemos el lenguaje que usemos, el compilador hará que de nuestro código solo lleguen 1s y 0s a la máquina.

1.2.1 CLASIFICACIÓN Y CARACTERÍSTICAS

La cantidad de lenguajes de programación es sencillamente abrumadora, cada uno con unas características y objetivos determinados, tal abrumador elenco de lenguajes de programación hace necesario establecer unos criterios para clasificarlos. Huelga decir que los criterios que clasifican los lenguajes de programación se corresponden con sus características principales.

Se pueden clasificar mediante una gran variedad de criterios, se podrían establecer hasta once criterios válidos diferentes con los que catalogar un lenguaje de programación. Algunos de dichos criterios pudieran ser redundantes, puesto que se encuentran incluidos explícitamente dentro de otros, como el determinismo o el propósito.

En este libro vamos a clasificar los lenguajes de programación siguiendo 3 criterios globales y reconocidos: el nivel de abstracción, la forma de ejecución y el paradigma.

Nivel de abstracción

Llamamos nivel de abstracción al modo en que los lenguajes se alejan del código máquina y se acercan cada vez más a un lenguaje similar a los que utilizamos diariamente para comunicarnos. Cuanto más alejado esté del código máquina, de mayor nivel será el lenguaje. Dicho de otro modo, podría verse el nivel de abstracción como la cantidad de "capas" de ocultación de código máquina que hay entre el código que escribimos y el código que la máquina ejecutará en último término.

Lenguajes de bajo nivel

- **Primera generación**: solo hay un lenguaje de primera generación: el código máquina. Cadenas interminables de secuencias de 1s y 0s que conforman operaciones que la máquina puede entender sin interpretación alguna.

Lenguajes de medio nivel

- **Segunda generación**: los lenguajes de segunda generación tienen definidas unas instrucciones para realizar operaciones sencillas con datos simples o posiciones de memoria. El lenguaje clave de la segunda generación es sin duda el lenguaje ensamblador.
- Aunque en principio pertenecen a la tercera generación, y por tanto serían lenguajes de alto nivel, algunos consideran a ciertos lenguajes de programación procedimental lenguajes de medio nivel, con el fin de establecerlos en una categoría algo inferior que los lenguajes de programación orientada a objetos, que aportan una mayor abstracción.

Lenguajes de alto nivel

- **Tercera generación**: la gran mayoría de los lenguajes de programación que se utilizan hoy en día pertenecen a este nivel de abstracción, en su mayoría, los lenguajes del paradigma de programación orientada a objetos, son lenguajes de propósito general que permiten un alto nivel de abstracción y una forma de programar mucho más entendible e intuitiva, donde algunas instrucciones parecen ser una traducción directa del lenguaje humano. Por ejemplo, nos podríamos encontrar una línea de código como ésta: *IF contador = 10 THEN STOP*. No parece que esta sentencia esté muy alejada de cómo expresaríamos en nuestro propio lenguaje *Si el contador es 10, entonces para*.
- **Cuarta generación**: son lenguajes creados con un propósito específico, al ser un lenguaje tan específico permite reducir la cantidad de líneas de código que tendríamos que hacer con otros lenguajes de tercera generación mediante procedimientos específicos. Por ejemplo, si tuviésemos que resolver una ecuación en un lenguaje de tercera generación, tendríamos que crear diversos y complejos métodos para poder resolverla, mientras que un lenguaje de cuarta generación dedicado a este tipo de problemas ya tiene esas rutinas incluidas en el propio lenguaje, con lo que solo tendríamos que invocar la instrucción que realiza la operación que necesitamos.
- **Quinta generación**: también llamados lenguajes naturales, pretenden abstraer más aún el lenguaje utilizando un lenguaje natural con una base de conocimientos que produce un sistema basado en el conocimiento. Pueden establecer el problema que hay que resolver y las premisas y condiciones que hay que reunir para que la máquina lo resuelva. Este tipo de lenguajes los podemos encontrar frecuentemente en inteligencia artificial y lógica.

Forma de ejecución

Dependiendo de cómo un programa se ejecute dentro de un sistema, podríamos definir tres categorías de lenguajes:

- **Lenguajes compilados**: un programa traductor (compilador) convierte el código fuente en código objeto y otro programa (enlazador) unirá el código objeto del programa con el código objeto de las librerías necesarias para producir el programa ejecutable.
- **Lenguajes interpretados**: ejecutan las instrucciones directamente, sin que se genere código objeto, para ello es necesario un programa intérprete en el sistema operativo o en la propia máquina donde cada instrucción es interpretada y ejecutada de manera independiente y secuencial. La principal diferencia con el anterior es que se traducen a tiempo real solo las instrucciones que se utilicen en cada ejecución, en vez de interpretar todo el código, se vaya a utilizar o no.
- **Lenguajes virtuales**: los lenguajes virtuales tienen un funcionamiento muy similar al de los lenguajes compilados, pero, a diferencia de éstos, no es código objeto lo que genera el compilador, sino un *bytecode* que puede ser interpretado por cualquier arquitectura que tenga la máquina virtual que se encargará de interpretar el código *bytecode* generado para ejecutarlo en la máquina; aunque de ejecución lenta (como los interpretados), tienen la ventaja de poder ser multisistema y así un mismo código *bytecode* sería válido para cualquier máquina.

Paradigma de programación

El paradigma de programación es un enfoque particular para la construcción de software, un estilo de programación que facilita la tarea de programación o añade mayor funcionalidad al programa dependiendo del problema que haya que abordar. Todos los paradigmas de programación pertenecen a lenguajes de alto nivel, y es común que un lenguaje pueda usar más de un paradigma de programación.

- **Paradigma imperativo**: describe la programación como una secuencia de instrucciones que cambian el estado del programa, indicando cómo realizar una tarea.
- **Paradigma declarativo**: especifica o declara un conjunto de premisas y condiciones para indicar qué es lo que hay que hacer y no necesariamente cómo hay que hacerlo.
- **Paradigma procedimental**: el programa se divide en partes más pequeñas, llamadas funciones y procedimientos, que pueden comunicarse entre sí. Permite reutilizar código ya programado y solventa el problema de la programación *spaghetti*.
- **Paradigma orientado a objetos**: encapsula el estado y las operaciones en objetos, creando una estructura de clases y objetos que emula un modelo del mundo real, donde los objetos realizan acciones e interactúan con otros objetos. Permite la herencia e implementación de otras clases, pudiendo establecer *tipos* para los objetos y dejando el código más parecido al mundo real con esa abstracción conceptual.
- **Paradigma funcional**: evalúa el problema realizando funciones de manera recursiva, evita declarar datos haciendo hincapié en la composición de las funciones y en las interacciones entre ellas.
- **Paradigma lógico**: define un conjunto de reglas lógicas para ser interpretadas mediante inferencias lógicas. Permite responder preguntas planteadas al sistema para resolver problemas.

1.3 OBTENCIÓN DE CÓDIGO EJECUTABLE

Como se ha venido comentando a lo largo de todo el capítulo, nuestro programa, esté programado en el lenguaje que esté y se quiera ejecutar en la arquitectura que sea, necesita ser traducido para poder ser ejecutado (con la excepción del lenguaje máquina). Por lo que, aunque tengamos el código de nuestro programa escrito en el lenguaje de programación escogido, no podrá ser ejecutado a menos que lo traduzcamos a un idioma que nuestra máquina entienda.

1.3.1 TIPOS DE CÓDIGO (FUENTE, OBJETO Y EJECUTABLE)

El código de nuestro programa es manejado mediante programas externos comúnmente asociados al lenguaje de programación en el que está escrito nuestro programa, y a la arquitectura en donde queremos ejecutar dicho programa.

Para ello, deberemos definir los distintos tipos de código por los que pasará nuestro programa antes de ser ejecutado por el sistema.

- **Código fuente**: el código fuente de un programa informático es un conjunto de instrucciones escritas en un lenguaje de programación determinado. Es decir, es el código en el que nosotros escribimos nuestro programa.
- **Código objeto**: el código objeto es el código resultante de compilar el código fuente. Si se trata de un lenguaje de programación compilado, el código objeto será código máquina, mientras que si se trata de un lenguaje de programación virtual, será código *bytecode*.
- **Código ejecutable**: el código ejecutable es el resultado obtenido de enlazar nuestro código objeto con las librerías. Este código ya es nuestro programa ejecutable, programa que se ejecutará directamente en nuestro sistema o sobre una máquina virtual en el caso de los lenguajes de programación virtuales.

Cabe destacar que, si nos encontrásemos programando en un lenguaje de programación interpretado, nuestro programa no pasaría por el compilador y el enlazador, sino que solo tendríamos un código fuente que pasaría por un intérprete interno del sistema operativo o de la máquina que realizaría la compilación y ejecución línea a línea en tiempo real.

1.3.2 COMPILACIÓN

Aunque el proceso de obtener nuestro código ejecutable pase tanto por un compilador como por un enlazador, se suele llamar al proceso completo “compilación”.

Todo este proceso se lleva a cabo mediante dos programas: el compilador y el enlazador. Mientras que el enlazador solamente une el código objeto con las librerías, el trabajo del compilador es mucho más completo.

Fases de compilación

- **Análisis lexicográfico**: se leen de manera secuencial todos los caracteres de nuestro código fuente, buscando palabras reservadas, operaciones, caracteres de puntuación y agrupándolos todos en cadenas de caracteres que se denominan lexemas.
- **Análisis sintáctico-semántico**: agrupa todos los componentes léxicos estudiados en el análisis anterior en forma de frases gramaticales. Con el resultado del proceso del análisis sintáctico, se revisa la coherencia de las frases gramaticales, si su "significado" es correcto, si los tipos de datos son correctos, si los *arrays* tienen el tamaño y tipo adecuados, y así consecutivamente con todas las reglas semánticas de nuestro lenguaje.
- **Generación de código intermedio**: una vez finalizado el análisis, se genera una representación intermedia a modo de pseudoensamblador con el objetivo de facilitar la tarea de traducir al código objeto.
- **Optimización de código**: revisa el código pseudoensamblador generado en el paso anterior optimizándolo para que el código resultante sea más fácil y rápido de interpretar por la máquina.
- **Generación de código**: genera el código objeto de nuestro programa en un código de lenguaje máquina relocalizable, con diversas posiciones de memoria sin establecer, ya que no sabemos en qué parte de la memoria volátil se va a ejecutar nuestro programa.
- **Enlazador de librerías**: como se ha comentado anteriormente, se enlaza nuestro código objeto con las librerías necesarias, produciendo en último término nuestro código final o código ejecutable.

Aquí podemos ver una ilustración que muestra el proceso de compilación de un modo gráfico más claro.

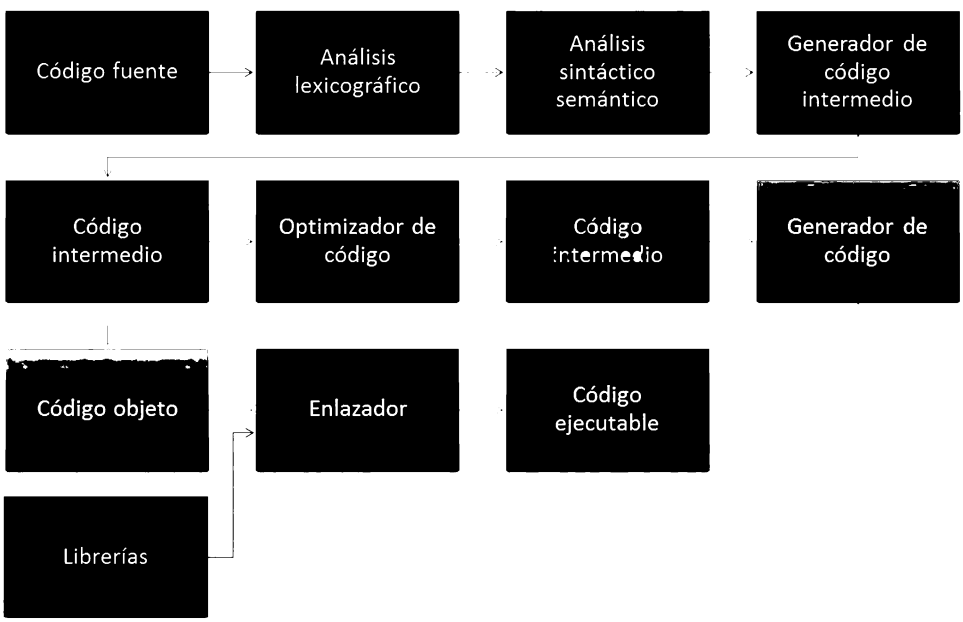


Figura 1.2. *Obtención de código ejecutable*

1.4 PROCESOS DE DESARROLLO

El desarrollo de un software o de un conjunto de aplicaciones pasa por diferentes etapas desde que se produce la necesidad de crear un software hasta que se finaliza y está listo para ser usado por un usuario. Ese conjunto de etapas en el desarrollo del software responde al concepto de ciclo de vida del programa. No en todos los programas ni en todas las ocasiones el proceso de desarrollo llevará fielmente las mismas etapas en el proceso de desarrollo; no obstante, son unas directrices muy recomendadas.

Hay más de un modelo de etapas de desarrollo, que de modo recurrente suelen ser compatibles y usadas entre sí, sin embargo vamos a estudiar uno de los modelos más extendidos y completos, el modelo en cascada.

Figura 1.3. *Modelo en cascada*

1.4.1 ANÁLISIS

La fase de análisis define los requisitos del software que hay que desarrollar. Inicialmente, esta etapa comienza con una entrevista al cliente, que establecerá lo que quiere o lo que cree que necesita, lo cual nos dará una buena idea global de lo que necesita, pero no necesariamente del todo acertada. Aunque el cliente crea que sabe lo que el software tiene que hacer, es necesaria una buena habilidad y experiencia para reconocer requisitos incompletos, ambiguos, contradictorios o incluso necesarios. Es importante que en esta etapa del proceso de desarrollo se mantenga una comunicación bilateral, aunque es frecuente encontrarse con que el cliente pretenda que dicha comunicación sea unilateral, es necesario un contraste y un consenso por ambas partes para llegar a definir los requisitos verdaderos del software. Para ello se crea un informe ERS (Especificación de Requisitos del Sistema) acompañado del diagrama de clases o de Entidad/Relación.

1.4.2 DISEÑO

En esta etapa se pretende determinar el funcionamiento de una forma global y general, sin entrar en detalles. Uno de los objetivos principales es establecer las consideraciones de los recursos del sistema, tanto físicos como lógicos. Se define por tanto el entorno que requerirá el sistema, aunque también se puede establecer en sentido contrario, es decir, diseñar el sistema en función de los recursos de los que se dispone.

En la fase de diseño se crearán los diagramas de casos de uso y de secuencia para definir la funcionalidad del sistema.

Se especificará también el formato de la información de entrada y salida, las estructuras de datos y la división modular. Con todos esos diagramas e información se obtendrá el cuaderno de carga.

1.4.3 CODIFICACIÓN

La fase más obvia en el proceso de desarrollo de software es sin duda la codificación. Es más que evidente que una vez definido el software que hay que crear haya que programarlo.

Gracias a las etapas anteriores, el programador contará con un análisis completo del sistema que hay que codificar y con una especificación de la estructura básica que se necesitará, por lo que en un principio solo habría que traducir el cuaderno de carga en el lenguaje deseado para culminar la etapa de codificación, pero esto no es siempre así, las dificultades son recurrentes mientras se modifica. Por supuesto que cuanto más exhaustivo haya sido el análisis y el diseño, la tarea será más sencilla, pero nunca está exento de necesitar un reanálisis o un rediseño al encontrar un problema al programar el software.

1.4.4 PRUEBAS

Con una doble funcionalidad, las pruebas buscan confirmar que la codificación ha sido exitosa y el software no contiene errores, a la vez que se comprueba que el software hace lo que debe hacer, que no necesariamente es lo mismo.

No es un proceso estático, y es usual realizar pruebas después de otras etapas, como la documentación. Generalmente, las pruebas realizadas posteriormente a la documentación se realizan por personal inexperto en el ámbito de las pruebas de software, con el objetivo de corroborar que la documentación sea de calidad y satisfactoria para el buen uso de la aplicación.

En general, las pruebas las realiza, idílicamente, personal diferente al que codificó la aplicación, con una amplia experiencia en programación, personas capaces de saber en qué condiciones un software puede fallar de antemano sin un análisis previo.

1.4.5 DOCUMENTACIÓN

Por norma general, la documentación que se realiza de un software tiene dos caras: la documentación disponible para el usuario y la documentación destinada al propio equipo de desarrollo.

La documentación para el usuario debe mostrar una información completa y de calidad que ilustre mediante los recursos más adecuados cómo manejar la aplicación. Una buena documentación debería permitir a un usuario cualquiera comprender el propósito y el modo de uso de la aplicación sin información previa o adicional.

Por otro lado, tenemos la documentación técnica, destinada a equipos de desarrollo, que explica el funcionamiento interno del programa, haciendo especial hincapié en explicar la codificación del programa. Se pretende con ello permitir a un equipo de desarrollo cualquiera poder entender el programa y modificarlo si fuera necesario. En casos donde el software realizado sea un servicio que pueda interoperar con otras aplicaciones, la documentación técnica hace posible que los equipos de desarrollo puedan realizar correctamente el software que trabajará con nuestra aplicación.

1.4.6 EXPLOTACIÓN

Una vez que tenemos nuestro software, hay que prepararlo para su distribución. Para ello se implementa el software en el sistema elegido o se prepara para que se implemente por sí solo de manera automática.

Cabe destacar que en caso de que nuestro software sea una versión sustitutiva de un software anterior es recomendable valorar la convivencia de sendas aplicaciones durante un proceso de adaptación.

1.4.7 MANTENIMIENTO

Son muy escasas las ocasiones en las que un software no vaya a necesitar de un mantenimiento continuado. En esta fase del desarrollo de un software se arreglan los fallos o errores que suceden cuando el programa ya ha sido implementado en un sistema y se realizan las ampliaciones necesitadas o requeridas.

Cuando el mantenimiento que hay que realizar consiste en una ampliación, el modelo en cascada suele volverse cíclico, por lo que, dependiendo de la naturaleza de la ampliación, puede que sea necesario analizar los requisitos, diseñar la ampliación, codificar la ampliación, probarla, documentarla, implementarla y, por supuesto, dar soporte de mantenimiento sobre la misma, por lo que al final este modelo es recursivo y cíclico para cada aplicación y no es un camino rígido de principio a fin.

1.5 ROLES QUE INTERACTÚAN EN EL DESARROLLO

A lo largo del proceso de desarrollo de un software deberemos realizar, como ya hemos visto anteriormente, diferentes y diversas tareas. Es por ello que el personal que interviene en el desarrollo de un software es tan diverso como las diferentes tareas que se van a realizar.

Los roles no son necesariamente rígidos y es habitual que participen en varias etapas del proceso de desarrollo.

- **Analista de sistemas**

  Uno de los roles más antiguos en el desarrollo del software. Su objetivo consiste en realizar un estudio del sistema para dirigir el proyecto en una dirección que garantice las expectativas del cliente determinando el comportamiento del software.

  Participa en la etapa de análisis.

- **Diseñador de software**

  Nace como una evolución del analista y realiza, en función del análisis de un software, el diseño de la solución que hay que desarrollar.

  Participa en la etapa de diseño.

- **Analista programador**

  Comúnmente llamado "desarrollador", domina una visión más amplia de la programación, aporta una visión general del proyecto más detallada diseñando una solución más amigable para la codificación y participando activamente en ella.

  Participa en las etapas de diseño y codificación.

- **Programador**

  Se encarga de manera exclusiva de crear el resultado del estudio realizado por analistas y diseñadores. Escribe el código fuente del software.

  Participa en la etapa de codificación.

- **Arquitecto de software**

  Es la argamasa que cohesiona el proceso de desarrollo. Conoce e investiga los *frameworks* y tecnologías revisando que todo el procedimiento se lleva a cabo de la mejor forma y con los recursos más apropiados.

  Participa en las etapas de análisis, diseño, documentación y explotación.

1.6 ARQUITECTURA DE SOFTWARE

La arquitectura de software es el diseño de nivel más alto de la estructura de un sistema, enfocándose más allá de los algoritmos y estructuras de datos. La arquitectura de software es un conjunto de decisiones que definen a nivel de diseño los componentes computacionales y la interacción entre ellos para garantizar que el proyecto llegue a buen término.

El objetivo principal de la arquitectura de software consiste en proporcionar elementos que ayuden a la toma de decisiones abstrayendo los conceptos del sistema mediante un lenguaje común. Dicho conjunto de herramientas, conceptos y elementos de abstracción se organizan en forma de patrones y modelos.

Los resultados obtenidos después de efectuar buenas prácticas de arquitectura de software deben proporcionar capas de abstracción y encapsulado, organizando el software de manera diferente dependiendo de la visión o criterio de la estructura.

1.6.1 PATRONES DE DESARROLLO

Los patrones de desarrollo, también llamados patrones de diseño, establecen los componentes de la arquitectura y la funcionalidad y comportamiento de cada uno.

Las directrices marcadas por los patrones de diseño facilitan la tarea de diseñar un software, aunque no en su totalidad. Los patrones no especifican todas las características o relaciones de los componentes en nuestro software, sino que están centrados en un ámbito específico. Cada patrón determina y especifica los aspectos de uno de los tres ámbitos principales: creacionales, estructurales y de comportamiento.

Nos podríamos encontrar con otro ámbito de patrones, los patrones de interacción, los cuales tienen el objetivo de definir diferentes directrices para la creación de interfaces, en donde prima la usabilidad. El aspecto gráfico y la usabilidad de los controles ofrecidos al usuario para manejar la aplicación son sin duda una parte sumamente importante en un software, y no es algo que se deba menospreciar. Los patrones de interacción se alejan parcialmente del ámbito de los otros patrones de diseño y no entran en su totalidad en los patrones de desarrollo.

Recalcando de nuevo, como se hizo en el párrafo anterior, la mala praxis que es descuidar la usabilidad de nuestras interfaces, nos vamos a centrar en los patrones de desarrollo pertenecientes a los ámbitos creacionales, estructurales y de comportamiento. Realizaremos una visión global de los diferentes patrones que podemos encontrar en cada ámbito mencionado.

Creacionales

Los patrones creacionales definen el modo en que se construyen los objetos, principalmente con el objetivo de encapsular la creación de los mismos haciendo que los constructores sean privados y el modo de crear una instancia sea mediante un método estático que devuelva el objeto. La característica fundamental de la programación orientada a objetos en la que recaen estas prácticas de patrones creacionales es el polimorfismo.

Fábrica abstracta

Se utiliza cuando se necesita crear diferentes objetos pertenecientes a la misma familia. Disponemos de una "factoría abstracta" que define las interfaces de las factorías y de varias "factorías concretas" que interpretan a una familia concreta. La mejor forma de ver este patrón es imaginarlo como factorías tangibles, podríamos pensar en la fábrica de coches como la fábrica abstracta y en el coche como el producto abstracto. Y podemos ver la fábrica de Gijón como una factoría concreta y el coche de Gijón como producto concreto. Aunque el producto sea el mismo, las diferentes fábricas en sus diferentes localizaciones tendrán acceso a unos recursos y proveedores diferentes.

Por tanto, tendríamos una clase con un método para crear coches abstractos, una clase por cada fábrica de coches que tendrá un método creador de coches con un parámetro de materiales. Para crear los coches utilizaremos el constructor abstracto, utilizando el parámetro de los materiales de la factoría concreta. Con ello, tenemos que para crear un coche en la fábrica de Gijón (factoría concreta) utilizaríamos el siguiente método:

Constructor virtual

EJEMPLO 1.1

CONSTRUCTOR VIRTUAL

```
Coche crearCoche() {
    FactoríaMateriales fm = new MaterialesGijon();
    Coche coche = new Coche(fm); // Uso de la factoría
    coche.montar();
    coche.pintar();
    return coche;
}
```

Se utiliza cuando de una misma factoría (utilizando el concepto del patrón anterior) se utilizan diferentes objetos complejos. Se crea un constructor abstracto por cada tipo de producto de la factoría y diferentes constructores concretos para cada producto específico. Consta de una clase producto, una clase abstracta constructor, varios constructores concretos, un director y un cliente.

Si tenemos una empresa que vende diferentes sets de ADSL, podríamos utilizar el patrón de constructor virtual de la siguiente manera:

EJEMPLO 1.2a

PRODUCTO

```
class Adsl {
    private String reuter = "";
    private Int velocidad = "";

    public void setReuter(String reuter) { this.reuter = reuter; }
    public void setVelocidad(Int velocidad) { this.velocidad = velocidad; }
}
```

EJEMPLO 1.2b

CONSTRUCTOR ABSTRACTO

```
abstract class AdslBuilder {
    protected Adsl adsl;

    public Adsl getAdsl() { return adsl; }
    public void crearNuevaAdsl() { adsl = new Adsl(); }

    public abstract void buildReuter();
    public abstract void buildVelocidad();
}
```

EJEMPLO 1.2c

CONSTRUCTOR CONCRETO

```
class BasicoAdslBuilder extends AdslBuilder {
    public void buildReuter()    { adsl.setReuter ("DLink T-504"); }
    public void buildVelocidad()    { adsl.setVelocidad(12); }
}
```

EJEMPLO 1.2d

DIRECTOR

```
class Montador {
    private AdslBuilder adslBuilder;

    public void setAdslBuilder(AdslBuilder ab) { adslBuilder = ab; }
    public Adsl getAdsl() { return adslBuilder.getAdsl(); }

    public void construirAdsl() {
       adslBuilder.crearNuevaAdsl();
       adslBuilder.buildReuter();
       adslBuilder.buildVelocidad();
    }

}
```

EJEMPLO 1.2e

CLIENTE

```
class InstalarAdsl {
    public static void main(String[] args) {
        Montador montador = new Montador();
        AdslBuilder basico_adslbuilder = new BasicoAdslBuilder ();
        AdslBuilder avanzado_adslbuilder = new AvanzadoAdslBuilder ();

        montador.setAdslBuilder(avanzado_adslbuilder);
        montador.construirAdsl();

        Adsl adsl = montador.getAdsl();
    }

}
```

Instancia única

La instancia única o *Singleton* es el patrón de diseño que nos permite asegurar que solo pueda existir una única instancia de una clase, regulando para ello el acceso al constructor. Se deberá tener en cuenta la posibilidad de multihilos y controlar dicha eventualidad mediante la exclusión mutua.

EJEMPLO 1.3

SINGLETON

```
public class Singleton
{
    private static readonly Lazy<Singleton> instance = new Lazy<Singleton>(() =>
new Singleton());

    private Singleton()
    {
    }
    public static Singleton Instance
    {
        get
        {

            return instance.Value;
        }
    }
}
public class Prueba
{
   private static void Main(string[] args)
   {
     Singleton s1 = Singleton.Instance;
     Singleton s2 = Singleton.Instance;
     if(s1==s2)
     {
       Console.WriteLine( "Es el mismo objeto");
     }
   }
}
```

Lo más importante que hay que tener en cuenta cuando se implementa el patrón es privatizar el constructor de la clase, declarar estático el método que nos devuelve la instancia y declarar al atributo de nuestra clase como privado, estático y de solo lectura. Éstas son las claves para implementar correctamente el patrón *Singleton* dentro de cualquier lenguaje.

Estructurales

Los patrones estructurales establecen las relaciones y organizaciones entre los diferentes componentes de nuestro software, resolviendo de una manera elegante diversos problemas que nos encontramos al implementar soluciones sin haber evaluado previamente todas las consecuencias y posibilidades. No hay que olvidar que el uso de patrones no es una cuestión unitaria, para solucionar un problema o eventualidad cuando surja, sino que nos plantean una serie de directrices que pueden solventar problemas futuros, hay que pensar a largo plazo y valorar siempre la expandibilidad del proyecto.

Decorador

Se utiliza este patrón cuando necesitamos añadir de manera dinámica diferentes funcionalidades a un objeto. Permite además retirar la funcionalidad si se necesita. Evitamos con este patrón definir cada funcionalidad mediante una clase heredada, utilizando para ello clases que implementan las funcionalidades necesitadas y que se asocian con la clase que necesita dicha funcionalidad.

Tenemos una empresa de creación de páginas webs y, a partir del modelo básico, tenemos diferentes funcionalidades que podemos añadir. Para facilitarnos la tarea de crear presupuestos, nos hemos puesto manos a la obra y estamos creando una aplicación para calcularlos. Partimos en un principio de un presupuesto base, correspondiente a la página web básica, y queremos añadir la posibilidad de que esa página web pueda contener un carrito de la compra y un sistema de autentificación de usuarios. Si nos pusiésemos a implementar una subclase para cada tipo de página, tendríamos cuatro clases en total: Página, PáginaConCarrito, PáginaConLogin y PáginaConCarritoYLogin. Si quisiésemos añadir en un futuro un libro de visitas y/o un foro, acabaríamos teniendo ocho y dieciséis clases respectivamente, con lo que tendríamos al final una solución insostenible.

Para solventar ese problema, creamos una subclase abstracta PaginaDecorator de la que heredan las clases CarritoDecorator y LoginDecorator. De este modo, podemos implementar tantas funcionalidades como queramos creando nuevas clases de herencia para cada funcionalidad.

La clase PaginaDecorator emula y encapsula el comportamiento de Página y utiliza composición recursiva para añadir tantos decoradores concretos como se necesiten. Con este patrón, en vez de definir los objetos con la funcionalidad, definimos las funcionalidades y luego se las añadimos al objeto que queramos.

EJEMPLO 1.4

DECORADOR

```
class Program
    {
        static void Main(string[] args)
        {
          CarritoDecorator paginaConCarrito = new CarritoDecorator(new Pagina());
            Console.WriteLine(paginaConCarrito.Calcular());

          LoginDecorator paginaConCarritoYLogin = new LoginDecorator(paginaConCarrito);
            Console.WriteLine(paginaConCarritoYLogin.Calcular());

            LoginDecorator paginaConLogin = new LoginDecorator(new Pagina());
            Console.WriteLine(paginaConLogin.Calcular());

          //En el extraño supuesto de que se necesiten dos sistemas de autentificación
          LoginDecorator paginaConDobleLogin = new LoginDecorator(paginaConLogin);
            Console.WriteLine(paginaConDobleLogin.Calcular());

            Console.Read();
        }
```

En cada decorador incluiríamos la funcionalidad (que en este caso sería añadir a la página el precio por la funcionalidad), pero podría ser cualquier otra cosa.

Objeto compuesto

Mediante el uso de una clase abstracta o de un *interface*, generaremos jerarquías de objetos que efectúen la misma acción tanto a sí mismos como a los objetos hijos que contienen.

Supongamos que tenemos una jerarquía de contenedores y párrafos, donde los contenedores pueden albergar tanto párrafos como otros contenedores. Necesitamos poder colorearlos y que hereden el color del contenedor raíz en el que se encuentren.

Para ello creamos una lista de componentes con estructura de composición recursiva en árbol. De éste modo, se crean objetos complejos formados por otros más pequeños, aplicando la acción en todos ellos y consiguiendo que se comporten como si fuesen un único objeto.

EJEMPLO 1.5a

COMPONENTE

```
public interface Componente {

   public void pintar(String color);
}
```

EJEMPLO 1.5b

COMPONENTE CONCRETO

```
public class Parrafo implements Componente{

   private String nombre;

      public Parrafo (String nombre) {
          this.nombre = nombre;
      }

   public void pintar(String color){
       System.out.println("Se pinto a " + nombre + " de color " +color);
}
```

EJEMPLO 1.5c

CONTENEDOR

```
public class Contenedor implements Componente {

   private String nombre;

   private ArrayList <componente> componentes;

   public Contenedor(String nombre) {
      setNombre(nombre);
      setComponentes(new ArrayList());
   }

   public Contenedor() {
      setNombre("");
      setComponentes(new ArrayList());
   }
   public String getNombre() {
      return nombre;
   }

   public void setNombre(String nombre) {
      this.nombre = nombre;
   }

   public void pintar(String color) {
     for (Componente c : componentes) c.pintar(color);
     System.out.println("Se pinto a " + nombre + " de color " + color);
   }

   public void add(Componente c) {
      getComponentes().add(c);
   }
}
```

Fachada

Cuando diseñamos una aplicación, es correcto, lógico y usual crear subdivisiones en el sistema para organizar nuestro código, utilizando diferentes criterios, ya sean por funcionalidad o conceptuales.

La fachada nos permite crear una interfaz que unifique el acceso a dichos subsistemas, de modo que simplifica el acceso y lo hace más fácil de usar. Podemos verlo como un nivel superior en la abstracción de nuestro código, donde para realizar complicadas o sencillas operaciones solo necesitamos tener acceso a la clase fachada para ejecutar nuestro programa.

Veamos un ejemplo conceptual sobre los usos del patrón fachada, emularemos un funcionamiento ficticio a la hora de abrir un archivo.

EJEMPLO 1.6

FACHADA

```
public class Fachada {
    private Memoria mem = new Memoria();
    private DiscoDuro dd = new DiscoDuro();
    private Procesador cpu = new Procesador();

    public void abrirArchivo(String archivo) {
        mem.cargar(dd.getDireccion(archivo));
        cpu.asignarProceso(mem.getPosUltimaCarga());
    }
 }
public class Memoria {
    public Memoria() { }
    public void cargar(String direccion) { /* ... */ }
    public MemPos getPosUltimaCarga() { /* ... */ }
}
public class DiscoDuro {
    public DiscoDuro() { }
    public void getDireccion() { /* ... */ }
}
public class Procesador {
    public Procesador() { }
    public void asignarProceso(MemPos posicion) { /* ... */ }
}
```

Analizando el uso de la fachada, podemos observar que no es un procedimiento muy distinto de lo que hemos tenido en cualquier clase o método. Es decir, combinamos y metodizamos las diferentes rutinas que tenemos que hacer para llevar a cabo una operación concreta. Agrupamos las diferentes operaciones para simplificarlas y solo necesitar de una llamada para realizar un cúmulo de acciones que conllevan la operación que necesitamos.

Gracias a la fachada, ahora si queremos abrir un archivo no necesitamos instanciar 3 clases y realizar las llamadas para abrir el archivo, solo necesitamos invocar al método *abrirArchivo* de nuestra fachada.

Comportamiento

Los patrones de comportamiento describen las comunicaciones entre objetos y clases y establecen directrices sobre cómo utilizar los objetos y clases para optimizar y organizar el comportamiento, interacción y comunicación entre ellos.

Estado

El patrón estado nos permite definir un comportamiento diferente dependiendo del estado interno en que se encuentre un objeto. Es decir, mediante la invocación del mismo método, el objeto se comporta de modo diferente dependiendo del estado.

Para crear el patrón estado, nos crearemos un contexto, que nos servirá de interfaz con el cliente, una interfaz estado para encapsular las responsabilidades del contexto en el estado en que se encuentre, y una serie de subclases estados concretos para definir los diferentes comportamientos de nuestro objeto.

Supongamos que estamos creando un software para manejar el botón de llamada a un ascensor, obviamente se comportará de forma diferente dependiendo de si se encuentra en el mismo piso desde el que se llama o si se ecuentra por debajo o por encima del piso desde el que se le llama.

EJEMPLO 1.7a

ESTADO

```
public interface Estado
 {
        void irPiso(int piso);
 }
```

EJEMPLO 1.7b

CONTEXTO

```
public class Contexto
 {
        private Estado estado;

        public void setEstado(Estado estado){
                this.estado = estado;
        }
        public Estado getEstado(){
                return estado;
        }
        public void llamar(int piso){
                estado.irPiso(piso);
        }
 }
```

EJEMPLO 1.7c

ESTADO CONCRETO

```
public class estadoAscensorAbajo implements Estado
 {
        public void irPiso(int piso){
          Ascensor ascensor = Ascensor.getAscensor();
          ascensor.subir(piso - ascensor.getPiso());
        }
 }
public class estadoAscensorArriba implements Estado
 {
        public void irPiso(int piso){
          Ascensor ascensor = Ascensor.getAscensor();
          ascensor.bajar(ascensor.getPiso() - piso);
        }
 }
public class estadoAscensorMismo implements Estado
 {
        public void irPiso(int piso){
          Ascensor ascensor = Ascensor.getAscensor();
          ascensor.abrirPuerta();
        }
 }
```

Aun teniendo en cuenta que en el ejemplo mostrado no parece extremadamente útil (ya que el estado viene especificado por una diferencia numérica), se puede observar tanto su funcionamiento como utilidad. Como añadido, podemos observar como nuestro código queda de este modo mejor organizado, y nos puede evitar utilizar condicionales innecesarios, así dejamos nuestro código más entendible y funcional, permitiendo crear estados sin meternos en complejos condicionales anidados.

Visitor

El patrón visitor pretende separar las operaciones de la estructura del objeto, para ello, se definen unas clases elemento con un método "aceptar" que recibirá al "visitante", teniendo un visitador por clase. De este modo, utilizamos las clases elemento para definir la estructura del objeto, y los visitantes para establecer los algoritmos y operaciones del objeto visitado.

Una de las particularidades más remarcables de este patrón reside en el método aceptar de un elemento, donde se define una llamada al método visitar del visitante y el argumento visitante al llamar al método aceptar en los métodos hijos de nuestra estructura de elementos.

Supongamos que tenemos una jerarquía de expresiones aritméticas sobre las que queremos definir visitantes, entre los que queremos que se encuentre un visitante que convierta la expresión aritmética en una cadena de caracteres.

EJEMPLO 1.8a

ELEMENTO

```
public abstract class Expresion {
  abstract public void aceptar(VisitanteExpresion v);
}
```

EJEMPLO 1.8b

ELEMENTO CONCRETO

```
public class Constante extends Expresion {
  public Constante(int valor) { _valor = valor; }
 public void aceptar(VisitanteExpresion v) {
    v.visitarConstante(this);
  }
  int _valor;
}
public class Variable extends Expresion {
  public Variable(String variable) { _variable = variable; }
  public void aceptar(VisitanteExpresion v) {
     v.visitarVariable(this);
  }
  String _variable;
}
public abstract class OpBinaria extends Expresion {
  public OpBinaria(Expresion izq, Expresion der) {
     _izq = izq; _der = der;
  }
  Expresion _izq, _der;
}
public class Suma extends OpBinaria {
  public Suma(Expresion izq, Expresion der) { super(izq, der); }
  public void aceptar(VisitanteExpresion v){v.visitarSuma(this);}
}
public class Mult extends OpBinaria {
  public Mult(Expresion izq, Expresion der) { super(izq, der); }
  public void aceptar(VisitanteExpresion v){v.visitarMult(this);}
}
```

EJEMPLO 1.8c

VISITANTE

```
public interface VisitanteExpresion {
  public void visitarSuma(Suma s);
  public void visitarMult(Mult m);
  public void visitarVariable(Variable v);
  public void visitarConstante(Constante c);
}
```

EJEMPLO 1.8d

VISITANTE CONCRETO

```
public class ExpressionToString implements VisitanteExpresion {
  public void visitarVariable(Variable v) {
        _resultado = v._variable;
  }
  public void visitarConstante(Constante c) {
         _resultado = String.valueOf(c._valor);
  }
  private void visitarOpBinaria(OpBinaria op, String pOperacion){
          op._izq.aceptar(this);
          String pIzq = obtenerResultado();

          op._der.aceptar(this);
          String pDer = obtenerResultado();

          _resultado = "(" + pIzq + pOperacion + pDer + ")";
  }
  public void visitarSuma(Suma s) {
        visitarOpBinaria(s, "+");
  }
  public void visitarMult(Mult m) {
        visitarOpBinaria(m, "*");
  }
  public String obtenerResultado() {
        return _resultado;
  }
  private String _resultado;
}
```

Iterador

El patrón iterador establece una interfaz, cuyos métodos permiten acceder a un conjunto de objetos de una colección. Los métodos necesarios para recorrer la colección pueden variar dependiendo de las necesidades que tengamos y de lo que necesitemos hacer, pero es habitual crear o necesitar métodos que permitan acceder al primer elemento, obtener el elemento actual y saber si hemos llegado al final de la colección.

Para ello nos creamos un iterador y un agregado, donde el iterador es una interfaz que permite recorrer el agregado encapsulando las operaciones necesarias y ocultándolas al cliente, algo parecido a lo que hacíamos con el patrón fachada.

EJEMPLO 1.9a

AGREGADO

```
public class Coleccion {
    public int[] _datos;

    public Coleccion(int valores){
        _datos = new int[valores];
        for (int i = 0; i < _datos.length; i++){
            _datos[i] = 0;
        }
    }
    public int getValor(int pos){
        return _datos[pos];
    }
    public void setValor(int pos, int valor){
        _datos[pos] = valor;
    }
    public int dimension(){
        return _datos.length;
    }
 public IteradorVector iterador(){
        return new IteradorColeccion(this);
    }
}
```

EJEMPLO 1.9b

ITERADOR

```
public class IteradorColeccion{
    private int[] _vector;
    private int _posicion;

    public IteradorColeccion(Coleccion vector) {
        _vector = vector._datos;
        _posicion = 0;
    }
    public boolean hayMas(){
        if (_posicion < _vector.length)
                    return true;
            else
                    return false;
    }
    public Object siguiente(){
        int valor = _vector[_posicion];
        _posicion++;
        return valor;
    }
}
```

Antipatrones

Los antipatrones son la contraparte de los patrones que hemos estado viendo anteriormente. Es decir, si los patrones son buenas prácticas de desarrollo de software, los antipatrones son justamente lo contrario, malas prácticas en el desarrollo de software.

La definición más acertada del antipatrón es la aplicación de un patrón de software en un contexto equivocado, aunque el uso que damos a la palabra "antipatrón" no se detiene ahí, y habitualmente la encontramos usada como "malos patrones" o "malas prácticas de desarrollo". Al final, todo se reduce al mismo concepto: situaciones o prácticas que no tenemos que hacer o tenemos que evitar a la hora de desarrollar un software.

La cantidad de antipatrones en los que podemos caer, o nos podemos encontrar, es sencillamente abrumadora, en esta sección vamos a ver una selección de los antipatrones más "populares" o habituales.

Código spaghetti

El código *spaghetti* era muy habitual encontrarlo en los inicios de la programación, antes de que se definiese el concepto de programación orientada a objetos, donde el código no estaba metodizado y era un absoluto caos. Cualquier cambio o movimiento en el código desmoronaba toda la aplicación, llena de saltos constantes a modo de llamadas o bucles. Se le dio este nombre por el dibujo resultante de tomar un lápiz y dibujar líneas mostrando la vida del programa, con tantos saltos a diferentes fragmentos de código, las líneas se enmarañaban y cruzaban una y otra vez, dejándonos un garabato con aspecto similar al de un plato de *spaghetti*. Hoy en día es habitual encontrarlo en scripts incrustados en el código, todos ellos carentes y necesitados de refactorizar.

Flujo de lava

Grandes cantidades de código desordenado, módulos y añadidos ingentes que rompen la estructura natural del software; documentación paupérrima o código abandonado serían varios de los indicadores o "síntomas" que se verían en un código "enfermado" con este antipatrón. Este antipatrón es más habitual encontrarlo en software codificado bajo una mala gestión, seguramente no sea un problema fruto de un programador descuidado, sino de una mala gestión por el jefe de proyecto y por las necesidades del cliente, que hace que tengamos fragmentos de código sin terminar y anexos a la aplicación fruto de los cambios recurrentes en la urgencia, prioridades, preferencias y necesidades del cliente.

Martillo dorado / Varita mágica

El apego injustificado e irresponsable a un paradigma, a un lenguaje o a un *framework* concreto para solucionar todos los problemas puede ocasionar infinidad de problemas. Las plataformas de trabajo, los lenguajes y los paradigmas tienen diferentes capacidades y limitaciones que nos ofrecen funcionalidades diferentes. Por lo que nuestras preferencias pueden hacernos escoger unos recursos inapropiados para el software que necesitamos desarrollar.

Reinventar la rueda

Este antipatrón aparece cuando implantamos soluciones a problemas que ya existen en el *framework* contra el que trabajamos. Al reimplementar esos componentes ya existentes y no reutilizar el código, no solo perdemos tiempo, sino que el software se vuelve más denso de forma innecesaria y en ocasiones podemos empeorar la cohesión del código dentro de la misma plataforma. Puede venir por un desconocimiento del *framework* donde se trabaja o por la injustificada necesidad de personalizar los componentes que utilizamos.

Infierno de las dependencias

En contrapartida al anterior antipatrón, depender de manera abusiva de las librerías y componentes de un entorno de desarrollo o plataforma puede ocasionarnos muchos problemas derivados de las diferentes versiones de las dependencias.

Manejo de excepciones inútil

Si establecemos condicionales con el fin de evitar que surjan excepciones para lanzar manualmente una excepción, estamos utilizando un control de excepciones problemático. De hecho, ni siquiera es un auténtico control de excepciones, ya que es solo fruto de pensar que dicha excepción se puede producir y lanzamos la excepción que necesitamos, además, por otro lado, estamos creando código innecesario, ya que el propio control de excepciones del lenguaje nos ofrece esta funcionalidad sin necesidad de crear condicionales para ello.

Cadenas mágicas

La utilización de cadenas de caracteres explícitas no es una práctica recomendada; en ocasiones, y por requisitos ajenos a nuestro código (ya sea por el *framework*, librería o similar), incluimos cadenas de caracteres a la hora de realizar llamadas o comparaciones de manera recurrente. El problema más obvio que apreciamos en esta mala práctica es que necesitamos modificar y recompilar el código en caso de que necesitemos cambiar la(s) cadena(s) de caracteres.

Copiar & Pegar

Siempre que a la hora de crear una nueva clase o método copiemos y peguemos código para ello, debemos tener en cuenta que, sin excepción, estamos haciendo algo mal. Duplicar el código en vez de reutilizarlo nunca es una práctica adecuada, la necesidad de duplicar código es síntoma de que nuestro código necesita ser refactorizado, por ejemplo con una refactorización de "Extraer método" o pensar un modo de hacer y modificar el método para que sea accesible desde los sitios donde necesitas invocarlo.

ACTIVIDADES 1.2

- ¿Qué patrones podrían usarse para construir una aplicación que se encargase de administrar y crear los extractos de un banco que tiene varias sucursales en una misma ciudad?
- Relacione los patrones y antipatrones vistos especificando qué patrones podrían sustituir al uso de algún antipatrón.

1.6.2 DESARROLLO EN TRES CAPAS

El desarrollo en capas nace de la necesidad de separar la lógica de la aplicación del diseño, separando a su vez los datos de la presentación al usuario.

Para solventar esa necesidad, se ideó el desarrollo en 3 capas, que separa la lógica de negocio, el acceso a datos y la presentación al usuario en tres capas que pueden tener cada una tantos niveles como se quiera o necesite.

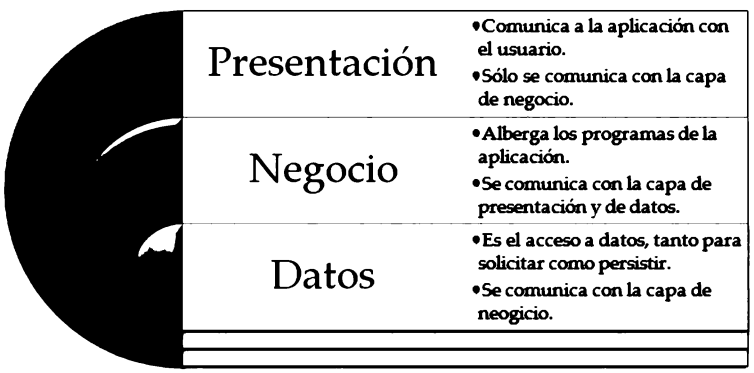


Figura 1.4. *Desarrollo en 3 capas*

El desarrollo por capas no solo nos mejora y facilita la estructura de nuestro propio software, sino que nos aporta la posibilidad de interoperar con otros sistemas ajenos a nuestra aplicación. Por ejemplo, podríamos necesitar acceder y utilizar datos contenidos tanto en nuestra propia base de datos, como en la base de datos de un banco, para ello utilizaríamos la capa de datos, en donde accederíamos a nuestro gestor de base de datos y al servicio que nos ofrezca

el banco para solicitar dichos datos. Esto podría hacerse sin necesitar un desarrollo en tres capas, pero la principal ventaja (aparte de la encapsulación y ocultación de código y datos entre las capas) que nos aporta reside en evitar modificar la lógica de negocio por necesitar acceder a diferentes datos, todo está perfectamente estructurado y nos permite modificar las fuentes y el modo en que accedemos a los datos de los programas que trabajan con ellos.

Modelo vista controlador

Dentro del desarrollo por capas, encontramos diferentes modelos de software, uno de ellos es el MVC (Modelo Vista Controlador).

El MVC define tres componentes para las capas del desarrollo del software, organiza el código mediante unas directrices específicas utilizando un criterio basado en la funcionalidad y no en las características del componente en sí mismo.

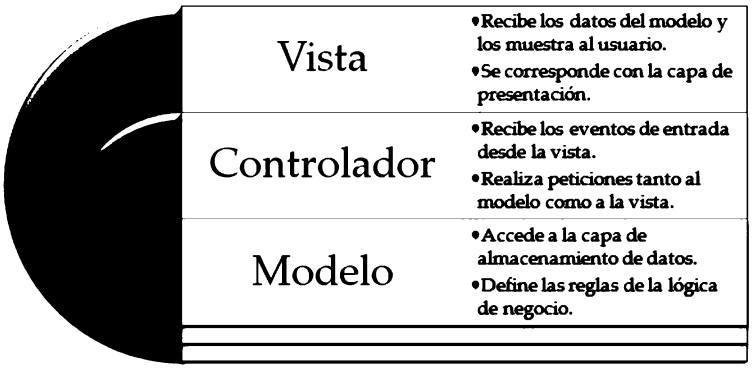


Figura 1.5. *Modelo Vista Controlador*

A la hora de mostrar los datos de modelo en las vistas, se suele establecer una serie de "bindings" que enlacen diferentes componentes de la vista a propiedades y campos de las entidades de los datos a los que tiene acceso el modelo.

Modelo vista vistamodelo

EL MVVM parte de un concepto muy similar al modelo MVC, tanto, que no resulta extraño pensar que es una ampliación o modificación al MVC. De hecho, muchas *frameworks* actuales ofrecen el uso del MVVM a través del MVC *framework* añadiéndole un ViewModel. Siendo objetivos, no es una visión muy apartada de la realidad, ya que en esencia parten de la misma necesidad y concepto.

A diferencia del MVC, la vista del MVVM es un observador que se actualiza cuando cambia la información contenida en el VistaModelo. El componente VistaModelo en MVVM contiene a su vez un controlador al igual que en el MVC, y además utiliza un VistaModelo que actúa como un modelo virtual y personalizado que contiene la información necesaria para mostrarla en la vista. Es decir, al igual que en el MVC, los eventos de la vista son recogidos por el controlador, pero a diferencia del MVC los datos de la vista son obtenidos y actualizados a través del VistaModelo, ocultando así al modelo de la vista, dejando al modelo como una mera representación de las entidades para la persistencia de los datos.

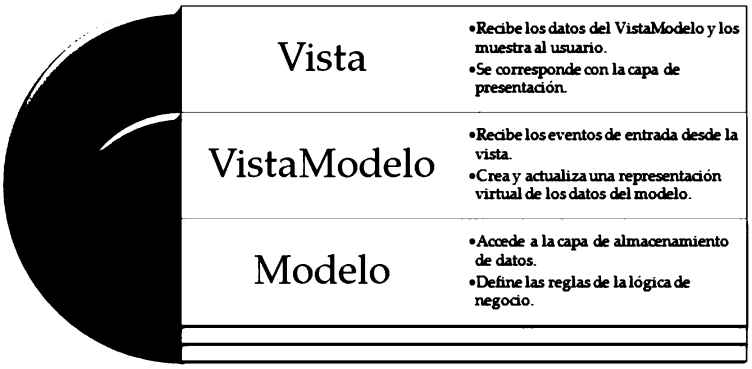


Figura 1.6. *Modelo Vista VistaModelo*

Mediante el uso del VistaModelo, hemos logrado ocultar el modelo a la vista, además de evitar la necesidad de crear "bindings" manuales entre la vista y el modelo, ya que el propio VistaModelo se puede "bindear" directamente.

ACTIVIDADES 1.3

- Relacione los dos modelos de desarrollo en 3 capas con los diferentes patrones, especificando qué patrones podrían incluirse en las diferentes capas de los modelos.

RESUMEN DEL CAPÍTULO

En este tema nos hemos introducido en el mundo del desarrollo de software. Esta primera toma de contacto nos ofrecerá una excelente base sobre la que tratar todos los temas de desarrollo y diseño de proyectos y estrategias de desarrollo de software.

Se han incluido conceptos y técnicas que el alumno no llegará a dominar hasta que profundice en sus conocimientos de programación orientada a objetos, pero aporta un excelente complemento conceptual que se puede practicar por medio del pseudocódigo.

El alumno debería ser capaz de identificar el lenguaje que necesita utilizar para desarrollar un software y reconocer los patrones necesarios para llevar dicha tarea a cabo. Más adelante podrá combinar el conocimiento de antipatrones mencionados en este tema con los "malos olores" de la refactorización, mejorando la calidad de software y consiguiendo una mayor calidad tanto en la aplicación desarrollada como en el código de la misma.

Se recomienda que una vez profundizados y consolidados los conocimientos de programación se vuelva a realizar una lectura de este tema, esto permitirá al alumno comprender mucho mejor los conceptos y técnicas de arquitectura de software.

EJERCICIOS PROPUESTOS

- **1.** Según el esquema de la máquina von Neumann. ¿Cuáles serían las microinstrucciones que se ejecutarían para multiplicar por tres el valor de R1, sumárselo a R2 y guardar el resultado en R3?
- **2.** La pizzería Borde Exterior tiene establecimientos y almacenes en Nueva York, Oslo y Lepe. ¿Qué patrón de desarrollo usaríamos para crear las diferentes pizzas que pueden realizar en dichos establecimientos? Impleméntelo.
- **3.** La posada El Poni Pisador ofrece diferentes servicios a sus clientes, tales como *spa*, piscina y servicios de habitaciones, además de la propia estancia en la habitación. ¿Qué patrón utilizaríamos para desarrollar una aplicación que nos calculase el total a pagar por el cliente? Impleméntelo.

4. Dada la siguiente estrucura, indique a qué patrón corresponde y rellene el código especificado en los comentarios.

CÓDIGO EJERCICIO 4

```
public class Interfaz {
    private LibreriaLibros libros = new LibreriaLibros();
    private LibreriaVideo videos = new LibreriaVideo();
    private LibreriaMusica musica = new LibreriaMusica();

    public void buscarLibros() {
        libros.buscarLibros();
    }

    public void buscarMusica() {
        musica.buscarMusica();
    }

public void buscarVideo() {
        videos.buscarVideo();
    }
}

public class LibreriaLibros {
    public LibreriaLibros() { }
    public void buscarLibros() { /* ... */ }
}

public class LibreriaVideo {

    public LibreriaVideo() { }
    public void buscarVideo() { /* ... */ }
}

public class LibreriaMusica {

    public LibreriaMusica() { }
    public void buscarMusica() { /* ... */ }
}

public class Cliente {

    public static void main(String args[]){
        //Rellenar
    }
}
```

5. Seleccione los diferentes antipatrones que aparecen en el siguiente código. Explique cómo podría evitarlos.

CÓDIGO EJERCICIO 5

```
public class Cliente {
    public static void main(String args[]){
   Consola consola = new Consola();
   Bucle:
   If (consola.leer()!=”*”)
       Goto Bucle;
   Else
       System.out.println(“¡Bonito asterisco!”);
    }
}
```

6. Cree el pseudocódigo necesario para crear una aplicación basada en MVVM que genere movimientos bancarios de ingreso, retirada y traspasos de efectivo.

7. ¿Qué patrón se está utilizando en el siguiente código?

CÓDIGO EJERCICIO 7

```
public class Test extends JFrame {
   public static void main(String args[]) {
      Test frame = new Test();
      frame.setTitle(“Swing Actions”);
      frame.setSize(500, 400);
      frame.setLocation(400, 200);
      frame.show();
   }
 public Test() {
      JMenuBar mb = new JMenuBar();
      JMenu fileMenu = new JMenu(“File”);
      fileMenu.add(new ShowDialogAction());
      fileMenu.add(new ExitAction());
      mb.add(fileMenu);
setJMenuBar(mb);
   }
}
class ShowDialogAction extends AbstractAction {
   public ShowDialogAction(){
   super(“show dialog”);
   }
 public void actionPerformed(ActionEvent e) {
      JOptionPane.showMessageDialog((Component)e.getSource(),
                                “An action generated this dialog”);
   }
}
class ExitAction extends AbstractAction {
   public ExitAction() {
      super(“exit”);
   }
 public void actionPerformed(ActionEvent e) {
      System.exit(0);
   }
}
```

TEST DE CONOCIMIENTOS

1 ¿Ante qué tipo de lenguaje estamos si procesa y traduce las instrucciones en tiempo de ejecución?
- a) De tercera generación.
- b) Interpretado.
- c) Compilado.
- d) Todos los anteriores.

2 El código objeto puede ser:
- a) Código máquina.
- b) *Bytecode*.
- c) Las respuestas *a* y *b* son correctas.
- d) Ninguna de las anteriores es correcta.

3 ¿Qué se hace durante el proceso de explotación de un software?
- a) Desplegar o distribuir nuestro software en el sistema.
- b) Comprobar el funcionamiento y seguridad del software.
- c) Asegurar y mantener las necesidades del software una vez distribuido.
- d) Todas las anteriores son correctas.

4 Un analista programador se encarga de:
- a) Codificar el diseño de un software en el lenguaje deseado.
- b) Diseñar o mejorar el diseño de un proyecto de software.
- c) Las respuestas *a* y *b* son correctas.
- d) Todas las anteriores son correctas.

5 ¿Qué antipatrones podemos encontrarnos al diseñar y codificar una aplicación mediante el desarrollo en tres capas MVVM?
- a) Ninguno, el desarrollo en 3 capas evita el uso de antipatrones.
- b) Cualquiera.
- c) Solo los antipatrones relacionados con la codificación.
- d) Ninguno de los anteriores.

6 Si creamos una clase cuyo único propósito sea el de declarar unos métodos de acceso más accesibles para la clase cliente… ¿qué estamos haciendo?
- a) Aumentar el nivel de abstracción de nuestro código.
- b) Una clase fachada.
- c) Todas las respuestas anteriores son correctas.
- d) Ninguna de las anteriores es correcta.

7 ¿Qué elemento se encarga de observar los cambios e interacciones en la interfaz de usuario?
- a) Modelo.
- b) Vista.
- c) VistaModelo.
- d) Controlador.

2 Instalación y uso de entornos de desarrollo

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer las características y funcionalidades de un entorno de desarrollo.
- ✓ Comprender el proceso de selección de un entorno de desarrollo que se ajuste a nuestras necesidades.
- ✓ Apreciar la necesidad de configurar y personalizar un entorno de desarrollo aumentando la comodidad y productividad a la hora de desarrollar software.
- ✓ Introducción al IDE Visual Studio 2010 Ultimate.

2.1 CARACTERÍSTICAS

Definición de entorno de desarrollo: "*Un entorno de desarrollo integrado o IDE (Integrated Development Environment) es un programa informático que tiene el objetivo de asistir al programador en la tarea de diseñar y codificar un software mediante la inclusión de múltiples herramientas destinadas para dicha tarea*".

Cada entorno de desarrollo (IDE a partir de ahora) tiene unas características y funcionalidades específicas que lo definen. No obstante, todos mantienen unos componentes comunes.

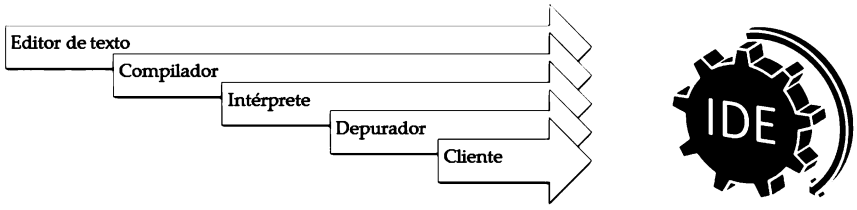


Figura 2.1. *Componentes básicos de un entorno de desarrollo*

2.1.1 EXTENSIONES Y HERRAMIENTAS

Dentro de un IDE tenemos a nuestra disposición un sinfín de herramientas con las que trabajar. Podríamos utilizar simplemente un editor de texto y un compilador (en caso de que se tratase de un lenguaje no interpretado), pero nos perderíamos muchas de las facilidades y herramientas que nos facilita el IDE.

Hoy en día, existen una gran cantidad de editores de texto que incluyen una de las funciones más triviales pero increíblemente útiles de un entorno de desarrollo: el coloreado de sintaxis. Se identifican las palabras reservadas y elementos clave del lenguaje coloreándolos para tener una mejor visión del código.

EJEMPLO 2.1a

CON COLOREADO DE SINTAXIS

```
double getVelocidad(){                                          (color azul)
// Hola! Soy un comentario!                                     (color verde)
  switch (_tipo){                                               (color azul)
    case EUROPEA:                                               (color azul)
           return getBVelocidadBase();
      case NORUEGO_AZUL:                                        (color azul)
           return (_esMoteado) ? 0 : getVelocidadBase(_voltaje);
   }
   throw new RuntimeException("Debería ser inalcanzable");      (colores azul y rojo)
}
```

EJEMPLO 2.1b

SIN COLOREADO DE SINTAXIS

```
double getVelocidad(){
// Hola! Soy un comentario!
  switch (_tipo){
    case EUROPEA:
           return getBVelocidadBase();
      case NORUEGO_AZUL:
           return (_esMoteado) ? 0 : getVelocidadBase(_voltaje);
   }
   throw new RuntimeException("Debería ser inalcanzable");
}
```

Como podemos apreciar, esta característica tan sencilla nos mejora la visión y el entendimiento del código con un simple vistazo. Obviamente, ésta no es la funcionalidad más importante que nos ofrece el IDE. Realmente no existe "la mejor herramienta del IDE", es una cuestión más subjetiva que depende del programador y de su forma de codificar un software.

Pese a ello, una de las características más importantes para cualquier programador es el autocompletado de código. Cada vez que comenzamos a escribir una palabra reservada, aparece un listado de sugerencias por el que podemos navegar para elegir la que queremos o seguir escribiendo mientras se filtra el listado. También resulta útil al mostrar los métodos y propiedades de un objeto o clase cuando escribimos el "." para acceder a ellos.

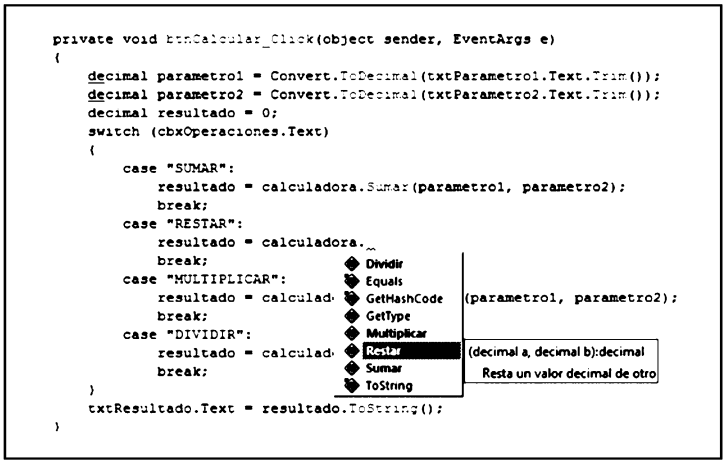


Figura 2.2. *Autocompletar / IntelliSense*

También nos puede crear las estructuras de clases o instrucciones de manera automática mediante *snippets*, por ejemplo, en Visual Studio si se escribe "for" y se presiona dos veces tabulador, se obtendrá la estructura de la instrucción *for*.

EJEMPLO 2.2

INSTRUCCIÓN FOR

```
for (int i = 0; i < length; i++)
{

}
```

Nos ofrecen herramientas de refactorización, una ejecución en depuración y muchas opciones más con el objetivo de facilitar y acortar el tiempo dedicado al desarrollo y codificación de software.

No todas las herramientas están integradas en el IDE, muchas de ellas las podemos incluir mediante extensiones que añaden, modifican y mejoran el IDE.

2.1.2 PERSONALIZACIÓN Y CONFIGURACIÓN

Los IDE son altamente configurables, ya que las necesidades de cada programador o grupo de trabajo pueden ser diferentes, el objetivo es ofrecer al usuario una aplicación amigable con la que trabajar, por lo que poder personalizar y configurar la herramienta es un aspecto verdaderamente importante.

La configuración del IDE permite entre otras cosas añadir y modificar las barras de herramientas, pudiendo crear comandos personalizados y atajos de teclado para cada una de ellas. Estableciendo el posicionamiento de las ventanas y barras conjuntamente con los atajos de teclado podremos mejorar sumamente nuestro rendimiento y aprovechar con mayor comodidad todas las funciones del IDE.

Las configuraciones de depuración y compilación de proyectos son una práctica recurrente en el desarrollo de aplicaciones, por lo que resulta extremadamente efectivo poder configurarlas al gusto, obteniendo mejores resultados al poder manejar las interrupciones de las excepciones o desenredar la pila de llamadas.

2.2 CRITERIOS DE ELECCIÓN DE UN IDE

Ya sabemos qué es un entorno de desarrollo integrado y qué particularidades y funcionalidades nos puede llegar a ofrecer, ahora solo necesitamos saber cuál elegir. Para poder elegir correctamente el IDE con el que vamos a trabajar, necesitamos saber las características que buscamos en él, si satisface nuestras necesidades y si nosotros mismos cumplimos los requisitos del IDE más allá de los requerimientos técnicos de hardware.

2.2.1 SISTEMA OPERATIVO

Sin lugar a dudas, uno de los criterios más restrictivos a la hora de seleccionar nuestro entorno de trabajo es saber en qué sistema operativo vamos a trabajar y, más importante aún, para qué sistema operativo vamos a desarrollar nuestro software.

Aunque hoy en día con los equipos modernos no es muy complicado virtualizar o emular el sistema operativo o el IDE que queremos usar, siempre es aconsejable no utilizar esos métodos si no es como último recurso. Siempre y cuando nuestro software no vaya a ser ejecutado mediante una máquina virtual, estaremos desarrollando para un sistema operativo concreto, por lo que si estamos desarrollando aplicaciones para Linux, resultaría bastante inusual desarrollar la aplicación en Windows para ello. Esto realmente no se debe a una restricción inherente al IDE, sino al compilador que tiene el IDE integrado. Si recordamos el proceso de obtención de código ejecutable del capítulo anterior, veremos que el compilador se encarga de traducir nuestro código fuente en código objeto, que es el que ejecutará el sistema. Este problema es fácilmente salvable compilando nuestro código fuente en un compilador de otro sistema operativo (siempre y cuando exista), por lo que dependiendo de nuestras necesidades pudiera ser un problema menor.

2.2.2 LENGUAJE DE PROGRAMACIÓN Y FRAMEWORK

Como comentamos anteriormente, un IDE puede soportar uno o varios lenguajes de programación, por lo que saber en qué lenguaje de programación vamos a codificar nuestro software y qué lenguajes nos ofrecen los distintos IDE es una información valiosa que hay que tener en cuenta.

Este criterio va de la mano con el sistema operativo, ya que si quisiéramos desarrollar en Visual Basic bajo un sistema operativo Linux no sería Visual Studio nuestra opción, sino que tendríamos que utilizar Gambas.

Lo mismo ocurre con las plataformas de trabajo, también llamadas *framework*, no solo depende de la plataforma de trabajo sobre la que vayamos a trabajar, también necesitamos saber bajo qué sistema operativo vamos a desarrollarla o ejecutarla. Siguiendo con el ejemplo anterior, si fuésemos a desarrollar con Visual Basic con la plataforma .NET bajo Linux, tendríamos que usar Mono Develop en lugar de Visual Studio.

2.2.3 HERRAMIENTAS Y DISPONIBILIDAD

Las diferentes herramientas de las que disponen los IDE son el último criterio de selección, seguramente nos encontremos con varios IDE que cumplen los requisitos de lenguaje y sistema operativo, pero no todos tienen las mismas funciones, por lo que saber cuáles son esas herramientas es un dato sumamente importante en nuestra decisión.

En ocasiones pueden ser restrictivos ya no solo por tus propias preferencias, sino por trabajar de manera colaborativa y el modo de utilizar e interpretar diferentes códigos entre diferentes IDE; por ejemplo, si nuestros compañeros de trabajo están utilizando el Team Server Foundation (TFS) como sistema de control de versiones, nosotros deberemos usarlo también, y el único modo de hacerlo es mediante Visual Studio; por otro lado, si nuestros compañeros están trabajando en Java con un sistema de control de versiones Subversion (SVN), podríamos usar indistintamente Netbeans, Eclipse o InteliJ DEA entre otros, ya que SVN está disponible para todos los IDE.

Fuera del marco de trabajo colaborativo, también tendremos nuestras preferencias y necesidades, por lo que, si necesitamos tener una funcionalidad para crear archivos de ayuda y documentación, deberíamos buscar un IDE que tuviese dicha funcionalidad o, en su defecto, que hubiese una extensión o *plugin* para ese IDE que aporte la funcionalidad deseada.

Podríamos también ir más allá de las meras funcionalidades añadidas e irnos a un ámbito mucho más centrado en el propio software, como podría ser la interfaz de usuario, los IDE pueden incluir sus propios y específicos controles que mejoran la interfaz y aportan mayor funcionalidad y usabilidad a nuestros formularios y aplicaciones. También podríamos ver este criterio desde un punto de vista más destinado a la codificación y pensar en qué refactorizaciones automáticas nos podemos encontrar entre los IDE a la hora de elegirlo.

El mayor problema que nos podemos encontrar no es ya elegir incorrectamente, sino no saber qué funcionalidades podría estar otorgando un IDE u otro, ya que podríamos no conocer todas las funcionalidades que puede llegar a ofrecer y, por tanto, deberemos invertir una gran cantidad de tiempo investigando y documentándonos a fondo sobre los IDE potenciales que podemos usar o invertir ese tiempo en probarlos de manera empírica.

A simple vista, parecen demasiados factores a tener en cuenta y una operación larga, tediosa y complicada, no debemos agobiarnos por esta cuestión, ya que, al igual que con los lenguajes de programación, no podemos pretender empezar sabiendo todas las posibilidades que nos ofrecen y tenemos que ir aprendiéndolas de modo incremental en función de nuestra experiencia.

Para que se vea de un modo más claro, observemos una comparativa sencilla sobre los tres IDE más populares para Java.

	NetBeans	Eclipse	IntelliJ DEA
Refactor-izaciones	6	23	29
Modelado inverso UML	Sí	No	Sí
Interpretar IGU	No	Sí	Sí

Figura 2.3. *Comparación IDE Java*

Aún nos quedaría además tratar el asunto de la disponibilidad, el cual, una vez comprobados todos los criterios de selección mencionados, puede desbaratar nuestra toma de decisiones. El aspecto más restrictivo de la disponibilidad reside en el precio de la aplicación, dependiendo de nuestro presupuesto podremos acceder a diferentes IDE.

Aunque nuestro presupuesto sea escaso, existen soluciones muy económicas e incluso gratuitas, por lo que, aunque no podamos tener el IDE que queremos, tendremos alternativas más que suficientes a nuestra disposición que cumplirán sin duda al menos la mayoría de los criterios que necesitamos.

2.3 USO BÁSICO DE UN IDE

Los entornos de desarrollo integrado son por definición una herramienta de desarrollo de software, por lo que la respuesta a la pregunta de cuál sería su uso o funcionalidad básica resulta bastante obvia: desarrollar software. No obstante, la tarea de un IDE no se queda ahí, ya que nos permite realizar una infinidad de operaciones que no podríamos realizar de otro modo. Además, si la tarea de un IDE solo fuese desarrollar aplicaciones, entonces un editor de texto y un compilador harían la misma función. La particularidad, como hemos visto a lo largo del capítulo, de un IDE reside en la cantidad y calidad de las funcionalidades añadidas que ofrece al desarrollador, por medio de herramientas integradas o externas.

Muchas de las herramientas más habituales que solemos usar conjuntamente con los IDE también se encuentran disponibles fuera de ellos, como podría ser el caso de una aplicación para crear modelados y diagramas o aplicaciones para automatizar las pruebas unitarias. En esencia, todas esas aplicaciones pueden o podrían estar disponibles sin necesidad de un entorno de desarrollo, o mejor dicho, sin que fuese un entorno de desarrollo integrado, ya que si utilizamos un editor de texto, un compilador, un programa de refactorización, un cliente de trabajo colaborativo con acceso a repositorios y un analizador de código, entonces todas esas aplicaciones que utilizamos por separado serían en concepto un entorno de desarrollo, aunque no estuviesen empaquetadas en una misma aplicación.

2.3.1 EDICIÓN DE PROGRAMAS Y GENERACIÓN DE EJECUTABLES

La necesidad básica que todo IDE debe cubrir es la creación o edición de programas y convertir ese código fuente en código ejecutable. Inicialmente, sin un IDE la operación tampoco sería tan complicada, tal y como hemos visto en el capítulo 1, compilaríamos nuestro código fuente y utilizaríamos un enlazador para combinar ese código objeto con nuestras librerías, obteniendo como resultado nuestro programa ejecutable. Un IDE realiza esa operación de manera conjunta y compacta. Gracias a un gestor de proyectos podemos ajustar las dependencias de cada sección de nuestro programa y todas las necesidades y opciones de compilación que queramos.

Los IDE, además, suelen ofrecer una funcionalidad añadida, ya que permiten ejecutar de manera virtual el programa que se está codificando en cualquier momento (siempre y cuando no se produzcan errores durante la compilación), de ese modo permiten comprobar la funcionalidad del programa sin tener que crear una publicación o despliegue de la aplicación para probar cualquier cambio o añadido.

2.3.2 DESARROLLO COLABORATIVO

El desarrollo colaborativo hace referencia al proceso de desarrollo de un software de manera descentralizada y distribuida, donde los desarrolladores no necesitan conocerse, tener el mismo (o algún) jefe, ni hablar el mismo idioma, pero trabajan en el mismo proyecto. Este modelo de desarrollo es muy habitual en proyectos de software libre, aunque no es restrictivo, cualquier empresa o grupo de desarrollo puede utilizar el desarrollo colaborativo o las herramientas que se utilizan en él. Para poder llevar un control del código realizado desde tantas y diversas fuentes, se utilizan los controles de versiones.

Los programas de control de versiones son aplicaciones que constan de servidor y cliente, donde en la parte del servidor se crean repositorios para que los clientes puedan descargar y subir código. Son herramientas asíncronas que permiten controlar y gestionar las fuentes y versiones del código del repositorio.

Podemos observar el funcionamiento y uso de un control de versiones con el siguiente diagrama:

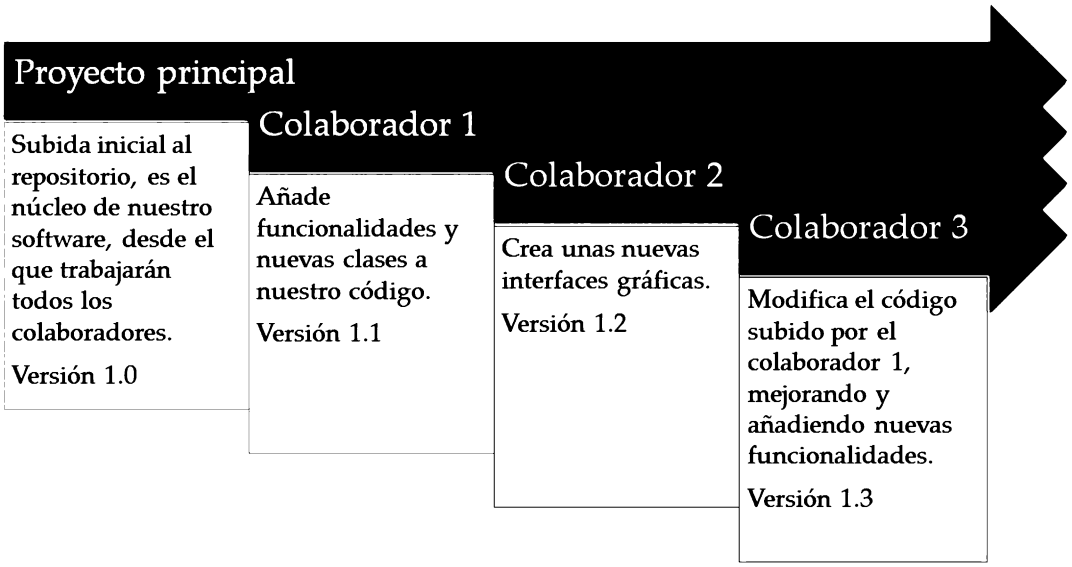


Figura 2.4. *Desarrollo colaborativo*

Como vemos en el diagrama, el proceso es continuo y acumulativo, es decir, el proyecto parte del mismo núcleo y se va modificando con el paso del tiempo, los siguientes desarrolladores no trabajan sobre el proyecto principal inicial, sino sobre el trabajo realizado por los anteriores colaboradores, es decir, el colaborador 2 trabaja sobre el código resultante de la intervención del colaborador 1, y así consecutivamente.

Podemos además elegir sobre qué versión trabajar, es decir, qué versión descargarnos en nuestro PC para trabajar sobre ella; por ejemplo, el colaborador 3 podría haber elegido trabajar sobre la versión 1.2 en vez de sobre la 1.1.

Todas estas operaciones se pueden realizar desde aplicaciones cliente externas o integradas, pero es en las herramientas integradas donde realmente está la magia, ya que, gracias a las herramientas del IDE, podemos hacer un uso mucho más rápido y avanzado de los controles de versiones, pudiendo elegir qué archivos actualizar en cualquier lado de nuestra conexión (servidor o cliente), omitir cambios para no pisar nuestro trabajo con el de otros, y viceversa, y una gran cantidad de operaciones de la misma índole. Un uso apropiado y sincronizado de los IDE y el control de versiones permiten trabajar de manera paralela sobre el mismo proyecto sin entorpecerse en el trabajo.

La sincronización de un IDE con el repositorio del proyecto permite además saber qué archivos han cambiado y por ende tener un control de versiones más allá del servidor, tenerlo directamente sobre el código que estamos actualmente desarrollando, viendo qué archivos han cambiado, borrado o incluido desde la última actualización al repositorio.

Figura 2.5. *Funcionamiento básico de un control de versiones*

Es muy habitual que en los centros de trabajo se utilice el control de versiones como una ayuda directa al desarrollador del proyecto en cuestión. Aunque no sea el propósito inicial de la herramienta, no es tampoco una idea descabellada, nos permite tener un control de versiones a modo de copia de seguridad selectiva. Cualquier error o fallo garrafal en una aplicación nos podría permitir volver a una posición anterior donde la aplicación era perfectamente estable.

En definitiva, la integración del control de versiones en un IDE nos ofrece una multitud de opciones tanto para el desarrollo diario como para realizar trabajos y proyectos de manera colaborativa, una opción única e indispensable a la que todo buen desarrollador debería prestar atención.

2.4 NUESTRA ELECCIÓN VISUAL STUDIO

Siguiendo los criterios que hemos aprendido en este capítulo, hemos elegido utilizar el Visual Studio Ultimate 2010 como entorno de desarrollo. Visual Studio es uno de los entornos de desarrollo más pulidos, profesionales y completos que podemos encontrar en el mercado, además, tiene a sus espaldas una excelente plataforma de trabajo: el *framework* .NET.

Hemos escogido el Visual Studio por ser un entorno de desarrollo muy completo con una gran cantidad de características integradas y la solidez del programa en sí mismo. Las herramientas de las que hablaremos durante el resto del libro serán específicas para Visual Studio, aunque se realizan las menciones que se consideren oportunas a herramientas de funcionalidad semejante en otros entornos de desarrollo.

El código que utilizaremos para los ejemplos será principalmente de C#, aunque será recurrente utilizar código en Visual Basic o Java para los ejemplos. También se incluirá código y uso de características propias y específicas de la plataforma .NET, obteniendo con ello una visión más global que si nos centrásemos de manera exclusiva en un único entorno de desarrollo y en un único lenguaje de programación.

2.4.1 INSTALACIÓN

Una vez que tenemos el producto en nuestras manos, procederemos a instalarlo, la instalación es igual de sencilla que la de cualquier otro programa.

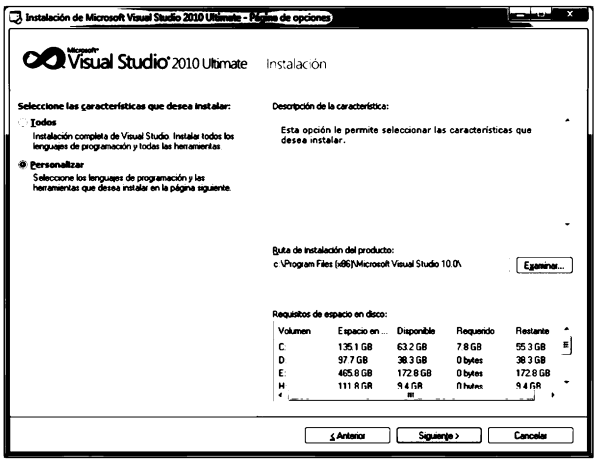


Figura 2.6. *Selección de ubicación y tipo de instalación VS2010 Ultimate*

Si queremos personalizar las características que instalaremos con Visual Studio, así como los lenguajes que vamos a utilizar, elegiremos la instalación personalizada. Y seleccionaremos las opciones deseadas.

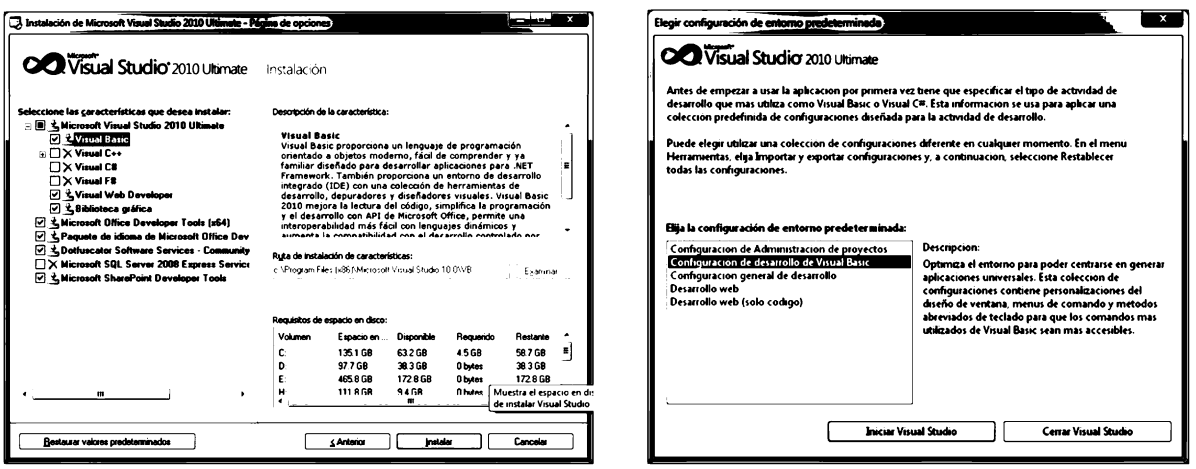


Figura 2.7. *Selección de características y configuración de entorno predeterminada*

Una vez terminado el proceso de instalación, pasamos a la primera ejecución del IDE. Nos aparecerá un cuadro de diálogo para escoger nuestra configuración de entorno. Visual Studio 2010 Ultimate tiene unas configuraciones predeterminadas dependiendo del uso que se le vaya a dar, o dependiendo del lenguaje de programación que se vaya a utilizar.

2.4.2 RECORRIDO POR LAS VENTANAS Y PALETAS PRINCIPALES

Como todo IDE que se precie, una vez que lo tengamos arrancado en nuestro equipo, visualizaremos una serie de ventanas acopladas y diversas barras de herramientas. Realizaremos un breve paseo general por las ventanas para familiarizarnos con el entorno.

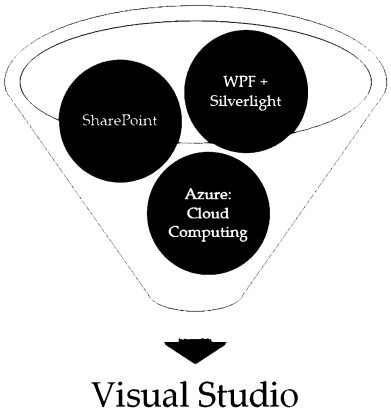


Figura 2.8. *Tecnologías y frameworks incluidas en Visual Studio 2010*

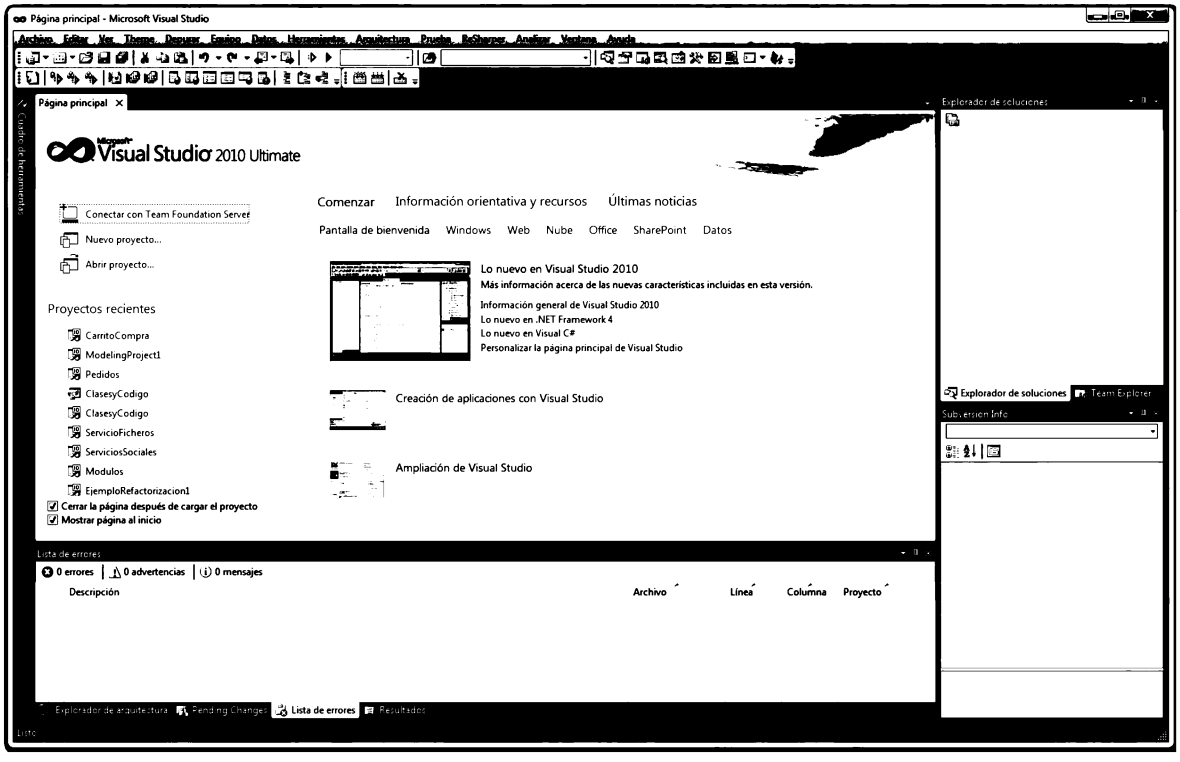


Figura 2.9. *Vista general del VS2010 Ultimate*

Página principal personalizada

Disponemos en primera plana de una página principal con información de actividad reciente que se puede personalizar para mostrar últimas noticias, tutoriales y guías, o simplemente información adicional sobre el IDE.

Figura 2.10. *Página principal del VS2010 Ultimate*

Explorador de soluciones

En la parte superior derecha del IDE tenemos el explorador de soluciones, en el que podemos ver la jerarquía de carpetas y proyectos de nuestras aplicaciones.

Nos permite además ver las referencias, conexiones de datos y dependencias de los diferentes proyectos, pudiendo establecer propiedades adicionales a las mismas.

Es dentro de esta ventana donde podremos acceder a las propiedades de los proyectos y soluciones, así como añadir nuevos elementos dentro de la jerarquía, ya sean clases, proyectos o ficheros varios.

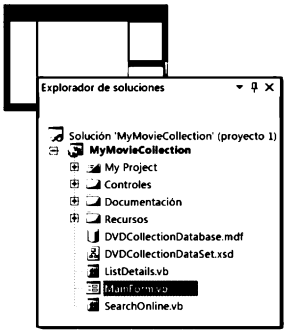


Figura 2.11. *Explorador de soluciones del VS2010 Ultimate*

Editor de diseño

Nuestras aplicaciones más básicas se basarán en el uso de la información, utilizando para ello distintos formularios, ya sean una aplicación de escritorio o una aplicación web.

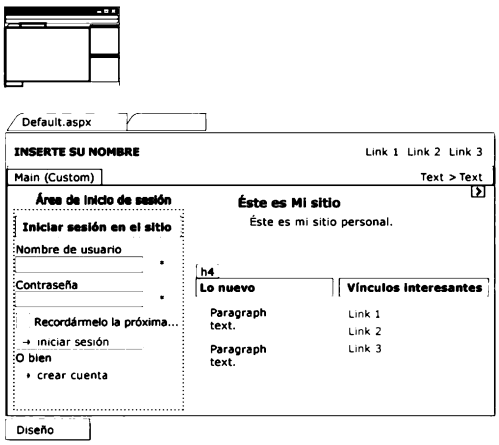


Figura 2.12. *Editor de diseño del VS2010 Ultimate*

Con el editor de diseño, podremos crear y posicionar los controles que nuestra interfaz necesita.

Editor de código

Como es lógico, las clases gráficas o formularios también necesitarán de una lógica que esté detrás de su comportamiento, para ello, podemos utilizar la vista de código del editor de formularios.

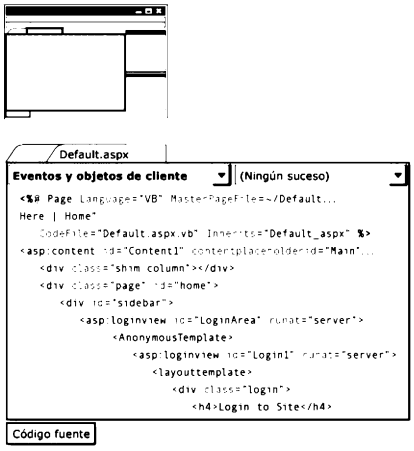


Figura 2.13. *Editor de código del VS2010 Ultimate*

Mediante los campos de selección situados en la parte superior del editor de código, podemos movernos por los diferentes controles y eventos para acceder a ellos de una manera más rápida y directa, facilitando y acortando el trabajo de codificación.

Editor de vista compartida

En ocasiones nos será de suma utilidad poder visualizar tanto el código como la interfaz de usuario relacionada, para ello, tenemos la vista compartida, que nos permite partir la pantalla de forma horizontal o vertical y modificar la posición de la línea divisoria para que se ajuste a nuestras necesidades.

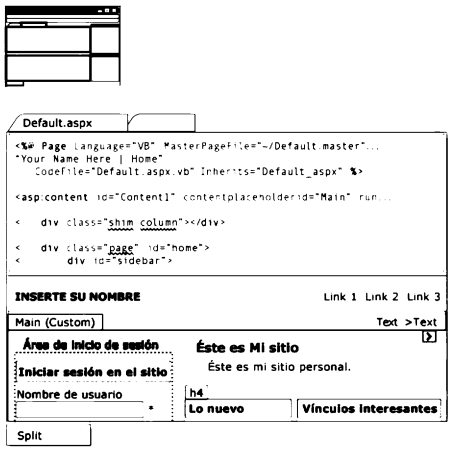


Figura 2.14. *Vista compartida en VS2010 Ultimate*

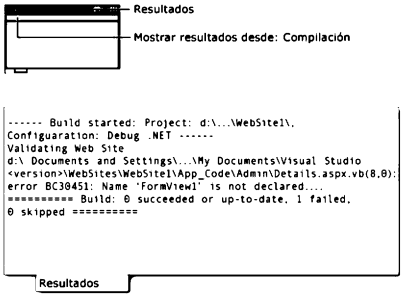


Figura 2.15. *Consola compartida del VS2010 Ultimate*

Consola de compilación

Además de la consola de pantalla que incluyen todos los entornos de desarrollo, también disponemos de una consola de compilación donde revisar los procesos y errores ocurridos durante dicho proceso.

Ventanas de depuración

Durante el proceso de depuración, podremos establecer puntos de observación e inspecciones para facilitar y mejorar la experiencia de depuración. Para ello, el Visual Studio nos ofrece una serie de pantallas específicas, visibles durante la depuración del programa con el objetivo de ofrecer toda la información necesaria de la manera más compacta mientras dura la ejecución de depuración.

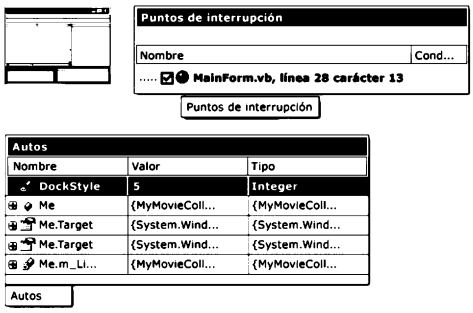


Figura 2.16. *Ventanas de depuración del VS2010 Ultimate*

2.4.3 PERSONALIZACIÓN Y CONFIGURACIÓN

Todo entorno de desarrollo debe permitir personalizar y añadir controles, y Visual Studio no es una excepción. La personalización del entorno de desarrollo es algo que no se debe infravalorar, recordemos que el objetivo principal de un entorno de desarrollo es facilitar al programador su tarea, por lo que poder redimensionar, reposicionar y modificar los elementos del entorno de desarrollo es una parte muy importante en la funcionalidad de un IDE.

Obviamente, la personalización no acaba en una configuración gráfica, sino que va más allá, estableciendo valores por defecto, y configuraciones específicas del comportamiento del IDE para ajustarlo a nuestras necesidades.

Ventanas

En Visual Studio Ultimate 2010, podemos añadir, quitar, acoplar y mover las ventanas de la forma que queramos de una manera muy sencilla. Como hemos visto antes, la configuración de ventanas inicial se compone de 4 partes: central, inferior, lateral izquierdo y lateral derecho. Un dato importante que hay que tener en cuenta es que tenemos dos tipos de ventanas: ventanas de documentos y ventanas de herramientas. Las ventanas de documentos siempre se encuentran en el bloque central del entorno y las de herramientas se encuentran en cualquier posición alrededor de ellas, por ello, vamos a diferenciar la personalización de posicionamiento y tamaño según el tipo de ventana.

Ventanas: Documentos

Las ventanas de documentos siempre se posicionan en el bloque central del entorno, aunque se pueden hacer flotantes arrastrándolas hacia cualquier punto de la pantalla. Se pueden colocar como una ventana independiente, como una parte de la organización de fichas (como si fuesen pestañas) o como parte de un grupo dividido de ventanas o grupo dividido de fichas. El modo más fácil y sencillo para lograrlo es arrastrando la ventana, una vez que la ventana

está siendo arrastrada, aparece un menú a modo de pequeñas imágenes para escoger la posición en la que queremos colocar la ventana.

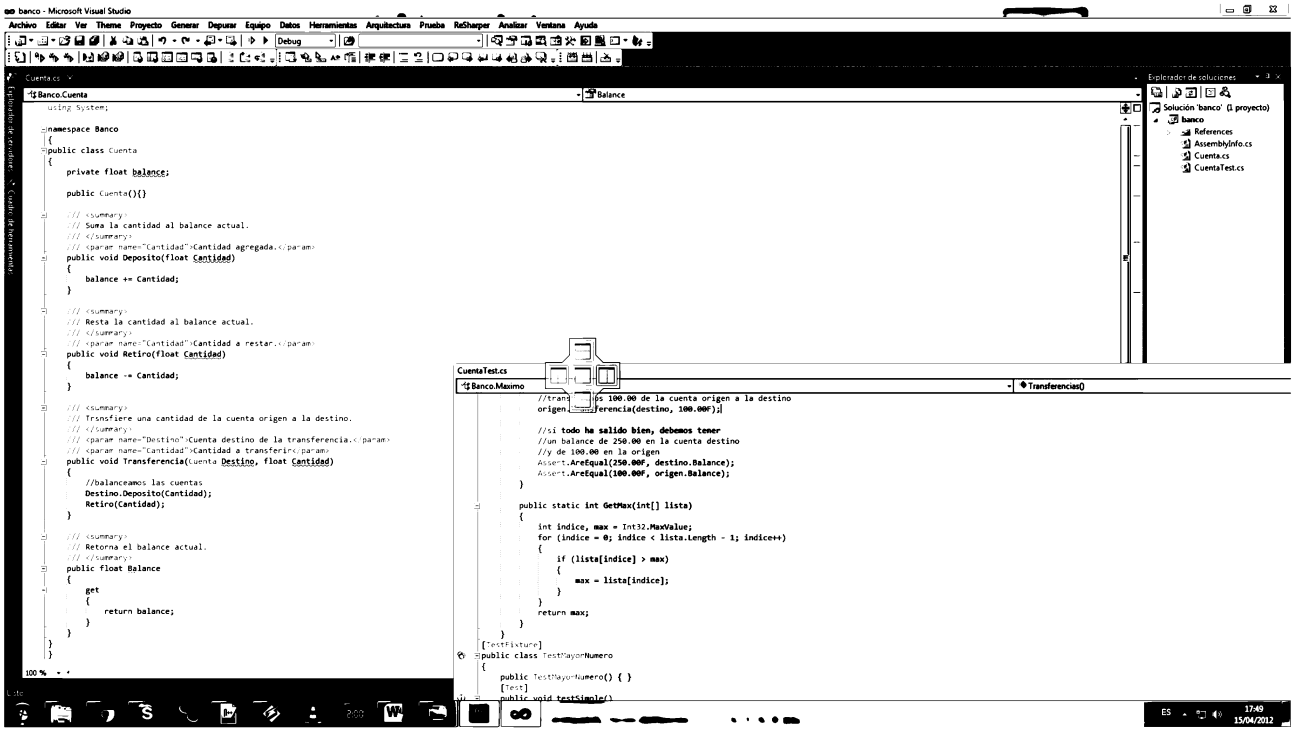


Figura 2.17. *Redistribución de ventanas en VS2010 Ultimate*

Ventanas: Herramientas

Las ventanas de herramientas tienen una mayor versatilidad de configuración, pueden colocarse en cualquier bloque o subbloque del entorno de desarrollo, incluyendo un bloque que viene desocupado por defecto, el superior, o en el bloque central.

Cualquier ventana de herramientas puede posicionarse en cualquier posición posible, incluyendo las pestañas o subbloques de los bloques de ventanas. Mediante el menú contextual del botón derecho delratón (o a través de la flecha en la parte superior derecha de la ventana) podemos elegir que esté acoplada, acoplada como ficha (como las fichas del bloque central) o que se oculte automáticamente, ocupando una porción ínfima y desplegándose al pasar el ratón por encima.

Por último, si queremos añadir una ventana al entorno de desarrollo, no tenemos más que ir al menú **Ver > Otras Ventanas** y escoger la que queramos.

Barras de herramientas

Como es habitual en muchas aplicaciones de escritorio, tenemos diversas barras de herramientas a nuestra disposición en los entornos de desarrollo. Se pueden añadir, quitar, mover y modificar.

El método más rápido para añadir barras de herramientas predefinidas es elegirla mediante el menú contextual que aparece al hacer clic con el botón derecho en la zona prefijada para las barras de herramientas, el bloque inmediatamente inferior al menú.

Para personalizar o crear nuevas barras de herramientas, utilizaremos la opción del menú **Herramientas > Personalizar**, y dentro de la ventana **Personalizar**, en **Comandos**. Dentro de esa ventana podremos modificar, reorganizar y añadir o eliminar elementos.

Opciones del entorno

Vamos a centrarnos más en la configuración del entorno de desarrollo que en la personalización de la interfaz del entorno. Para ello, nos vamos a ir a **Herramientas > Opciones** para modificar lo que necesitemos.

En el cuadro de diálogo que nos aparecerá, seleccionamos la opción **Mostrar todas las configuraciones**, que aparece en la parte inferior izquierda de la ventana.

Ahora, ya podemos ver todas las opciones básicas que nos ofrece el entorno de desarrollo, tendremos que tener cuidado con las modificaciones que hagamos, ya que muchas de ellas cambian sensiblemente la funcionalidad del entorno.

Dentro del apartado entorno, podemos ver una de las características de los entornos de desarrollo que resaltábamos al principio de este capítulo: el coloreado de código. Dentro de esta sección podremos elegir la fuente y especificar uno a uno cómo queremos que nos coloree los diferentes fragmentos de código para visualizarlos de la manera que nos parezca más clara o estemos más acostumbrados.

En el apartado de depuración tenemos una de las opciones más útiles: **Editar y continuar**, si lo activamos, podremos modificar el código en tiempo de ejecución mientras estamos depurando, lo cual nos permite resolver o hacer pequeños cambios para pruebas de una manera mucho más rápida que tener que parar la depuración, realizar la modificación y volver a depurar, sobre todo si nuestro proyecto es grande y tarda algunos minutos en compilar.

Dependiendo de la naturaleza y sistema objetivo de la aplicación, las configuraciones de rendimiento pueden ser claves para observar problemas de rendimiento al poder visualizar el tiempo en función de los ciclos de proceso y no de los milisegundos que ha tardado en realizarlo; los milisegundos dependen de la frecuencia y velocidad de proceso, los ciclos son más objetivos y, lo más importante, constantes, tal y como vimos en el tema 1, una operación siempre tarda los mismos ciclos (microinstrucciones) independientemente del procesador que lo ejecute.

Como podéis observar en un vista preliminar, hay una gran cantidad de opciones ofrecidas por el IDE para su configuración y personalización, siempre es útil invertir algo de tiempo en leer todas las opciones para comprobar si alguna opción nos resultaría útil modificarla, añadirla o incluso retirarla.

En caso de duda, se recomienda documentarse al respecto en la MSDN de Microsoft, la cual es muy completa y viene avalada por una gran cantidad de ejemplos e ilustraciones que facilitan la comprensión y ayudan a entender mejor las opciones y necesidades que podemos suplir con la configuración del IDE.

Aún no hemos añadido extensiones ni herramientas adicionales al entorno de desarrollo, por lo que todas ellas hacen referencia a las funcionalidades internas del IDE. Dependiendo de las extensiones instaladas, podremos encontrar opciones de configuración en esta misma ventana de diálogo, relacionadas con la extensión.

Opciones del proyecto

Las diferentes opciones y configuraciones que nos ofrece el entorno de desarrollo no se aplican solamente al entorno en sí mismo, sino a los proyectos que podemos crear.

Dependiendo del tipo de proyecto, también nos podremos encontrar con diferentes opciones dentro de las propiedades del proyecto, por ejemplo en un proyecto de aplicación web, tenemos las configuraciones del servidor de desarrollo o la ruta del directorio virtual del IIS, opciones inexistentes en una aplicación de escritorio.

Para acceder a las propiedades del proyecto, deberemos hacer clic con el botón derecho en el proyecto y seleccionar **Propiedades** en el menú contextual emergente que nos aparece. Una vez en dicha pantalla (por defecto se coloca como una ficha en nuestra ventana de documentos), tenemos un menú con todas las categorías de opciones disponibles en la parte izquierda.

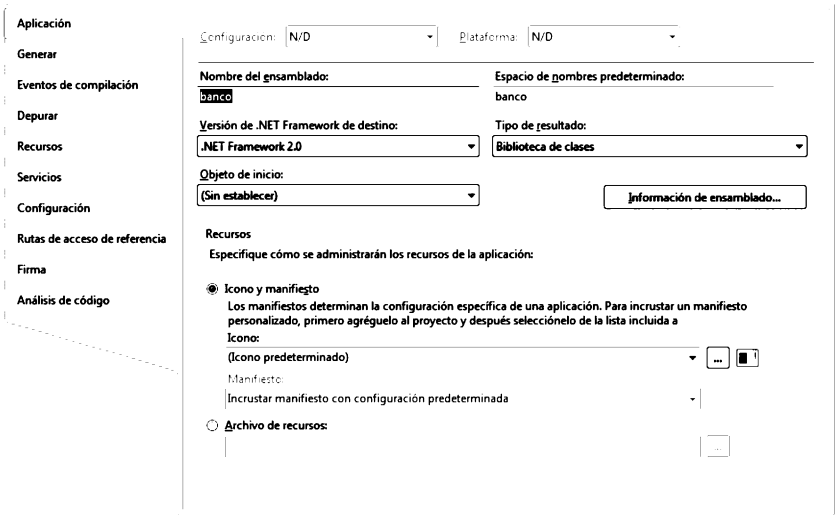


Figura 2.18. *Opciones de proyectos en VS2010 Ultimate*

Dentro de la sección Aplicación, nos encontramos con varias opciones interesantes, una de ellas de suma importancia, la que define cuál será el objeto de inicio, pudiendo usar para ello un formulario, o un método "Sub Main" como primer bloque de código que hay que ejecutar al arrancar la aplicación.

Habilitar marco de trabajo de la aplicación también resulta una opción importante que hay que tener en cuenta, nos permite, entre otras cosas, establecer que la aplicación sea de instancia única (no puedes tener dos procesos de la aplicación abiertos) o utilizar el método de identificación de Windows para registrar ese usuario en la aplicación. Si tu aplicación tendrá una autentificación de usuarios mediante bases de datos, Windows o directorio activo (LDAP) es una opción importante para elegir la configuración adecuada y evitar así problemas derivados de una mala propiedad de proyecto.

Como es obvio, las opciones del proyecto no solo dependen del tipo de proyecto, sino del lenguaje utilizado en el proyecto, si estuviésemos con una aplicación de escritorio Windows Forms con Visual Basic, veríamos unas opciones dentro de la sección Compilar que no veríamos con un proyecto de C#, como por ejemplo las opciones de compilación *optión explicit*, *option strict*, *option compare* u *option infer*.

En la sección Publicar podremos especificar la versión de publicación y la posibilidad de que se incremente automáticamente cada vez que publiquemos la aplicación. Dentro de esta sección, podemos también realizar la acción de publicar nuestro proyecto directamente.

El análisis de código establece las reglas para avisar de posibles errores durante el codificado y compilación del proyecto. Las reglas se pueden especificar dentro del propio entorno de desarrollo, para todos los proyectos, y no tener que configurarlo proyecto por proyecto; no obstante, en ocasiones podremos querer crear o modificar reglas para un proyecto concreto debido a sus características.

RESUMEN DEL CAPÍTULO

Una vez establecida una base en el mundo del desarrollo del software, ya hemos entrado en la materia que nos interesa: los entornos de desarrollo.

En este tema hemos aprendido las características que debe tener un IDE y en cuáles nos tenemos que fijar a la hora de elegir nuestro IDE. Hemos realizado un recorrido global por las diferentes funcionalidades de los IDE, consiguiendo con ello saber qué debemos esperar de un IDE, sean cuales sean sus propósitos.

El alumno debería ser capaz de evaluar y conocer sus necesidades, esto le permitirá escoger el IDE apropiado mediante un breve período de evaluación y documentación. El alumno, además, ha tenido una primera toma de contacto con el entorno de desarrollo Visual Studio Ultimate 2010, aprendiendo cómo instalarlo y configurarlo, así como a familiarizarse con los diferentes bloques de herramientas necesarias para su utilización.

Este capítulo introductorio a los IDE y Visual Studio carece de actividades, ya que el propio contenido del capítulo establece una larga actividad en el apartado 2.4. Por ello, se recomienda estudiar dicho apartado teniendo delante el entorno de desarrollo, esto permitirá una mayor comprensión de dicho bloque temático.

En este capítulo, hemos escogido el entorno de desarrollo sobre el cual se va a centrar este libro, por lo que las herramientas y uso que se explicarán de aquí en adelante serán específicos para Visual Studio 2010 Ultimate.

TEST DE CONOCIMIENTOS

1 ¿Cuáles son los componentes comunes básicos de los IDE?
- a) Editor de texto, Compilador, Intérprete, Depurador y Cliente.
- b) Editor de texto, Compilador, Máquina virtual y Cliente.
- c) Editor de texto, Compilador, Intérprete, Depurador y Ejecutor.
- d) Ninguna de las respuestas anteriores es correcta.

2 El autocompletado realiza las siguientes funciones:
- a) Visualiza los métodos y propiedades accesibles.
- b) Visualiza un listado de sugerencias al empezar a escribir un nombre de una variable, clase u objeto entre otros.
- c) Las respuestas *a* y *b* son correctas.
- d) Ninguna de las respuestas anteriores es correcta.

3 ¿Cuáles son los criterios globales a la hora de elegir un IDE?
- a) Sistema operativo, lenguajes de programación y disponibilidad.
- b) Sistema operativo, arquitectura del procesador, herramientas y disponibilidad.
- c) Sistema operativo, lenguajes de programación, herramientas y disponibilidad.
- d) Todas las respuestas anteriores son correctas.

4 El objetivo principal de un IDE consiste en:
- a) Edición de programas.
- b) Facilitar la tarea al programador.
- c) Integrar herramientas de desarrollo en una sola aplicación.
- d) Todas las respuestas anteriores son correctas.

5 ¿Qué no podemos hacer mediante herramientas de trabajo colaborativo?
- a) Mantener un control de versiones del proyecto.
- b) Tener una copia de respaldo de nuestro proyecto en un repositorio.
- c) Trabajar de manera conjunta con desarrolladores de diferentes ubicaciones.
- d) Ninguna de las respuestas anteriores es correcta.

6 En las propiedades de un proyecto, podemos elegir una opción de compilación llamada “Convertir aplicación de instancia única”. ¿Qué hace dicha opción?
- a) Automatiza un patrón de instancia única para todas las creaciones de instancias de nuestro proyecto.
- b) Evita que se pueda ejecutar otra instancia de la aplicación de manera simultánea.
- c) La aplicación solo se podrá ejecutar una única vez en cada equipo.
- d) Ninguna de las respuestas anteriores es correcta.

3 Depuración y realización de pruebas

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer el proceso de depuración y su utilidad en el desarrollo de software.
- ✓ Aprender las diferentes herramientas de depuración.
- ✓ Comprender el concepto de pruebas de software y sus diferentes tipos.
- ✓ Aprender a crear pruebas unitarias y a utilizar herramientas para ello.

3.1 HERRAMIENTAS DE DEPURACIÓN

La depuración es uno de los procesos más importantes en el desarrollo de software, nos permite identificar y corregir errores de programación mediante la ejecución controlada del software.

Esta tarea la realiza el depurador, uno de los componentes básicos de un entorno de desarrollo, es, sin duda, una de las herramientas más importantes de un IDE, y una ayuda extrema y constante a la hora de buscar problemas en el código. Una vez que te has acostumbrado a utilizar un depurador, encontrarnos con la ausencia de él en un IDE te hace sentir como un hombre ciego en un cuarto oscuro buscando un gato negro que no está allí.

Gracias al depurador, podemos ver el proceso de un programa paso a paso, observando los valores de nuestros métodos, variables y objetos, lo cual facilita sumamente la tarea, o podemos simplemente establecer puntos de control que interrumpen la ejecución del programa mostrándonos el código en donde pusimos el punto de interrupción con los valores actuales.

3.1.1 PUNTOS DE RUPTURA

Los puntos de ruptura o puntos de interrupción son puntos de control situados en líneas concretas de nuestro código fuente. Cuando el depurador pasa por uno de esos puntos, detiene la ejecución del programa.

Colocar puntos de ruptura en Visual Studio es muy sencillo, lo único que hay que hacer es un clic con el botón izquierdo en la parte izquierda de nuestro editor de texto en la línea concreta en donde queremos que el programa se detenga. Una vez hecho, se coloreará el fondo de la línea seleccionada de color rojo, y aparecerá una bola del mismo color en el lugar en que hicimos clic.

Se pueden colocar todos los puntos de interrupción que se desee, lo cual resulta muy útil cuando se está depurando un error que tiene una larga pila de llamadas.

También se pueden colocar puntos de interrupción de forma manual, escribiendo unas sencillas líneas de código.

EJEMPLO 3.1

PUNTO DE RUPTURA MANUAL

```
if (System.Diagnostics.Debugger.IsAttached)
{
    System.Diagnostics.Debugger.Break();
}
```

La utilidad de un punto de ruptura manual recae en despliegues de software en donde no se puede depurar, como por ejemplo que, una vez desplegada una aplicación web, ésta no funcione correctamente, sin embargo en nuestro servidor de desarrollo funciona correctamente, si no podemos emular las condiciones en las que se produce el error, habrá que buscarlo en el servidor en donde se desplegó la aplicación.

Los puntos de interrupción se pueden personalizar, añadiendo una condición o un filtro. Para añadir una condición a un punto de interrupción hay que hacer clic con el botón derecho del ratón en el punto de ruptura y seleccionar **Condición**. Aparecerá un cuadro de diálogo en donde establecer la condición en la cual queremos que el punto de interrupción salte, y, además, especificar si el condicional es de tipo "Es true", con lo cual se interrumpirá el programa cuando la condición sea verdadera, o escoger "Ha cambiado", con lo que el programa se detendrá cuando el resultado de la condición cambie, obteniendo un funcionamiento similar al que conseguimos con la instrucción while.

3.1.2 PUNTOS DE SEGUIMIENTO

Los puntos de seguimiento son puntos de interrupción que realizan una acción especificada, pero no interrumpen la ejecución del programa. Se utilizan para poder llevar un seguimiento de porciones concretas de código problemático, para observar lo que ha ocurrido cuando el programa ha pasado por dicho punto. Al igual que los puntos de ruptura, se pueden configurar con condiciones o filtros funcionando de igual modo, pero sin detener la ejecución del programa.

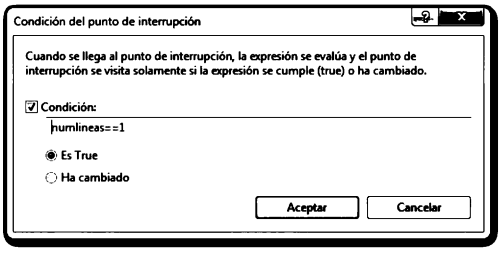


Figura 3.1. *Establecer un condicional para el punto de interrupción*

3.1.3 INSPECCIONES

Durante la depuración de código, es importante conocer el valor que tienen nuestras variables, las propiedades de nuestros objetos, los elementos y valores de nuestras colecciones o el resultado de un condicional.

Para ello, agregamos inspecciones, podemos agregar inspecciones existentes en nuestro código o personalizar los nuestros propios, por ejemplo, podríamos crear una inspección de un condicional que no existe en nuestro código o comprobar el valor de una propiedad de una instancia concreta de un objeto en un momento determinado.

Aunque esto podríamos hacerlo pasando simplemente el ratón por encima de aquello de lo que queremos conocer más información, también las podemos agregar a nuestro panel de inspecciones. Para ello, solo tenemos que hacer clic con el botón en la variable o propiedad que queremos inspeccionar, o seleccionar la porción de línea que queremos inspeccionar y hacer clic con el botón derecho en ella para agregarla.

También podríamos hacerlo de manera manual, lo que nos permitirá además crear inspecciones manuales que no tengan por qué existir en nuestro código. Lo más usual es utilizar esas inspecciones manuales para incluir condicionales.

Una vez agregadas las inspecciones, podemos ver los valores y cómo van cambiando mientras depuramos, lo que nos permitirá identificar dónde se encuentra un problema determinado mientras depuramos.

ACTIVIDADES 3.1

Cree un sencillo programa que lea un archivo de texto línea a línea y lo muestre por pantalla.

a. Establezca un punto de interrupción en la cabecera del bucle para leer las líneas. Observe que se detiene en cada iteración.

b. Agregue una inspección que permita observar el contenido de la línea que está leyendo en ese momento. Reflexione sobre los diferentes cambios que tendría que realizar a dicha inspección si utilizase otras instrucciónes de bucle.

c. Defina un punto de interrupción con un condicional si se trata de la segunda línea.

d. Depure el programa paso a paso (F11) y observe cómo es el flujo de ejecución en la línea de declaración del bucle y cómo va ejecutando las diferentes partes de dicha línea.

e. Escriba el código necesario para establecer un punto de ruptura manual, establezca un punto de interrupción en la línea que detiene el programa. Observe qué ocurre.

3.2 ANÁLISIS DE CÓDIGO

Una de las tareas más importantes que deberemos realizar mientras desarrollamos software es asegurarnos de que nuestro código funcionará correctamente. Para ello contamos con infinidad de herramientas y técnicas que nos facilitan dicha tarea, no obstante, el diseño apropiado y detallado del código es una tarea que nos evitará muchos problemas; si nos tomamos el tiempo necesario para analizar lo que debemos hacer para conseguir el producto deseado, evitaremos que nos ocurran problemas que no parecen evidentes a simple vista, como organizar nuestra estructura de clases de una forma que aseguremos la coherencia del contexto de una petición *http* en una aplicación web.

Aun teniendo todas esas consideraciones en cuenta, siempre podremos cometer errores, ya sea de funcionalidad y coherencia o errores en el propio código.

Para ayudarnos a identificar en tiempo real errores en el código producto de despistes o de malas prácticas de desarrollo de software, contamos con el analizador estático de código de Visual Studio y la definición de reglas para el análisis.

3.2.1 ANALIZADOR ESTÁTICO DE CÓDIGO

El analizador de código tiene tres modos de hacernos saber que algo va mal o podría ir mal: los errores, las advertencias y los mensajes. Inicialmente, Visual Studio nos muestra en la ventana de errores los fallos de compilación, es decir, nos avisa de que algo en nuestro código no podrá ser compilado debido a los errores que contiene. Si por ejemplo tuviésemos el siguiente código en nuestro programa:

EJEMPLO 3.2

ERROR DE CONVERSIÓN

```
int numeros = new int[2];
```

Veremos que nos avisa de la imposibilidad de convertir implícitamente el tipo “int[]” en “int”. Desde luego esto se trata de un despiste por nuestra parte. Este tipo de avisos pueden venir por la detección de un error de compilación o por una violación en alguna de las reglas de análisis de código que tenemos activadas en nuestro proyecto.

Por defecto, tenemos una serie de reglas o paquetes de reglas disponibles, pero podremos descargar más reglas para activarlas en nuestro código o incluso definir qué reglas queremos usar y cuáles no.

En el apartado de advertencias, nos solemos encontrar con posibles problemas o errores de código que provienen de la creación de un “Contrato de código” o de una regla que valida una convención a la hora de realizar una operación concreta. Existen reglas para asegurar la protección y estructura del desarrollo en tres capas y sus diferentes modelados como el MVC.

Podemos ver detalladamente el conjunto de reglas denominado “Todas las reglas de Microsoft”, lo que sin duda nos dará una idea más clara del uso y potencial de esta herramienta. Para ello, iremos a las propiedades del proyecto, a la pestaña **Análisis de código** y haremos clic en el botón **Abrir**.

Podemos observar que las reglas ahí descritas generan únicamente advertencias, por lo que podemos tener por seguro que los errores de los que nos avise Visual Studio no provienen de ninguna de estas reglas.

Si disponemos de conexión a Internet, podemos ver una descripción detallada de cada regla contenida en el conjunto de reglas directamente desde la MSDN de Microsoft, lo cual nos aportará el motivo y solución de un mensaje concreto o, en su defecto, nos permitirá discernir con un mayor acierto las reglas que queremos o deberemos usar en nuestro proyecto para llevar el desarrollo de software de manera apropiada.

Podremos también definir un conjunto de reglas personalizado para evitar molestos mensajes que no nos interesen y asegurarnos de que los mensajes indiquen un fallo que queremos solucionar. Para ello nos dirigiremos a **Archivo > Nuevo > Archivo** y elegiremos **Conjunto de reglas de análisis de código** como la plantilla que hay que utilizar.

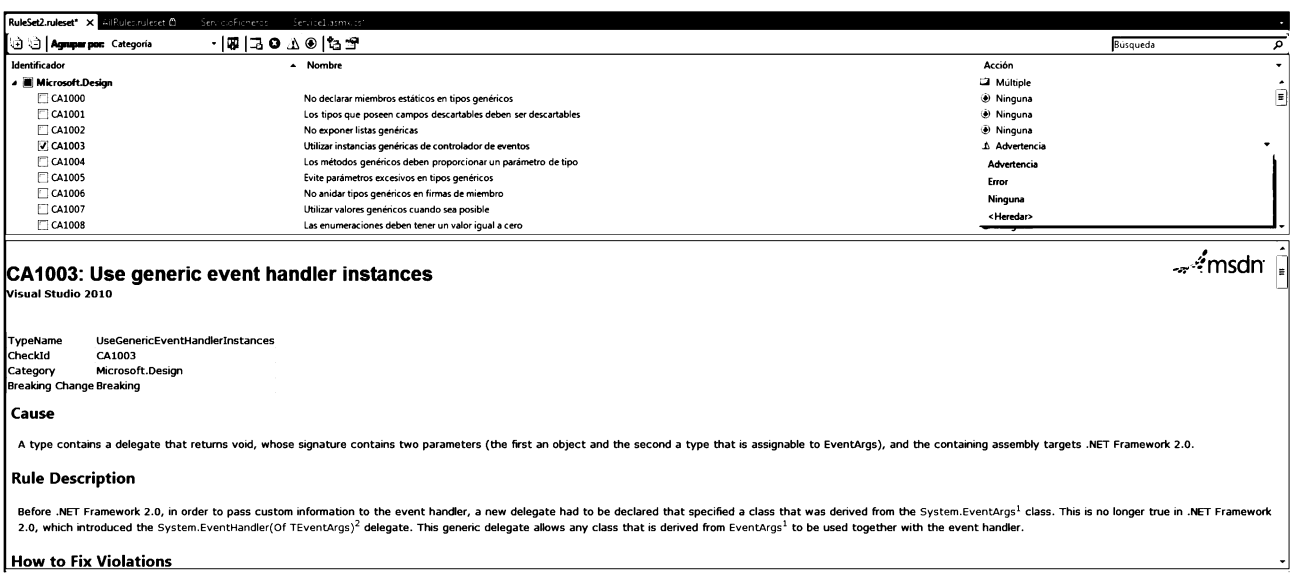


Figura 3.2. *Crear un conjunto de reglas personalizado*

También podemos cambiar por qué ventana nos avisa Visual Studio del incumplimiento de alguna norma si abrimos una regla, pero las reglas que tenemos instaladas por defecto están protegidas, por lo que tendremos que guardarlas en un archivo nuevo.

Una vez ahí, al igual que cuando abrimos el conjunto de reglas de Microsoft, podemos ver e informarnos sobre todas las reglas que tenemos disponibles y elegir las que más se ajusten a nuestras necesidades. Además de poder elegir las reglas concretas que queremos activar, podemos establecer qué tipo de mensaje queremos que nos muestre, lo cual nos permite elegir las reglas que más importantes nos parecen para que nos las muestre por la ventana de errores en vez de por la ventana de advertencias.

Contratos de código

Los contratos de código son un modo de generar advertencias y mensajes para casos que determinemos. De este modo podemos estar atentos a los diferentes problemas que podrían surgir, siempre y cuando hayamos pensado en ellos antes.

Para ello se utiliza System.Diagnostics.Contracts, que nos permitirá utilizar los métodos Ensure y Requires para nuestros propósitos. Supongamos que estamos realizando un programa en el que tenemos dos métodos, uno que calcula la dimensión de un *array* que debemos dimensionar, y otro que dimensiona dicho *array* mediante un valor pasado por parámetro.

EJEMPLO 3.3

USO DE ENSURE

```
public static int ObtenerDimension()
        {
            Contract.Ensures(Contract.Result<int>() > 0);

            //Código del método

            return dimension;
        }
```

EJEMPLO 3.4

USO DE REQUIRES

```
public static void DimensionarArray(int[] array,int dimension)
        {
           Contract.Requires<ArgumentException>(dimension>0);

            //Código del método
        }
```

Mediante este sencillo procedimiento, hemos creado un contrato de código que el analizador de código revisará y nos avisará de las necesidades de estos dos métodos mientras programamos, resaltando si hemos roto la regla de algún modo, como, por ejemplo, asignando a "dimensión" el valor 0).

3.3 CASOS DE PRUEBA

Los casos de prueba son una serie de condiciones que se establecen con el objetivo de determinar si la aplicación funciona correctamente según lo esperado. Para cada tarea, pueden surgir diversos casos de prueba, teniendo en cuenta todos los factores posibles para no dejar ningún cabo suelto sin probar y evitar así que ocurran errores no conocidos.

En principio los casos de prueba tienen un enfoque genérico y posteriormente se añaden más condiciones y variables que lo completen; por ejemplo, si tenemos una aplicación que genera un archivo de texto partiendo de los datos de un formulario, primeramente definiríamos el caso de prueba que compruebe que el archivo de texto se genere correctamente y posteriormente diversos casos de prueba anidados al caso de prueba inicial en los que se estipulen las diferentes entradas y tipos de datos del formulario, además de comprobar lo que ocurre si el archivo existe, si no existe, si existe y está vacío, si existe y tiene contenido, si existe y tiene solo saltos de línea…

Como vemos, hasta la operación más sencilla podría generar una cantidad de casos de prueba abrumadora, normalmente esto no es así, pues la validación de formularios, así como otros métodos de entrada y salida, son validados generalmente antes de realizar cualquier operación directamente desde el código sin que el analista haya tenido que considerarlo en sus casos de prueba.

Los casos de prueba suelen tener un formato concreto para llevar su seguimiento e informe, especificando qué es lo que se prueba, cuáles son las condiciones o premisas, cuál es el resultado esperado y por supuesto cuál es el resultado obtenido.

Tabla 3.1 Formato modelo de los casos de prueba

ID	¿Qué?	Descripción	Requisitos	Res. esperado	Res. obtenido
001	APP	Carga fichero	Que exista	OK/KO	OK/KO

Existen diversos tipos de casos de prueba (muchas veces, confundidos por su similitud en algunos aspectos por los casos de uso) que se definen por la naturaleza de las pruebas y no por el tipo de operación que conlleva la prueba, por ejemplo, aunque las pruebas de caja blanca vayan sumamente ligadas al procedimiento en sí como una correcta evaluación de un condicional y las pruebas de caja negra se realicen desde el punto de vista del usuario final, podemos estar comprobando la misma operación y los mismos datos aunque sea desde un punto de vista diferente.

3.3.1 CAJA BLANCA

Las pruebas de caja blanca se centran en el funcionamiento interno del programa, observando y comprobando **cómo** se realiza una operación.

Son siempre las primeras pruebas que hay que realizar, pues revisan la estructura y funcionalidad interna del programa. Se pretende con ello encontrar defectos básicos de software no relacionados con la interfaz de usuario.

Existen diversos tipos de pruebas dentro de las pruebas de caja blanca, pudiendo medir y comprobar métodos o algoritmos concretos, además de los caminos ciclomáticos propuestos por Tom McCabe, que veremos a continuación.

Prueba del camino básico

El método del camino básico se diseñó con el objetivo de obtener una medida de la complejidad lógica para usarla como guía de un conjunto de caminos de ejecución.

Este método se basa en el principio que establece que cualquier diseño procedimental se puede representar mediante un grafo de flujo. La complejidad ciclomática de dicho grafo establece el número de caminos independientes, cada uno de esos caminos se corresponde con un nuevo conjunto de sentencias o una nueva condición.

El funcionamiento de este método es muy sencillo, consiste en definir los diferentes bloques para cada camino posible y obtener los diferentes caminos para recorrerlo por todas las opciones posibles evitando saltarnos ninguna.

Pongamos un ejemplo sencillo: tenemos un programa que comprueba si dos números son pares y, de ser así, comprueba si son múltiplos recíprocos, si los dos son impares, comprueba si son múltiplos de 3, y si no es ninguna de las anteriores, comprueba también si son múltiplos recíprocos. En este caso tendríamos 5 bloques.

- Inicio.
- Son los dos pares.
- Son los dos impares.
- Uno es par y otro es impar.
- Son múltiplos recíprocos.
- Son múltiplos de tres.

Tendríamos por lo tanto los siguientes caminos:

- 1,2,5 | 1,3,6 | 1,4,5

En este caso concreto tan simple, los caminos posibles nos indican 3 cosas: tienen la misma complejidad, uno de los caminos es redundante y tenemos que comprobar tres caminos para cubrir todas las posibilidades. Si ajustásemos los condicionales, para fusionar el bloque 2 con el bloque 4, tendríamos dos caminos de la misma complejidad.

Prueba de condiciones

Al igual que en el caso anterior, las pruebas de condiciones evalúan los caminos posibles, en este caso de forma que solo provengan de condicionales. Con un objetivo añadido, ya que se estipula que si un mismo conjunto de casos de prueba evalúan correctamente una condición o serie de condiciones, en este caso se utiliza un método muy similar a una tabla de verdad, en donde se especifica también la operación relacional.

Es importante resaltar que para que las pruebas sean efectivas, no deben ser redundantes, por lo que a la hora de construir nuestra tabla de la verdad, debemos tener presentes las condiciones cortocircuitadas. De este modo, para evaluar una expresión del estilo (E1 | | E2), solo deberíamos evaluar el valor de E2 cuando E2 sea falso, ya que cuando la expresión E1 es verdadera la comprobación termina por no ser necesario evaluar el valor de E2.

Como nota adicional, en la programación orientada a objetos, debemos tener presente esa capacidad de cortocircuitar una condición a la hora de construirla. Por ejemplo, sabemos que no podemos invocar métodos de objetos instanciados con valor nulo, ya que generará una excepción, por lo que si primero evaluamos si dicha instancia es nula con la cortocircuitación adecuada, nunca tendremos el problema de la referencia nula.

Pruebas de bucles

Las pruebas de bucles no evalúan como en el resto de pruebas de caja blanca las condiciones de dicho bucle (al menos no directamente), sino las posibilidades que nos ofrece. Todos los bucles tienen una condición que nos establece la cantidad de iteraciones que nos va a realizar dicho bucle. Por ello, para evaluar correctamente todas las opciones, tenemos que valorar los siguientes comportamientos para un bucle con "n" iteraciones.

- El fujo del programa no entra ninguna vez al bucle.
- Pasa una única vez por el bucle.
- Pasa dos veces por el bucle.
- Pasa m veces por el bucle, donde $m<n$.
- Hace n-1 y n+2 iteraciones en el bucle.

Para los bucles anidados, tendremos que realizar dichas evaluaciones en el bucle más interno e ir subiendo nivel a nivel. Primero de manera independiente y luego de forma coherente con el resto de bucles de los que depende por la anidación.

Como comentario adicional, si se desea realizar correctamente las pruebas de caja blanca para bucles no estructurados, se deben rediseñar para que se ajusten a las condiciones de la programación estructurada.

3.3.2 CAJA NEGRA

Las pruebas de caja negra se enfocan en los métodos de entrada y salida de la aplicación, no en cuestión de formato, sino de validar y controlar los datos de entrada para evitar errores y, por supuesto, al igual que en todas las pruebas, obtener los resultados esperados.

Estas pruebas deben hacerse mediante el uso de la interfaz de la aplicación, ya que es en el controlador de la interfaz en donde deberíamos validar los datos de entrada.

Partición equivalente

La partición equivalente es un método de prueba consistente en dividir y separar los campos de entrada según el tipo de dato y las restricciones que conllevan.

Para evitar que tengamos que hacer una cantidad de pruebas excesiva para cada entrada, se definen unas pruebas comunes dependiendo del tipo de dato del campo en cuestión. De este modo, agruparemos los campos en diferentes baterías de pruebas, asegurándonos de realizar las pruebas de la manera más completa y eficaz.

A la hora de especificar los diversos criterios para las pruebas, podemos centrar nuestra división de campos en 4 grupos.

- Si el campo debe de encontrarse en un rango, se especifica una clase de equivalencia válida y dos inválidas (los límites inferiores y superiores).
- Si el campo requiere de una entrada específica, se define una clase de equivalencia válida y dos inválidas.
- Si el campo especifica a un elemento de un conjunto, se define una clase de equivalencia válida y otra inválida.
- Si el campo especifica una condición de entrada lógica, se define una clase de equivalencia válida y otra inválida.

Para ver un ejemplo que lo deje más claro, si por ejemplo tenemos un campo "Código Postal" que no es obligatorio rellenar, pero que solo admite entradas de 5 dígitos numéricos. Deberíamos valorar si está o no está presente, permitiendo las dos opciones y, en caso de estar presente, deberíamos confirmar que se trata de un número de 5 cifras, bien estableciendo un máximo y un mínimo (10000-99999) o una condición que compruebe que la longitud sea 5 y que solo sean números.

Por supuesto, en caso de que la validación no se haya codificado, habría que comprobar cómo se comporta el sistema si el dato introducido no se corresponde con las condiciones que debería tener.

Análisis de valores límite

El AVL es una técnica complementaria a la partición equivalente, básicamente, nos indica que si especificamos un rango delimitado de valores o un número de valores específicos, también se deberá probar por el valor inmediatamente superior e inmediatamente inferior de dichas cotas.

Es decir, si nuestras cotas de valores son "a" y "b", deberemos probar los valores "a-1", "a+1", "b-1" y "b+1".

También tendremos que realizar la misma operación en las colecciones que estén delimitadas en tamaños, como por ejemplo un *array* de dimensión 5.

Para todas estas cuestiones y criterios, se deberán aplicar estos valores en la salida de la aplicación para evitar errores en los límites de los elementos en donde escribimos la salida de la aplicación.

3.3.3 RENDIMIENTO

Las pruebas de rendimiento miden el tiempo que le ha tomado a la aplicación realizar una acción específica. Si bien esto depende también de manera importante de la máquina en donde se esté ejecutando la aplicación, sigue teniendo validez, ya que nos permite probar y controlar tanto el tiempo para un equipo concreto como realizar operaciones de diferentes formas y ver cuál conlleva un mejor rendimiento.

EJEMPLO 3.5

USO DE STOPWATCH

```
static void Main(string[] args) {
            IList<int> lista = new List<int>();
            IList<int> lista2 = new List<int>();
            for (int i = 0; i < 1000; i++)
                lista.Add(i);
            Stopwatch crono = new Stopwatch();
            crono.Start();
            foreach (int num in lista)
                if (num % 2 == 0)
                    lista2.Add(num);
            crono.Stop();
            TimeSpan duracion1 = crono.Elapsed;

            System.Console.WriteLine("Tiempo : {0}",duracion1);
        }
```

Para realizar estas mediciones, usaremos un sencillo cronómetro llamado StopWatch, disponible en .NET. Gracias a este cronómetro, podremos ver lo que ha tardado nuestra aplicación en ejecutar nuestros comandos mediante milisegundos, aunque también podríamos usar los *ticks* o ciclos de proceso.

Como vemos, su uso es muy sencillo, solo hay que instanciar el cronómetro y arrancarlo y pararlo para que nos ofrezca una medición del tiempo transcurrido.

En las aplicaciones web, este tipo de pruebas también nos sirven de método de control para saber si es la aplicación la que va lenta, o en su lugar es el servidor o la conexión los que tardan en enviar la petición y recibir la respuesta.

Si se ha detectado una operación que conlleva mucho tiempo, habrá que particionar la operación en diversos cronómetros para identificar dónde se encuentra el problema e intentar solucionarlo.

EJEMPLO 3.6

COMPROBAR RENDIMIENTOS

```
static void Main(string[] args) {
            IList<int> lista = new List<int>();
            IList<int> lista2 = new List<int>();
            for (int i = 0; i < 1000; i++)
                lista.Add(i);
            Stopwatch crono = new Stopwatch();
            crono.Start();
            foreach (int num in lista)
                if (num % 2 == 0)
                    lista2.Add(num);
            crono.Stop();
            TimeSpan duracion1 = crono.Elapsed;

        crono.Reset();
            crono.Start();
            lista2 = lista.Where(n => n % 2 == 0).ToList();
            crono.Stop();
            TimeSpan duracion2 = crono.Elapsed;


            System.Console.WriteLine("Tiempo 1 : {0}\n",duracion1);
            System.Console.WriteLine("Tiempo 2 : {0}",duracion2);
        }
```

Como hemos comentado antes, también nos puede servir para optimizar las operaciones en cuestión de rendimiento utilizando técnicas o recursos diferentes para ejecutar una acción. En el ejemplo anterior, se utilizó un sencillo algoritmo que recorría una colección y guardaba en otra lista los valores pares de dicha colección, para ello, hemos utilizado un método muy sencillo y directo, pero ¿qué ocurriría si, en lugar de ello, hacemos uso de LINQ y el siempre útil IEnumerable?

Observamos que la operación realizada con LINQ tarda mucho más que si utilizamos un filtrado "manual" sencillo. Recordad que no siempre lo más sencillo es lo más eficaz. En ocasiones te parecerá que es más sencillo realizar un "select" completo en una consulta SQL y luego filtrar los resultados mediante código, pero con este sistema podemos comprobar que la búsqueda que realiza el gestor de la base de datos es tremendamente más eficaz que cualquier filtrado u ordenación mediante código que quisiéramos hacer.

En ocasiones, la heurística de la operación nos impedirá obtener caminos más cortos para mejorar el rendimiento, pero no se recomienda caer en esa mala praxis con asiduidad, ya que a la larga produce errores graves de rendimiento.

ACTIVIDADES 3.2

- Hemos visto en el ejemplo 3.4 que LINQ tardaba más en realizar el filtrado, pero, ¿es esto siempre así? Comprobémoslo. Modifique en el *for* que carga los números en la lista para que añada 100.000 elementos, ¿qué ocurre con el rendimiento? ¿Y si fuesen 40.000.000?
- Utilizando el ejemplo anterior, testee el rendimiento de la carga de la lista mediante el cronómetro. Cambie la lista por un *array*, ¿el rendimiento se ve afectado?
- Realice una consulta a la base de datos para obtener todas las tuplas y filtre los resultados para un campo concreto con la condición que quiera y testee su rendimiento. Modifique la condición, ¿cómo se ve afectado el rendimiento?
- Realice la misma operación de la actividad anterior, pero añadiendo la condición directamente en la consulta SQL. Compare los resultados.

3.3.4 COHERENCIA

Las pruebas de coherencia son subjetivas, es decir, no están ligadas a la propagación ni a los datos en sí mismos. Las pruebas de coherencia se enfocan, entre otras cosas, en el estudio del flujo de trabajo (*workflow*) de la aplicación de un modo coherente.

Con ellas comprobamos si la funcionalidad de la aplicación es correcta y no si la aplicación funciona correctamente. El mejor modo de entender las pruebas de coherencia es con un ejemplo, así que vamos a ello.

Supongamos que estamos desarrollando una aplicación que gestionará los empleados y pedidos de un restaurante de comida a domicilio. Los empleados tienen un "tipo" que especifica su función en la empresa, como "cocina" o "repartidor". Por ello, aunque nuestra aplicación establezca perfectamente un empleado asignado para repartir el pedido, no tiene sentido si el empleado asignado es un cocinero, por tanto la aplicación no es coherente, puesto que funciona correctamente, sin errores, pero permite operaciones que no tienen sentido en el ámbito sobre el que trabajamos. Este ejemplo en cuestión bien podría ser una prueba de caja negra, ya que deberíamos filtrar correctamente el tipo de empleado para elegir en la aplicación, pero es un buen modo de escenificar lo que implican las pruebas de coherencia.

De todos modos, pongamos un ejemplo más específico y exclusivo de este tipo de pruebas. Imaginemos que los pedidos contienen un campo "Observaciones" para que el cliente exponga información adicional sobre el pedido, ¿tendría sentido que se pueda rellenar ese campo una vez se ha servido y finalizado el pedido? Dependiendo del uso de la aplicación, podría tenerlo o no, si solo pueden ser observaciones sobre el pedido en sí mismo, no tendría sentido que se pueda rellenar una vez servido el producto, pero, si esas observaciones también pueden contener valoraciones sobre el servicio, entonces sí que tendría sentido, por tanto hay que tener todas estas cuestiones en cuenta a la hora de diseñar, e intentar ser lo más específicos y explícitos posible.

Cuando surgen dudas por la ambigüedad de uso o del *workflow* de una aplicación, suele ser un problema de diseño, los diagramas de secuencia deberían dejar bien claro el flujo del programa, pero no siempre es así, en ocasiones las peticiones son tan ambiguas o, por el contrario, tan específicas que complican y ofuscan el trabajo.

Por ejemplo, utilizando el ejemplo inicial, si en un principio la aplicación no se diseñó para que existiesen tipos de empleado, no podríamos controlar si los empleados escogidos para repartir el pedido son realmente repartidores, cocineros o bedeles, por tanto, habrá que añadir ese caso si queremos evitar problemas de coherencia o de ambigüedad.

3.4 PRUEBAS UNITARIAS

En el epígrafe anterior hemos hablado de las pruebas de caja blanca, las cuales son las primeras pruebas que se deben realizar, las pruebas unitarias son pruebas de caja blanca; si bien antes hemos hablado de las diferentes técnicas y metodologías para diseñar las pruebas de caja blanca, ahora vamos a hablar de cómo realizarlas una vez diseñadas.

Sin duda, las pruebas unitarias son una labor fundamental en el trabajo de todo desarrollador, no obstante, suelen ser creadas y ejecutadas por personal especializado en *software testing*.

Las pruebas unitarias son pruebas individuales para un método o clase, realizadas de manera sistemática a modo de batería de pruebas donde conocemos los datos de entrada y sabemos cuál sería el resultado esperado.

Hoy en día contamos con una serie de herramientas que nos facilitan esta tarea: desde la programación de pruebas automáticas que se crean cada vez que se implementa una nueva funcionalidad o método, o las más usuales, que nos permiten ejecutar las baterías de pruebas de una manera ordenada visualizando todos los resultados de una manera directa y eficaz.

Cabe resaltar que este tipo de pruebas son tremendamente útiles, prácticamente imprescindibles en el desarrollo colaborativo.

3.4.1 METODOLOGÍA

Las pruebas deberían implementarse de manera sistemática con cada nueva funcionalidad que se añada, teniendo de este modo las pruebas actualizadas en cada momento. Es importante que las pruebas se realicen y se ejecuten de manera incremental, de este modo, aunque hayamos probado una funcionalidad anterior en varias ocasiones con los resultados esperados, comprobaríamos en todas las pruebas que dicho funcionamiento no se ha visto alterado.

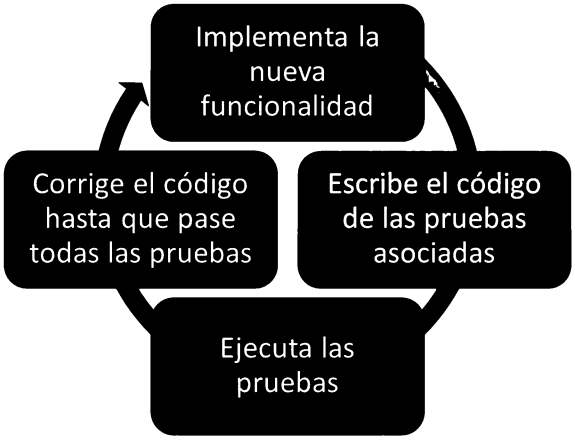


Figura 3.3. *Ciclo de pruebas unitarias sistemáticas*

Es recomendable que, antes de implementar una nueva funcionalidad, primero pensemos cómo deberíamos probarla para revisar que se ejecuta correctamente. De este modo, nos permite llevar un desarrollo en donde tenemos las ideas muy claras y podemos crear las pruebas inmediatamente después de codificar la funcionalidad sin que nuestro rendimiento se vea afectado.

3.4.2 NUNIT

Para realizar estas pruebas en nuestras aplicaciones, vamos a utilizar NUnit. NUnit es una aplicación disponible como extensión que nos facilitará la integración con el entorno de desarrollo.

Para poder trabajar con NUnit, primero deberemos descargarlo desde *http://www.nunit.org* e instalarlo. El instalador nos proveerá de las dlls necesarias en nuestro sistema. Para poder utilizarlo en nuestro proyecto, deberemos agregar una referencia a "nunit.framework" e incluir el espacio de nombres en nuestras clases test.

Figura 3.4. *Referencia y espacio de nombres de NUnit*

Una vez instalado y referenciado, vamos a instalar la extensión de VS para integrar los resultados en el sistema. Buscamos "nunit" desde el administrador de extensiones y la marcamos para instalar.

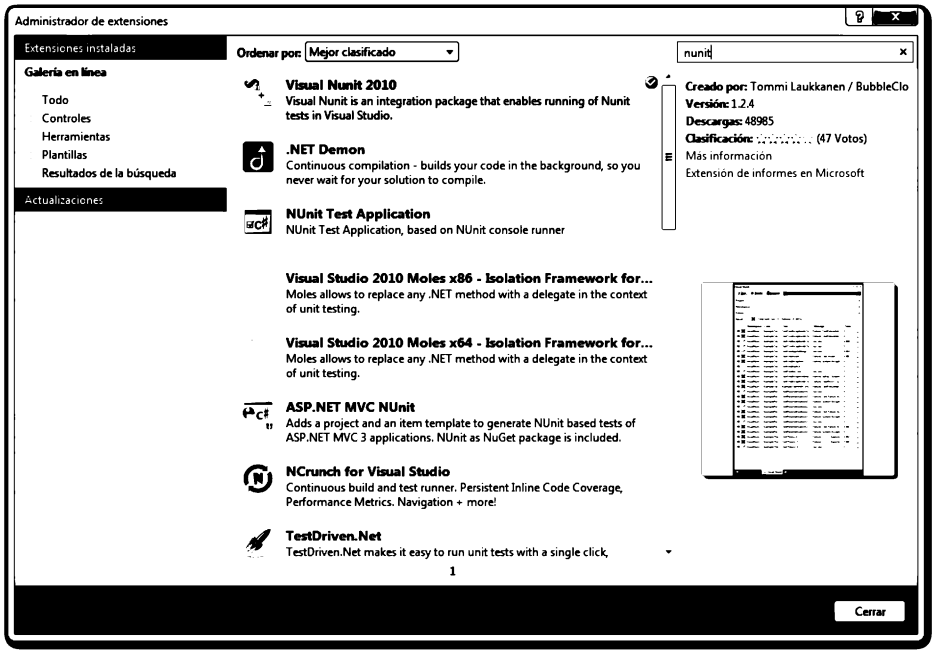


Figura 3.5. *Añadir la extensión Visual NUnit 2010*

Con esto ya tendríamos nuestro IDE preparado para trabajar sin problemas de manera rápida y sencilla con NUnit.

Para comenzar con el uso de NUnit vamos a ver el funcionamiento mediante el "Hola mundo" de las pruebas unitarias de NUnit. Crearemos un nuevo proyecto, una biblioteca de clases por ejemplo, no necesitamos ejecutar el proyecto para realizar las pruebas unitarias. Dentro de dicho proyecto, crearemos dos clases, en una implementaremos diversos métodos y en la otra realizaremos las pruebas unitarias, será nuestra clase *test*.

EJEMPLO 3.7

CLASE CUENTA

```
public class Cuenta
{
   private float balance;

   public Cuenta(){}

   public void Deposito(float cantidad) {
          balance += cantidad;
   }

   public void Retiro(float cantidad) {
          balance -= cantidad;
   }

   public void Transferencia(Cuenta destino, float cantidad) {
          destino.Deposito(cantidad);
          Retiro(cantidad);
   }

   public float Balance{
          get{return balance;}
   }
}
```

La clase cuenta es muy sencilla, es una implementación de un funcionamiento básico de una cuenta virtual en donde tenemos cuatro operaciones: ingresar, retirar, transferir y ver el saldo actual de la cuenta.

Para realizar las pruebas, instanciaremos la clase cuenta y realizaremos varias operaciones con ella para luego comprobar si el balance es el esperado. Instanciaremos otro objeto de la clase cuenta para utilizarla como cuenta de destino en la transferencia, con todo ello deberíamos ser capaces de establecer una serie de operaciones que nos permitan comprobar el funcionamiento de las operaciones sabiendo de antemano los valores de *balance* esperados.

EJEMPLO 3.8

CLASE CUENTATEST

```
[TestFixture]
public class CuentaTest
{
   public CuentaTest(){}

   [Test]
   public void Transferencias()
   {
Cuenta origen = new Cuenta();
         origen.Deposito(200.00F);

         Cuenta destino = new Cuenta();
         destino.Deposito(150.00F);

         origen.Transferencia(destino, 100.00F);

         Assert.AreEqual(250.00F, destino.Balance);
         Assert.AreEqual(100.00F, origen.Balance);
   }
}
```

Podemos observar en el código de la clase test que hemos especificado unos atributos en la clase y en el método de testeo, mediante dichos atributos es como avisamos a NUnit cuáles son nuestras clases test (mediante el atributo *TestFixture*) y cuáles son nuestros métodos test (mediante el atributo *Test*).

A la hora de obtener los resultados con NUnit, veremos que las clases marcadas con *TestFixture* serían nuestros casos de prueba, y los métodos marcados con el atributo *Test* serían nuestras diferentes pruebas. Por cada prueba, podemos además realizar todas las comprobaciones que queramos mediante la invocación de los métodos estáticos de la clase *Assert* de NUnit, por ejemplo podríamos añadir más operaciones, más instancias de cuenta y establecer muchos más *Asserts*.

Vamos a ver ahora los resultados de nuestras pruebas, como hemos instalado la extensión Visual NUnit, la operación es realmente sencilla, podemos utilizar el atajo de teclado Ctrl + F7 o irnos a **Ver > Otras Ventanas > Visual NUnit**, para que aparezca la ventana de NUnit.

Una vez allí, podemos ver que nos detecta automáticamente la clase de prueba que tenemos en nuestro proyecto (¡ojo! No os olvidéis de compilar el proyecto), y si no, solo tendríamos que elegirla mediante los campos de selección. Para ejecutar las pruebas, solo tenemos que hacer clic en el botón **Run**, la operación será inmediata, ya que nuestro proyecto es muy sencillo y nuestras pruebas muy simples.

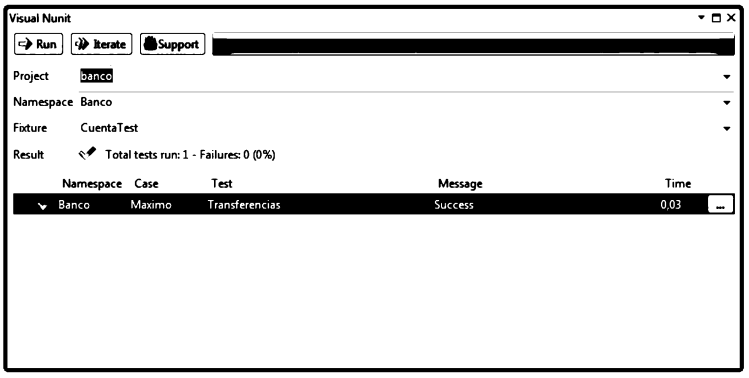


Figura 3.6. *Ventana de resultados de Visual NUnit*

Nuestras pruebas han sido un éxito, tal y como esperábamos, nuestro programa se ha comportado correctamente. Mediante el botón que se encuentra en el listado de casos de prueba con un texto de puntos suspensivos, podemos ver los detalles de las pruebas, en este caso, se encontrará vacío, ya que no ha habido errores ni tenemos más información añadida que mostrar.

Como vemos, el uso y funcionamiento de NUnit es muy sencillo y fácil de implementar, además de útil y rápido. Es una excelente herramienta que nos puede acompañar en todas las pruebas unitarias que necesitemos.

Es habitual encontrarse con compañeros de trabajo, de estudio, o simplemente con otros programadores, que desestiman la utilidad de las pruebas unitarias y solo se dedican a realizar las pruebas de caja negra, no es una actitud recomendable. Hay errores que no se pueden percibir desde la interfaz, nunca lo olvide.

ACTIVIDADES 3.3

- Añada o modifique un *Assert* incorrecto (con un resultado esperado incorrecto). Observe los resultados de las pruebas.
- Añada más operaciones y más *Asserts* asociados a esas operaciones y ejecute las pruebas. Reflexione sobre los resultados del test.
- Cree un nuevo método *Test* con otras operaciones para observar el muestreo de resultados de las pruebas.
- Cree una nueva operación que cancele la cuenta (la establece a cero) y defina pruebas que realicen operaciones con una cuenta que ha sido cancelada previamente estableciendo los *Asserts* que considere necesarios.

RESUMEN DEL CAPÍTULO

En este tema hemos aprendido a manejar, controlar y verificar la funcionalidad de nuestros programas a la hora de solucionar o comprobar la funcionalidad.

Nos hemos valido de diferentes herramientas y recursos disponibles para nuestro entorno de desarrollo, aprendiendo sus diferentes utilidades y casos de uso. Hemos aprendido también la piedra angular de todo entorno de desarrollo y de todo desarrollo de software: la depuración.

Según vamos avanzando nos damos cuenta de todas las funcionalidades y posibilidades que nos ofrece un entorno de desarrollo frente a un simple editor de texto y un compilador, pero también es fundamental darse cuenta de que nuestro raciocinio y capacidad de análisis es importante, pues constantemente tenemos que diseñar lo que queremos que el IDE haga por nosotros, y no es simplemente una herramienta mágica.

El uso de NUnit que hemos visto en este capítulo es tan importante como simple, las posibilidades que nos ofrece NUnit son muchas y muy variadas. Profundizar e investigar sobre el uso y posibilidades de la herramienta es una excelente recomendación para todos los estudiantes y desarrolladores.

EJERCICIOS PROPUESTOS

1. De modo conceptual, diseñe una serie de pruebas de caja blanca y caja negra de un programa que recibe como parámetro una serie de numéros y devuelve los números primos que hay en la lista.

2. Tenemos las siguientes clases:

CÓDIGO EJERCICIO 2a

CLASE MAXIMO

```
public static int GetMax(int[] lista) {
    int indice, max = Int32.MaxValue;
    for (indice = 0; indice < lista.Length-1; indice++) {
        if (lista[indice] > max) {
            max = lista[indice];
        }
    }
    return max;
}
```

CÓDIGO EJERCICIO 2b

CLASE MAXIMOTEST

```
[TestFixture]
public class MaximoTest
{
    public MaximoTest() { }
    [Test]
    public void testSimple() {
        Assert.AreEqual(9, Maximo.GetMax(new int[] { 3, 7, 9, 8 }));
    }
    public void testOrden() {
        Assert.AreEqual(9, Maximo.GetMax(new int[] { 9, 7, 8 }));
        Assert.AreEqual(9, Maximo.GetMax(new int[] { 7, 9, 8 }));
        Assert.AreEqual(9, Maximo.GetMax(new int[] { 7, 8, 9 }));
    }
    [Test]
    public void testDuplicados() {
        Assert.AreEqual(9, Maximo.GetMax(new int[] {9, 7, 9, 8}));
    }
    [Test]
    public void testSoloUno() {
        Assert.AreEqual(7, Maximo.GetMax(new int[] {7}));
    }
    [Test]
    public void testTodosNegativos() {
        Assert.AreEqual(-4, Maximo.GetMax(new int[] {-4, -6, -7, 22}));
    }
```

Ejecute las pruebas con NUnit y observe los resultados. En caso de que se produzcan errores, soluciónelos siguiendo el patrón de procedimiento explicado en el apartado 3.4.1. Para localizar los errores utilice puntos de interrupción condicionales.

- **3.** Implemente unas pruebas de rendimiento para el ejercicio anterior. Compare los resultados obtenidos con los ofrecidos por NUnit.
- **4.** Modifique las pruebas anteriores para valorar los resultados mediante *ticks* o ciclos de proceso.
- **5.** Diseñe y especifique pruebas del camino básico para el método *GetMax()*.
- **6.** ¿Cuál sería el AVL para las llamadas al método *GetMax()*? ¿Ha encontrado algúna ambigüedad de coherencia mientras diseñaba el AVL? De ser así, ¿cuáles?
- **7.** Utilice la clase *Maximo* para implementar la búsqueda de números primos y realice pruebas unitarias en ella. Comprueba también si el rendimiento de la aplicación o de las pruebas se ha visto afectado.
- **8.** Implemente las pruebas de rendimiento para las dos operaciones de la clase *Maximo* en las pruebas unitarias que realizas en el clase *MaximoTest*.
- **9.** Codifique un programa que cree un palíndromo a partir de una cadena dada (si la entrada es "Hola" el programa debería devolver "HolaaloH"). Diseñe e implemente las pruebas de caja blanca y de rendimiento.
- **10.** Cree una tabla de casos de prueba con los resultados obtenidos en todos los ejercicios de este bloque temático.

TEST DE CONOCIMIENTOS

1 ¿Qué es un caso de prueba?

a) Las diferentes funcionalidades disponibles para el usuario final.
b) Una relación de valores de entrada y resultados esperados.
c) Una serie de condiciones que determinarán si el programa funciona correctamente.
d) Todas las respuestas anteriores son correctas.

2 ¿Cómo se llama el procedimiento de ejecución controlada que nos permite descubrir errores en el programa?

a) Depuración.
b) Interrupción.
c) Inspección.
d) Pruebas unitarias.

3 ¿Cuál es el objetivo de modificar el conjunto de reglas del análisis de código?

a) Facilitar la detección de errores o posibles errores en el programa.
b) Adecuar los errores, advertencias y mensajes del IDE a nuestras necesidades.
c) Ninguna de las respuestas anteriores es correcta.
d) Todas las respuestas anteriores son correctas.

4 Las pruebas unitarias pertenecen a los casos de prueba de:

a) Caja blanca.
b) Caja negra.
c) Rendimiento.
d) Coherencia.

5 Si establecemos una o varias líneas del programa para controlar lo que ocurre en ese momento de la ejecución sin interrumpir el programa, estamos hablando de:

a) Puntos de ruptura.
b) Contratos de código.
c) Pruebas unitarias.
d) Ninguna de las respuestas anteriores es correcta.

6 ¿Qué recursos podríamos utilizar si queremos comprobar los valores de ciertas variables o sentencias?

a) Puntos de interrupción.
b) Puntos de seguimiento.
c) Inspecciones.
d) Todas las respuestas anteriores son correctas.

7 ¿Qué diferencia hay entre el AVL y la partición equivalente?

a) Ninguna, son diseños equivalentes de pruebas de caja negra.
b) El AVL contiene la partición equivalente.
c) La partición equivalente está contenida en el AVL.
d) Ninguna de las respuestas anteriores es correcta.

4 Optimización y documentación

OBJETIVOS DEL CAPÍTULO

- ✓ Comprender el concepto de refactorización y sus implementaciones más usuales.
- ✓ Aprender a reconocer antipatrones en nuestro software.
- ✓ Conocer el uso de un control de versiones y su aplicación en el desarrollo colaborativo.
- ✓ Aprender a comentar y documentar nuestro software.

4.1 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 Fowler* 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".

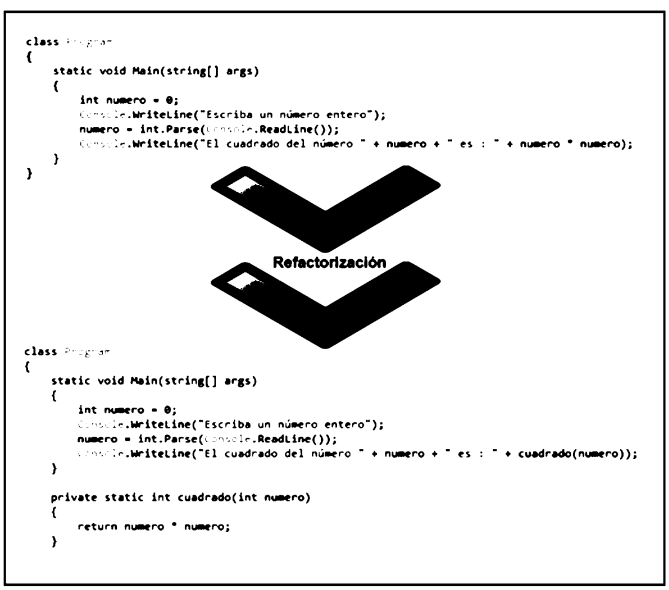


Figura 4.1. *Ejemplo de refactorización. Extracción de método en C#*

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 facilitaría 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.

4.1.1 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 4.1

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> GetHijosFromParticipantes(Persona persona) {
            IList<Persona> Hijos = new List<Persona>();
            if(participantes.Count() > 0)
                Hijos = participantes
                      .Where(p => p.padre == persona.id).ToList();

            return Hijos;
        }
    }
```

En el ejemplo 4.1, 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 4.2

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**. Recuerde que puede personalizar el formato del código mediante el menú **Herramientas > Opciones > Editor de texto** de Visual Studio.

4.1.2 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 4.3

EXTRAER 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);
}
```

- **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 4.4

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 4.5

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 4.6

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 4.7

CONSOLIDAR FRAGMENTOS DUPLICADOS EN CONDICIONALES

```
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();
```

Descomponer un Condicional

- Tenemos una complicada declaración en el condicional.
- Extraemos métodos de la condición y del cuerpo del condicional.

EJEMPLO 4.8

DESCOMPONER UN 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;
}
```

Consolidar Expresiones Condicionales

- Tenemos una secuencia de condicionales con el mismo resultado.
- Los combinamos en una sola expresión y lo extraemos.

EJEMPLO 4.9

CONSOLIDAR EXPRESIONES CONDICIONALES

```
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
```

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.

EJEMPLO 4.10

REEMPLAZAR CONDICIONAL POR POLIMORFISMO

```
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

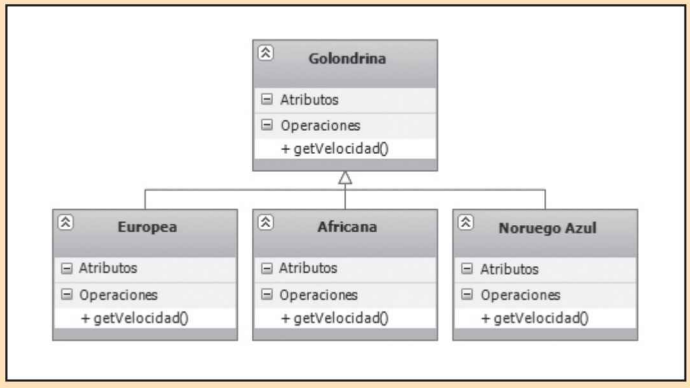


Figura 4.2. *Refactorización de reemplazo de condicional por polimorfismo*

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 4.11

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 masa * CONSTANTE_GRAVITACIONAL * altura;
}
static final double CONSTANTE_GRAVITACIONAL = 9.81;
```

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.

EJEMPLO 4.12

EREEMPLAZAR 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 Datos y Valores por Objetos

- Tenemos un atributo que necesita información o comportamiento adicional.
- Convertimos el atributo en un objeto.

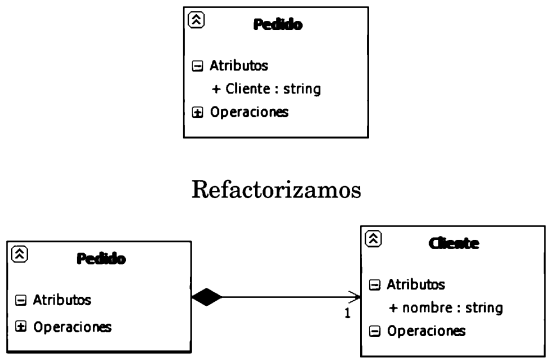


Figura 4.3. *Refactorización de reemplazo por objetos*

De este modo, hemos pasado de tener una clase Pedido con un atributo cliente, a tener una clase Pedido y una clase Cliente que tiene como atributo el nombre. En un futuro se le podrán añadir atributos y métodos a la clase Cliente para añadir la información o comportamiento que necesitemos.

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 4.13

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 4.14

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 público.
- Lo convertimos en una propiedad del objeto.

EJEMPLO 4.15

ENCAPSULAR ATRIBUTO COMO PROPIEDAD

```
public String _nombre
```

Refactorizamos

```
public virtual string _nombre {get; set;)
```

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.

Refactorizamos

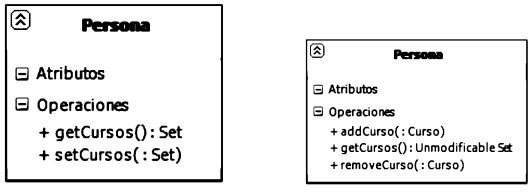


Figura 4.4. *Refactorización de encapsulación de colección*

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.

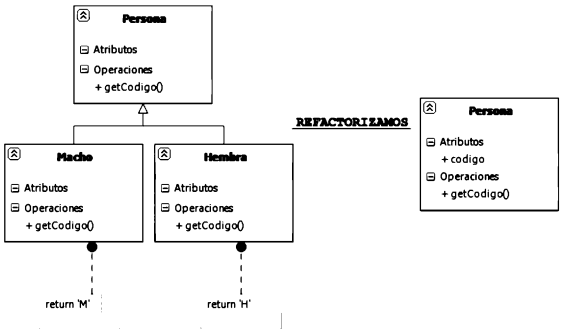


Figura 4.5. *Refactorización de reemplazo de subclases por atributos*

Extraer SubClase

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

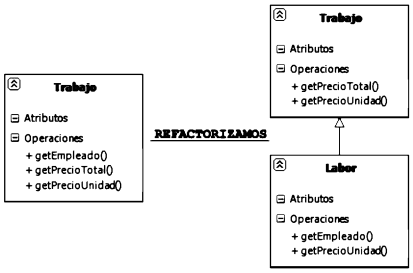


Figura 4.6. *Refactorización de extracción de subclase*

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.

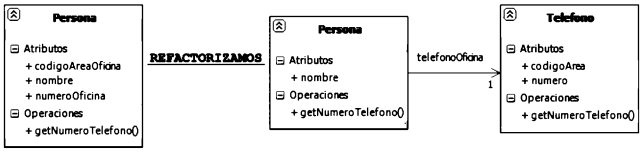


Figura 4.7. *Refactorización de reemplazo de subclases por atributos*

4.1.3 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 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".
- **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.

4.1.4 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.

4.1.5 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.

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 ⇆ obtendremos:

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

Si al final del último corchete escribimos:

```
else
```

y le damos al ⇆ obtendremos:

```
    if (true)
    {
    }
    else
    {

}
```

O si por ejemplo escribimos:

```
for
```

y le damos al ⇆ obtendremos:

```
    for (int i = 0; i < length; i++)
    {
    }
```

Dejándonos la estructura lista para ser usada y rellenada. Cabe destacar que el tabulador (⇆) 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.

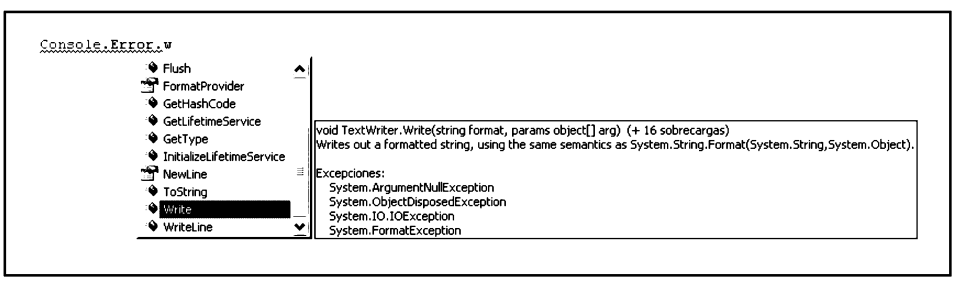


Figura 4.8. *IntelliSense de Visual Studio*

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 entrono 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.

Vamos a ver cómo se efectuarían en Visual Studio cada uno de estos patrones de refactorización.

Renombrar

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

EJEMPLO 4.16

RENOMBRAR VISUAL STUDIO

```
int num = 0;
for (int i = 0; i < 10; i++){
         if (i % 2 == 0)
              num++;
      }
Console.WriteLine("Hay " + num + " números pares del 0 al 10");
```

Hacemos botón derecho en la variable que queremos renombrar y elegimos **Cambiar nombre**.

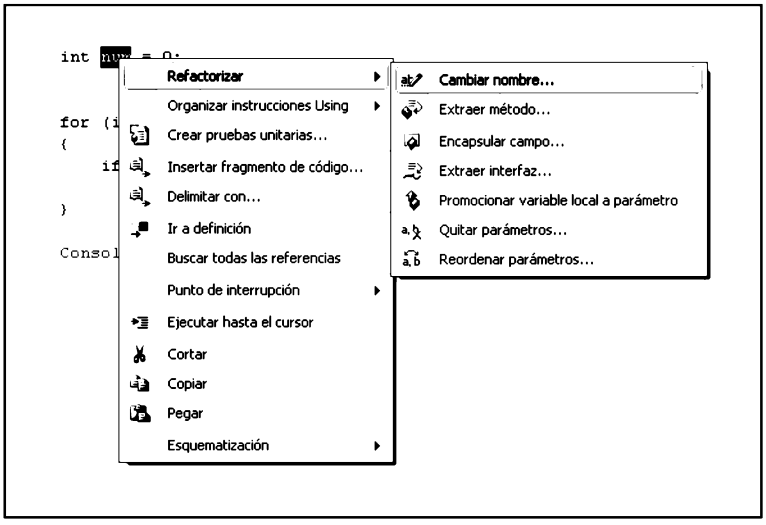


Figura 4.9. *Cambiar nombre Visual Studio*

Escribimos el nuevo nombre para nuestra variable y pulsamos **Aceptar**.

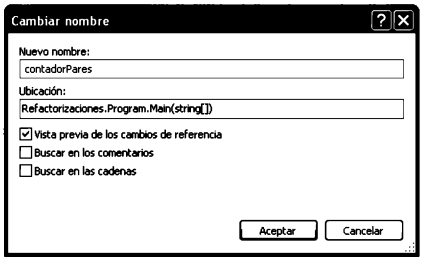


Figura 4.10. *Cambiar nombre Visual Studio*

Como hemos marcado **Vista previa de los cambios de referencia**, nos aparece una ventana que nos indica dónde se realizarán los cambios y cuál será el resultado final. Una vez comprobado que los resultados son los que queremos, pulsamos en **Aplicar**.

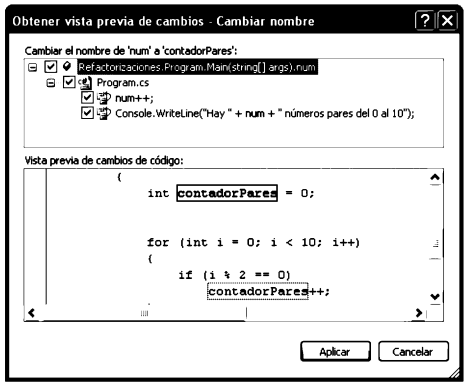


Figura 4.11. *Vista previa Cambiar nombre Visual Studio*

Obteniendo así el código con el cambio de nombre.

EJEMPLO 4.17

RENOMBRAR VISUAL STUDIO

```
int contadorPares = 0;
for (int i = 0; i < 10; i++){
         if (i % 2 == 0)
              contadorPares++;
}
Console.WriteLine("Hay "+contadorPares+" números pares del 0 al 10");
```

Extraer método

Siguiendo con el ejemplo anterior, queremos convertir nuestro algoritmo para calcular pares en un método.

Para ello, seleccionamos el fragmento de código que queremos convertir en método, hacemos botón derecho y elegimos **Extraer método**.

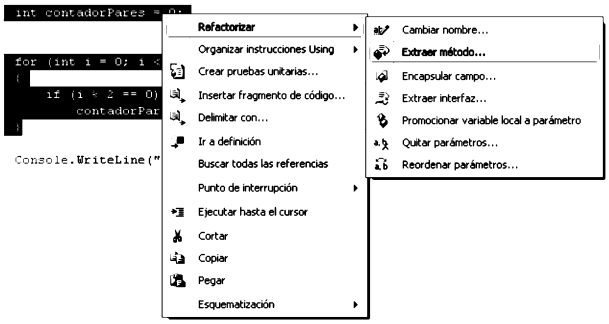


Figura 4.12. *Extracción de método con Visual Studio*

En la siguiente pantalla, escogemos el nombre del método que vamos a crear, en nuestro caso escribimos "contarPares" y pulsamos en **Aceptar**. Como resultado, obtenemos el siguiente código.

EJEMPLO 4.18

EXTRAER MÉTODO CON VISUAL STUDIO

```
int contadorPares = contarPares();

Console.WriteLine("Hay "+contadorPares+" números pares del 0 al 10");

private static int contarPares(){
int contadorPares = 0;

for (int i = 0; i < 10; i++){
if (i % 2 == 0)
  contadorPares++;
}
return contadorPares;
}
```

Podríamos mejorar esta refactorización automática, evitando esa asignación de variable, dejando el cuerpo del método *main* reducido a la siguiente línea:

```
Console.WriteLine("Hay "+ contarPares() + " números pares del 0 al 10");
```

Encapsular campo

Tenemos un sencillo programa que nos pide por teclado el nombre, apellidos y DNI de un alumno.

EJEMPLO 4.19a

ENCAPSULAR CAMPO CON VISUAL STUDIO

```
class Program {
static void Main(string[] args) {
Alumno miAlumno = new Alumno();
Console.WriteLine("Escriba el nombre del alumno");
miAlumno.nombre = Console.ReadLine();
Console.WriteLine("Escriba los apellidos del alumno");
miAlumno.apellidos = Console.ReadLine();
Console.WriteLine("Escriba el dni del alumno");
miAlumno.dni = Console.ReadLine();


Console.WriteLine("Alumno: "+miAlumno.nombre + " "+miAlumno.apellidos);
Console.WriteLine("DNI: " + miAlumno.dni);
}

}
class Alumno {
public string nombre;
public string apellidos;
public string dni;

public Alumno() { }
}
```

Accedemos a sus atributos de manera directa, es decir, no tenemos encapsulados los atributos, por lo que habría que privatizar los atributos y crear los métodos de acceso. Lo haremos utilizando la refactorización automática de Visual Studio. Seleccionamos los atributos que hay que encapsular uno a uno, hacemos botón derecho y elegimos **Encapsular campo**.

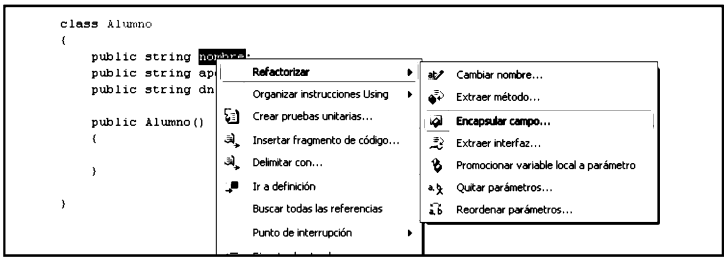


Figura 4.13. *Encapsular campo con Visual Studio*

Le damos a **Aceptar** en la siguiente pantalla, y obtendremos la encapsulación del atributo, con su declaración privada y sus métodos *get* y *set* para retornar y asignar valores al atributo.

EJEMPLO 4.19b

ENCAPSULAR CAMPO CON VISUAL STUDIO

```
class Alumno {
private string nombre;

public string Nombre {
          get { return nombre; }
set { nombre = value; }
      }
public string apellidos;
public string dni;

public Alumno() { }
}
```

Lo hacemos ahora para el resto de atributos, y tendríamos al final un código como el que se muestra a continuación.

EJEMPLO 4.19c

ENCAPSULAR CAMPO CON VISUAL STUDIO

```
static void Main(string[] args) {
Alumno miAlumno = new Alumno();

Console.WriteLine("Escriba el nombre del alumno");
miAlumno.Nombre = Console.ReadLine();
       Console.WriteLine("Escriba los apellidos del alumno");
       miAlumno.Apellidos = Console.ReadLine();
Console.WriteLine("Escriba el dni del alumno");
       miAlumno.Dni = Console.ReadLine();

       Console.WriteLine("Alumno: "+miAlumno.Nombre+" "+miAlumno.Apellidos);
       Console.WriteLine("DNI: " + miAlumno.Dni);
```

```
}
class Alumno {
private string nombre;

public string Nombre {
get { return nombre; }
set { nombre = value; }
}
private string apellidos;

public string Apellidos {
get { return apellidos; }
set { apellidos = value; }
}
private string dni;

public string Dni {
get { return dni; }
set { dni = value; }
}

public Alumno() { }
}
```

Observamos que además de privatizar los atributos y de crear las propiedades con sus *get* y *set*, nos ha renombrado los accesos a los atributos por las propiedades, utilizando el *get* o el *set* dependiendo de la operación realizada.

Extraer interfaz

Vamos a utilizar el ejemplo anterior para realizar esta refactorización. Para ello, utilizaremos una refactorización anterior, la de *Extraer método*, para crear el método que nos escriba la información del alumno por pantalla. Y, además, esa información nos la va a dar un método de la clase alumno, nuestro programa quedaría ahora del siguiente modo.

EJEMPLO 4.20a

EXTRAER INTERFAZ CON VISUAL STUDIO

```
static void Main(string[] args) {
Alumno miAlumno = new Alumno();

Console.WriteLine("Escriba el nombre del alumno");
miAlumno.Nombre = Console.ReadLine();
Console.WriteLine("Escriba los apellidos del alumno");
      miAlumno.Apellidos = Console.ReadLine();
      Console.WriteLine("Escriba el dni del alumno");
      miAlumno.Dni = Console.ReadLine();

imprimirAlumno(miAlumno);

}
private static void imprimirAlumno(Alumno alumno) {
Console.WriteLine(alumno.toString());
}
class Alumno {
private string nombre;

public string Nombre {
get { return nombre; }
set { nombre = value; }
}
private string apellidos;

public string Apellidos {
get { return apellidos; }
set { apellidos = value; }
      }
private string dni;

public string Dni {
get { return dni; }
set { dni = value; }
   }

public Alumno() { }
public string toString() {
   return "Alumno: "+Nombre+" "+Apellidos+"\n"+"DNI: "+Dni;
      }
}
```

Hemos aplicado refactorizaciones aprendidas anteriormente, ahora para extraer la interfaz, lo único que hay que hacer es situar (o seleccionar) la clase de la que se desea extraer la interfaz, hacer botón derecho y elegir **Extraer interfaz**.

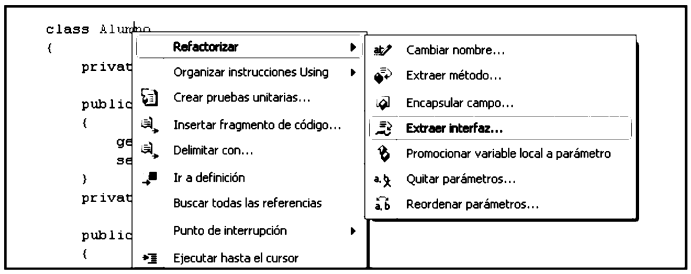


Figura 4.14. *Extraer interfaz con Visual Studio*

En la siguiente ventana, escogeremos qué campos y métodos queremos que tenga nuestra interfaz y el nombre que tendrá.

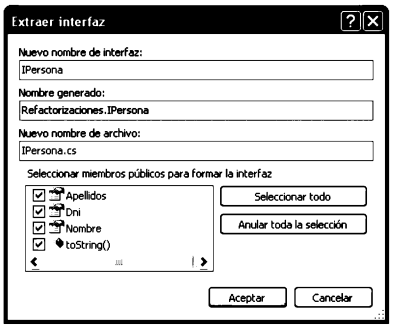


Figura 4.15. *Extraer interfaz con Visual Studio*

Lo que nos creará un archivo con nuestra interfaz, dispuesta a ser implementada por alguna otra clase elegible para implementar nuestra interfaz.

EJEMPLO 4.20b

EXTRAER INTERFAZ CON VISUAL STUDIO

```
interface IPersona {
string Apellidos { get; set; }
      string Dni { get; set; }
      string Nombre { get; set; }
      string toString();
}
```

Cuando eso ocurra y queráis implementar una interfaz, podéis hacerlo desde Visual Studio automáticamente haciendo botón derecho en la interfaz.

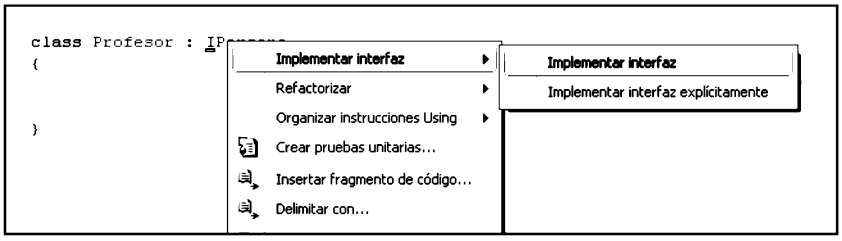


Figura 4.16. *Implementar interfaz con Visual Studio*

Lo que nos generará una implementación en donde tendremos que rellenar el código específico para esa clase de los métodos y campos implementados.

EJEMPLO 4.21a

IMPLEMENTAR INTERFAZ CON VISUAL STUDIO

```
class Profesor : IPersona {
     #region Miembros de IPersona
     public string Apellidos {
          get {throw new NotImplementedException();}
            Set {throw new NotImplementedException();}
     }
     public string Dni {
          get {throw new NotImplementedException();}
            Set {throw new NotImplementedException();}
     }
     public string Nombre {
               get {throw new NotImplementedException();}
               set {throw new NotImplementedException();}
     }
     public string toString(){
               throw new NotImplementedException();
     }
     #endregion
}
```

Lo único que tendríamos que hacer ahora sería escribir el código de nuestra clase sustituyendo las excepciones de la implementación automática que acabamos de hacer.

EJEMPLO 4.21b

IMPLEMENTAR INTERFAZ CON VISUAL STUDIO

```
[…]
public string toString() {
     return "Profesor: " + Nombre + " " + Apellidos + "\n" + "DNI: " + Dni;
}
[…]
```

Promocionar variable local a parámetro

Hemos modificado el programa usado en ejemplos anteriores para que ahora, a la vez que nos imprime por pantalla un alumno, también nos lo guarde en un archivo de texto.

EJEMPLO 4.22a

PROMOCIONAR VARIABLE CON VISUAL STUDIO

```
private static void imprimirAlumno(Alumno alumno) {
Console.WriteLine(alumno.toString());
guardaralumno(alumno);
}
private static void guardaralumno(Alumno alumno) {
const string rutaFichero = @”E:\tmp\Alumno.txt”;

StreamWriter sw = new StreamWriter(rutaFichero);
sw.WriteLine(alumno.toString());
sw.Close();
}
```

Seguramente queramos poder modificar nuestra ruta de archivo a nuestro antojo desde la invocación al método, y no que sea una constante que esté en el propio método, por lo que vamos a convertir esa variable local en un parámetro. Hacemos botón derecho en el nombre de la variable y seleccionamos **Promocionar variable local a parámetro**.

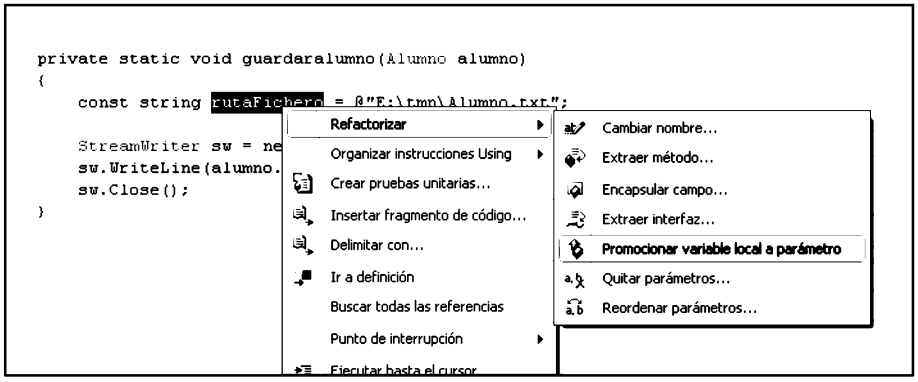


Figura 4.17. *Promocionar variable local con Visual Studio*

Automáticamente nos inserta la variable como parámetro y nos manda el valor asignado a la variable como parámetro en la invocación (o invocaciones) del método.

EJEMPLO 4.22b

PROMOCIONAR VARIABLE LOCAL CON VISUAL STUDIO

```
private static void imprimirAlumno(Alumno alumno) {
Console.WriteLine(alumno.toString());
guardaralumno(alumno, @"E:\tmp\Alumno.txt");
}
private static void guardaralumno(Alumno alumno, string rutaFichero) {
StreamWriter sw = new StreamWriter(rutaFichero);
sw.WriteLine(alumno.toString());
sw.Close();
}
```

Quitar parámetros

Hemos estado modificando el funcionamiento de un método al que se le pasaban un número determinado de parámetros, según se ha ido cambiando el método, diversos parámetros han dejado de tener utilidad dentro del método, por lo que habrá que quitarlos, ya que no se usan. Tiene el inconveniente de tener que quitar también todos esos parámetros de todas las invocaciones a dicho método.

EJEMPLO 4.23

QUITAR PARÁMETROS CON VISUAL STUDIO

```
[…]
imprimirAlumno(nuevoAlumno(miAlumno, "Pedro", "Gomez",     "4815462342L"));
imprimirAlumno(nuevoAlumno(miAlumno, "Tata", "Ogg", "314159682L"));

}
private static void imprimirAlumno(Alumno alumno) {
Console.WriteLine(alumno.toString());
guardaralumno(alumno, @"E:\tmp\Alumno.txt");
}
private static void guardaralumno(Alumno alumno, string rutaFichero) {
StreamWriter sw = new StreamWriter(rutaFichero);
sw.WriteLine(alumno.toString());
sw.Close();
}
private  static  Alumno  nuevoAlumno(Alumno  alumno,string  nombre,string
apellidos,string dni) {
Alumno nuevoAlumno = new Alumno();
nuevoAlumno.Nombre = nombre;
      nuevoAlumno.Apellidos = apellidos;
      nuevoAlumno.Dni = dni;
      return nuevoAlumno;
}
```

Como vemos, el parámetro alumno ya no se utiliza bajo ninguna circunstancia, con lo que tendríamos que eliminarlo, hacemos botón derecho en el parámetro y elegimos **Quitar parámetro**.

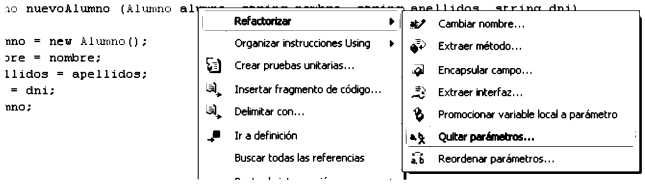


Figura 4.18. *Quitar parámetros con Visual Studio*

La refactorización automática nos quitará el parámetro de la declaración del método y también de la invocación, pero nunca toca el cuerpo del método, con lo que, si dicho parámetro sigue en uso, la refactorización provocará errores.

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

- CodeRush:

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

Como comentario adicional, el enlace que se muestra de CodeRush se corresponde con su versión Xpress y contiene una seri de ejemplos realizadas a modo de animaciones, con lo que se puede apreciar perfectamente el funcionamiento y uso de cada funcionalidad de esta herramienta.

ACTIVIDADES 4.1

- Pruebe el funcionamiento de la refactorización automática de Visual Studio Reordenar parámetros en cualquiera de los ejemplos o proyectos vistos hasta ahora.
- Pruebe el formateo automático de documentos de Visual Studio con diferentes configuraciones de formato en las opciones del editor de texto.
- Revise los ejemplos y códigos vistos hasta ahora. ¿Detecta "malos olores"? ¿Hay algún patrón de refactorización que los pueda solucionar?
- Probemos la verdadera utilidad de IntelliSense. Realice 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.

4.2 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.

4.2.1 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.

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).

Para ello, vamos a utilizar la herramienta Visual SVN Server, disponible para descarga desde su página oficial (*http://www.visualsvn.com/server/download/*), y procedemos a su instalación.

El proceso de instalación es muy sencillo, una vez aceptados los términos y condiciones y elegida la opción de instalación **Visual SVN Server and Management Console**, veremos la ventana de personalización de la instalación. En esta instalación no vamos a modificar ninguno de los parámetros, por lo que solamente tendremos que darle a **Seguir** y la instalación procederá a instalar los archivos y servicios necesarios en nuestro equipo.

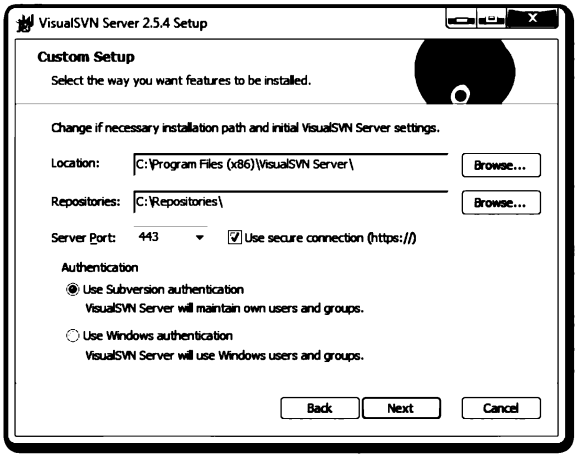


Figura 4.19. *Quitar parámetros con Visual Studio*

Una vez finalizada la instalación, se ejecutará el Visual SVN Manager y podremos proceder a configurarlo.

Empezaremos creando un nuevo repositorio haciendo clic con el botón derecho en **Repositories** y eligiendo la opción **Create New Repository**.

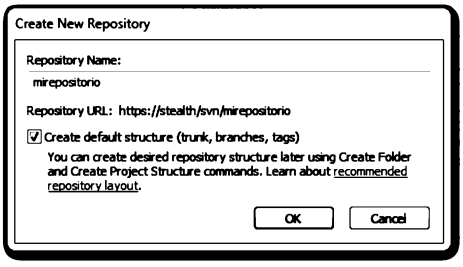


Figura 4.20. *Crear nuevo repositorio en Visual SVN*

Marcaremos la opción de crear la estructura de directorios por defecto y escribiremos el nombre de nuestro repositorio.

Crearemos ahora un nuevo usuario para poder loguearnos en el repositorio, para ello, haremos clic con el botón derecho en **Users** y elegiremos la opción **Create User**.

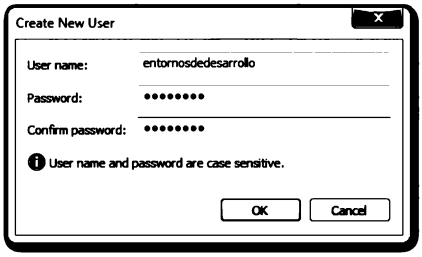


Figura 4.21. *Crear un nuevo usuario en Visual SVN*

Escribiremos nuestro nombre de usuario y nuestra contraseña. Con estos sencillos pasos ya tendríamos creado nuestro repositorio y debería ser perfectamente funcional, para ello, utilizaremos cualquier navegador para navegar por la ruta que nos indicó a la hora de crear el repositorio, que lleva el siguiente formato:

https://nombredelequipo/svn/nombredelrepositorio

Deberíamos cambiar "nomredelequipo" por el nombre de nuestro equipo o la dirección IP del mismo, y el nombre del repositorio por el nombre que le dimos a nuestro repositorio al crearlo.

Si todo ha funcionado correctamente, al acceder a esa dirección nos pedirá un usuario y una contraseña. Escribiremos los datos que rellenamos al crear el usuario y tendría que aparecer una ventana como ésta:

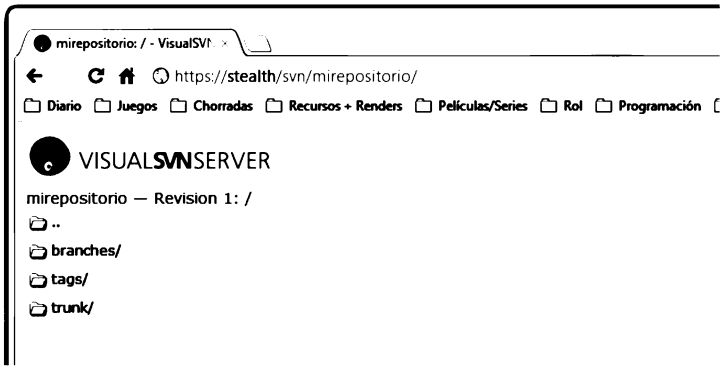


Figura 4.22. *Acceso web al repositorio de Visual SVN*

4.2.2 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.

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://msmvps.com/blogs/ffagas/archive/2009/08/31/vsts-2010-instalaci-243-n-del-team-foundation-server-2010.aspx*.

Cliente de control de versiones: ANKH SVN

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.collab.net/downloads*.

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**.

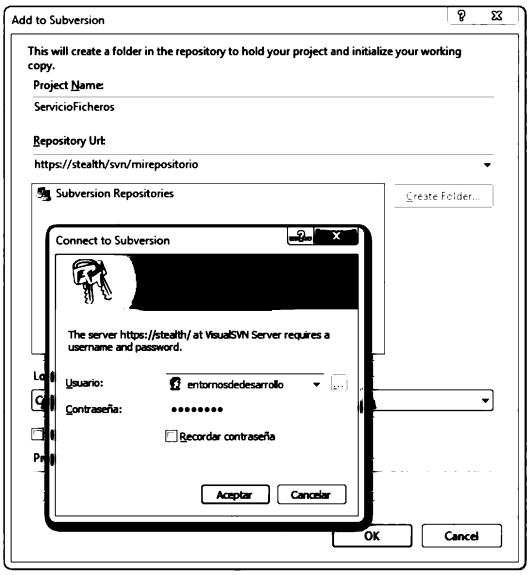


Figura 4.23. *Añadir una solución al repositorio*

En la ventana emergente que nos aparecerá, escribiremos el nombre del repositorio que creamos previamente con el Visual SVN, nada más escribirlo, nos aparecerá una ventana que nos pedirá nuestro nombre de usuario y contraseña (señal inequívoca de que todo funciona correctamente) y nos conectaremos al repositorio.

Elegiremos a continuación una carpeta del repositorio, por defecto se suele usar "trunk" como carpeta donde albergar el núcleo de nuestro software, por lo que será ésa la carpeta que elegiremos y haremos clic en **OK**.

Como último paso, nos aparece una ventana donde podremos escribir una entrada asociada a la primera subida al repositorio antes de asociar la solución al repositorio.

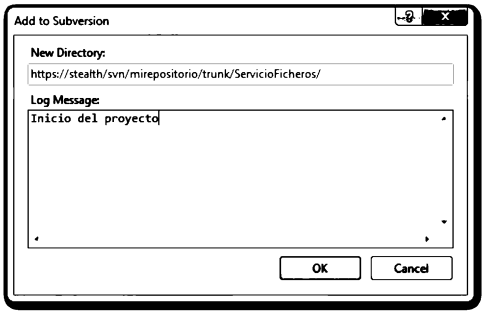


Figura 4.24. *Añadir un mensaje de inicio*

Nuestra solución ahora está asociada al repositorio, podemos ver como en el explorador de proyectos nos aparecen unos iconos de "**+**" al lado de cada archivo, que nos indican que esos archivos serán subidos como archivos nuevos la próxima vez que actualicemos el repositorio.

Nuestro proyecto no está todavía subido al repositorio, por lo que podemos trabajar antes de realizar nuestra primera actualización sin que el control de versiones perciba ningún cambio, aún es todo nuevo para él. Una vez que esté listo para subirlo al repositorio, haremos clic con el botón derecho en la solución desde el explorador de soluciones y seleccionaremos la opción **Commit solution changes**, donde nos aparecerá una ventana con todos los cambios que se van a realizar en el repositorio, en nuestro caso todos los archivos salen marcados como "New", ya que será nuestra primera subida al repositorio. Una vez confirmemos la operación, comenzará la subida de ficheros, y los archivos que antes estaban marcados con un símbolo "**+**" de color azul ahora tienen un icono de verificación también de color azul que nos indica que el proyecto está perfectamente sincronizado con el repositorio.

Actualizar ficheros locales a una versión específica

Cada vez que queramos actualizar el repositorio, tendríamos que realizar la misma operación. Pero ¿y si lo que queremos es hacer la operación contraria? Es decir, actualizar nuestra copia de trabajo con una versión existente en el repositorio.

Una vez más, haremos clic con el botón derecho en nuestra solución y elegiremos el submenú **Subversión** y, dentro de él, la opción **Update to specific version**. Nos aparecerá una ventana donde, entre otras opciones, hay una lista desplegable con el rótulo **Type**. En ella, 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.

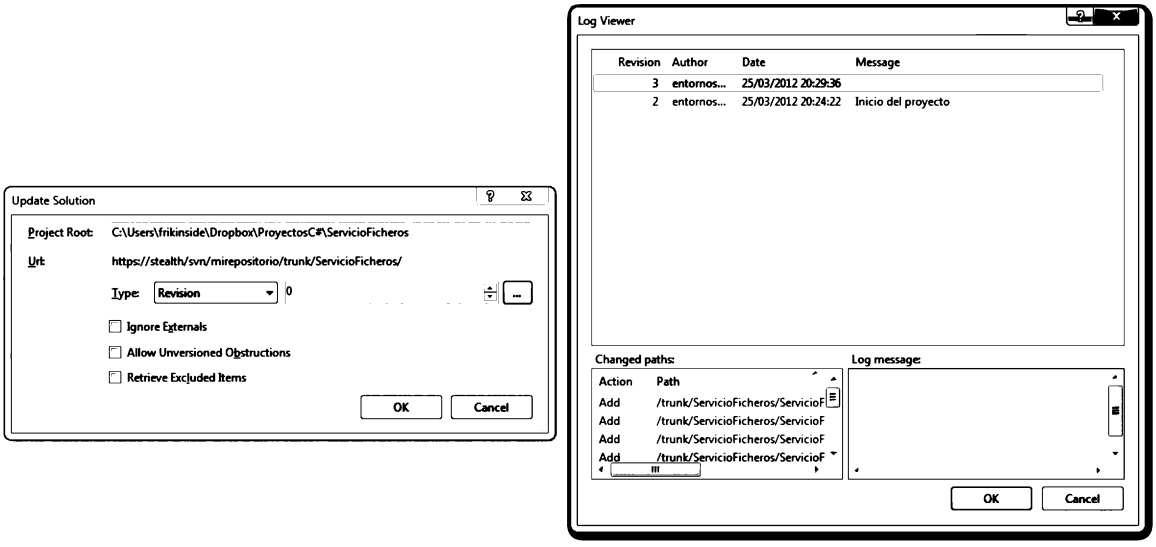


Figura 4.25. *Actualizar a una versión específica*

No tenemos mucho donde elegir, por lo que escogeremos nuestra última versión, donde subimos todos los ficheros a nuestro repositorio. El proyecto se quedaría igual que como estaba, ya que no hemos realizado ningún cambio.

Ahora ya sabemos cómo utilizar AnkhSVN para actualizar o descargar código del repositorio permitiéndonos llevar un control de nuestro código.

ACTIVIDADES 4.2

- Modifique el código de tu proyecto sin guardar los cambios. ¿Observa algún cambio en el explorador de soluciones?, ¿y si guarda o compila?
- Realice cambios en el código que tiene vinculado al control de versiones y actualice el repositorio. ¿Qué ocurre si actualizamos a una versión anterior?, ¿podemos volver a la última versión que hemos actualizado? Reflexione sobre las posibilidades de los resultados obtenidos.
- [OPCIONAL] Si dispone de un servidor TFS, intente realizar las operaciones de actualización y subida de código utilizando los conocimientos adquiridos con el uso de AnkhSVN.

4.3 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.

4.3.1 USO DE COMENTARIOS

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.

EJEMPLO 4.24

COMENTARIOS

```
// Esto es un comentario de línea
int i = 0;
/*
* Esto es un comentario de bloque
* Sólo es necesario establecer el
* principio y el fin de bloque
*
* */
```

Como vemos en el ejemplo, los comentarios de bloque nos ahorran tener que comentar las líneas una a una cuando queremos comentar toda una porción de código.

Si dejamos de lado el "mal olor" que define a los comentarios como una falla de código propio descriptivo, diríamos que los comentarios se utilizan para clarificar algún algoritmo complicado, además de las opciones que hemos comentado previamente. Además, también se suelen utilizar en las pruebas de caja blanca, explicando entre otras cosas el porqué de los datos utilizados y de los resultados esperados.

Comentarios de documentación

Además de los dos tipos de comentarios que hemos visto en el epígrafe anterior, 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.

Los comentarios de documentación suelen establecerse en los IDE como un disparador que se activa mediante una combinación de teclas, un modo de autorellenar el formato del comentario de la manera apropiada de un modo similar al uso que utilizábamos para que el IDE nos crease la estructura de diferentes estructuras mediante la pulsación de la tecla ⇆.

En Visual Studio, para que nos rellene ese formato se utiliza la triple pulsación del carácter / (///).

EJEMPLO 4.25

COMENTARIOS 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) {
   […]
}
```

Con solo escribir los tres caracteres del comentario (///) nos aparece automáticamente la estructura que vemos rellenada arriba. Observamos también que es un formato estructurado como el XML y su funcionamiento real es muy similar. También podríamos añadir etiquetas propias personalizadas o utilizar etiquetas interpretadas por el entorno de desarrollo como *<example>*, *<remarks>*, *<exception>*, *<value>* o *<list>*. Algunas de esas etiquetas (como *<exception>*) deben tener una sintaxis determinada y serán revisadas por el analizador sintáctico de Visual Studio.

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.

4.3.2 HERRAMIENTAS

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.

Para ello contamos con una sencilla y eficaz herramienta, VSDocMan, disponible como extensión para Visual Studio, en este caso, la extensión debe ser descargada de manera externa, por lo que lo descargaremos de la siguiente dirección:

http://www.helixoft.com/common/product-downloads.html

En esa página, también veremos que VSDocMan es un programa de pago y que gratuitamente solo nos ofrecen una versión trial limitada al uso temporal, existen alternativas gratuitas como SandCastle, pero están teniendo problemas con la última versión del FrameWork .NET y el resultado es en ocasiones inesperado, por ello, seguiremos utilizando VSDocMan, que tiene además un funcionamiento extremadamente efectivo.

Una vez descargado e instalado, ya estaría disponible para su uso en Visual Studio. Nos aparece un nuevo icono en nuestra interfaz de usuario, que utilizaremos para ejecutar la aplicación. Primero cargaremos un proyecto que tenga nuestros comentarios de documentación y haremos clic en dicho icono.

Nos aparecerá una ventana de configuración para la documentación que queremos crear.

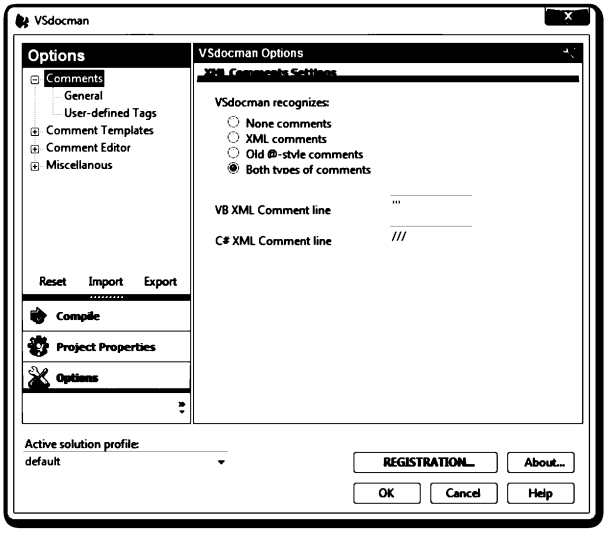


Figura 4.26. *Ventana de configuración de VSDocMan*

En esa ventana vemos diversos parámetros de configuración, que se utilizarán para formatear el documento final y para especificar cuál será la fuente y el formato de donde sacará la información para crear la documentación del proyecto.

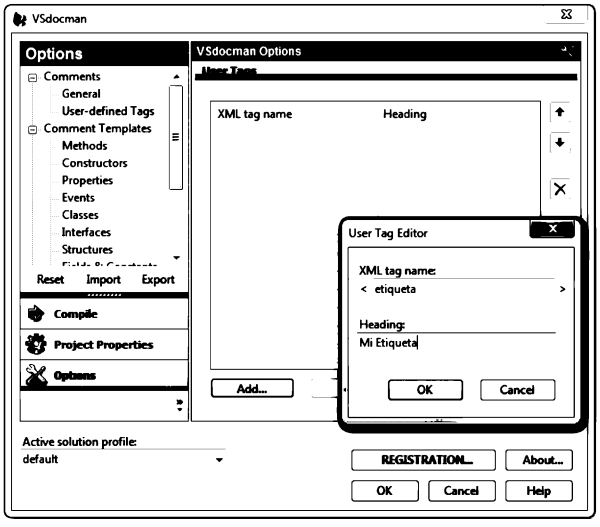


Figura 4.27. *Creación de etiquetas personalizadas en VSDocMan*

Nuestra documentación es muy básica y no habría que realizar ninguna configuración adicional. En caso de que hayamos incluido etiquetas personalizadas en nuestros comentarios, podremos añadirla en la configuración de los comentarios, en el apartado **User-defined Tags**, para que VSDocMan la reconozca.

Una vez que lo tenemos configurado completamente, haremos clic en **Compile** en la ventana principal para que comience el proceso de creación de la documentación.

Solo nos quedaría hacer clic en el botón **Show** una vez que el programa haya terminado para ver la documentación que hemos creado de una manera rápida y sencilla aprovechando los comentarios de documentación.

RESUMEN DEL CAPÍTULO

En este capítulo hemos aprendido la importancia de un código modificable, flexible y entendible, centrándonos en reconocer los diferentes problemas que puede tener nuestro código y diversas formas de solucionarlos.

Hemos aprendido el uso de varias herramientas internas y externas al entorno de desarrollo, que nos permiten agilizar y mejorar el proceso de codificación de software, y a aprovecharlas al máximo automatizando procesos que nos llevarían mucho tiempo de tener que hacerlo de modo manual.

Nos hemos adentrado más en el mundo del trabajo colaborativo y el control de versiones, aprendiendo a crear servidores virtuales para el uso personal o compartido y a llevar un control inteligente de todas las revisiones de nuestro código, pudiendo acceder a cualquier versión anterior de nuestro proyecto de manera rápida y sencilla.

La utilidad de los comentarios no debe ser desestimada, incluso si caemos en el “mal olor” de los comentarios, no debemos ser demasiado estrictos en ese aspecto, ya que hemos comprobado su verdadera utilidad y la necesidad de utilizarlos en determinadas ocasiones.

TEST DE CONOCIMIENTOS

1 ¿Qué es un "mal olor"?
a) Antipatrones de código.
b) Fallos en el funcionamiento del programa.
c) Un código mal documentado.
d) Ninguna de las respuestas anteriores es correcta.

2 ¿Con cuántos repositorios podemos vincular nuestro código?
a) Uno.
b) Dos.
c) Con tantos como se quiera.
d) Depende del control de versiones.

3 ¿Cómo podemos tener un código más claro y entendible?
a) Mediante la refactorización.
b) Utilizando sangrados y tabulados.
c) Ninguna de las respuestas anteriores es correcta.
d) Todas las respuestas anteriores son correctas.

4 La herramienta Team Foundation Server es:
a) Un gestor de refactorizaciones.
b) Una herramienta de documentación.
c) Un sistema de control de versiones.
d) Todas las respuestas anteriores son correctas.

5 ¿Debemos tener acceso a un repositorio para usar un sistema de control de versiones?
a) No.
b) Sí.
c) Solo es necesario si queremos realizar el control de versiones fuera del equipo.
d) No, el repositorio es una herramienta de documentación.

6 ¿El uso de comentarios es siempre una práctica recomendable?
a) Sí.
b) Solo los comentarios de documentación.
c) No, a veces implica que nuestro código no es lo suficientemente descriptivo por sí solo.
d) Solo se deben usar comentarios para crear la documentación o para omitir instrucciones en el proceso de depuración.

5 Diseño orientado a objetos. Diagramas de clase

OBJETIVOS DEL CAPÍTULO

- ✓ Aprender lo que es el estándar UML y su importancia en el diseño de software.
- ✓ Manejar herramientas de modelado para crear diagramas UML.
- ✓ Comprender el concepto de los diagramas de clase y su utilidad.
- ✓ Saber diseñar una estructura de clases a partir de la descripción de un problema o requisitos de un software.

5.1 INTRODUCCIÓN A UML

Para realizar labores de diseño de software, es vital realizar modelados o diagramas representando gráficamente la estructura de la aplicación y su funcionalidad, de una manera fácil de entender y rápida de crear. Antiguamente, los diagramas de cada diseñador eran únicos, ya que, salvo un par de diagramas conocidos por todos (como los diagramas de flujo o los diagramas de entidad-relación), cada diseñador o analista realizaba los diagramas de la manera que mejor los entendía, por lo que podría suponer un problema si varias personas tenían que entender los diagramas que uno o varios diseñadores les presentaban para definir el software.

Con el fin de evitar ese problema se creó UML (*Unified Modeling Language*), un conjunto unificado de estándares para las diferentes necesidades y usos que un diseñador pudiera tener a la hora de plantear una representación gráfica de un programa. Son diagramas de propósito general que se presuponen conocidos por todos, con unas técnicas de notación conocidas, de modo que cualquiera pueda crear diagramas entendibles por todos.

UML es un lenguaje de modelado y, como hemos dicho, estandarizado, tiene su aplicación más importante en el desarrollo de software, siendo extremadamente útil para dar soporte a una gran cantidad de metodologías de software, pero no de modo restrictivo, es decir, un diagrama UML solo define una semántica mediante una serie de reglas y notaciones, pero no especifica cuál sería la metodología o procedimiento que hay que usar.

UML ha pasado por diferentes versiones hasta convertirse en lo que es hoy en día, actualizándose y mejorando en cada paso, adaptando y soportando un cada vez más variado elenco de técnicas de diseño. Principalmente, podemos ver que hay tres etapas en el progreso de los diagramas UML: UML1.0, UML1.x y UML 2.0. La versión actual (UML 2.0) está siendo soportada y respaldada por OMG (*Object Management Group*), quienes deciden las características y notaciones del lenguaje.

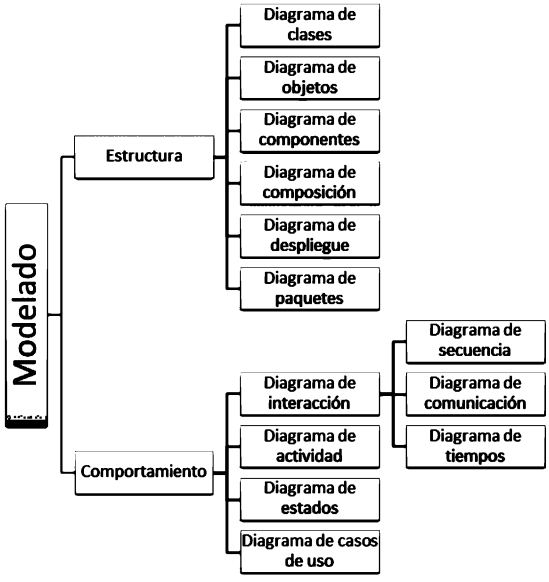


Figura 5.1. *Superestructura UML 2.0*

En la versión 2.0 de UML se definió un completo árbol de superestructuras donde se relacionaban y complementaban todos los tipos de diagramas UML a la hora de definir un software por completo. Con esa nueva especificación de estructuras y una actualización en las notaciones de los diagramas, se observaba de manera más clara el funcionamiento y estructura de un modelado completo de UML para un software.

A pesar de prescribir unas notaciones y semánticas estandarizadas, UML sigue siendo criticado por algunos grupos al no tener unas reglas lo suficientemente estrictas como para evitar problemas de interpretación. Por muy bien especificadas que estén las notaciones y semántica del diagrama, muchos diagramas tendrán que ser interpretados, no todo el mundo podrá realizar la misma implementación partiendo de los mismos diagramas. No obstante, podemos confirmar que la aplicación resultante debería tener un funcionamiento y un ciclo de vida idéntico. No se puede pretender que un diagrama no deba ser interpretado, aunque ocasione interpretaciones diferentes, un diagrama no deja de ser una guía, un recurso utilizado en la fase de diseño de un software, y, como es de esperar, el software final no será una traducción directa de dichos diagramas por muy bien trabajados que estén.

Teniendo presente la superestructura de diagramas de UML, podríamos pensar que la tarea de definir y realizar dichos diagramas (y que por supuesto sean coherentes entre sí) sería una tarea abrumadora, pero no es necesario ni se suelen realizar todos los diagramas para modelar un software, por norma general se utilizan solo una serie de diagramas para modelarlo, lo más clásico sería la tríada diagrama de clases, de secuencia y de casos de uso.

En este capítulo vamos a ver el diagrama más básico y sencillo de todos, el diagrama de clases, que tiene un gran parecido con el diagrama clásico de entidad-relación. Además, es uno de los diagramas que sí tienen una traducción directa, es decir, cuando implementas un diagrama de clases, la implementación será idéntica a la realizada por otro programador.

5.2 DISEÑO DE CLASES EN UML

Uno de los diagramas más básicos e importantes que realizaremos en nuestras labores de diseño de software serán los diagramas de clases. Se basan en ciertas reglas y notaciones sencillas para relacionar las clases y sus diferentes operaciones entre sí. En la programación orientada a objetos son un recurso básico y recurrente, usado para mostrar los bloques de construcción de cualquier sistema. Se utilizan principalmente para describir la capa del modelo y las relaciones entre las entidades del sistema.

5.2.1 CLASES, ATRIBUTOS Y MÉTODOS

Cada bloque dentro del diagrama representa a una clase; una clase, como sabemos, dispone de unos atributos y de unos métodos asociados. Representaríamos las clases como cajas en donde escribiríamos sus atributos y sus métodos.

Las clases representan a nuestros objetos, los atributos definen las propiedades y características del objeto y los métodos especifican las acciones que podemos realizar con dicho objeto.

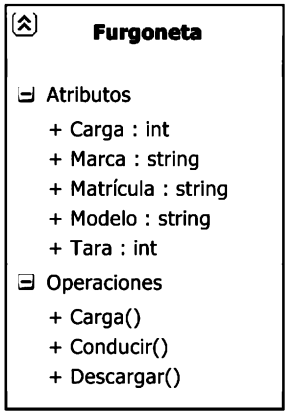


Figura 5.2. *Clase Furgoneta en UML*

Estos diagramas son particularmente útiles en el desarrollo de la capa del modelo, para mapear de un modo gráfico las entidades y sus relaciones en nuestra aplicación, ese tipo de diagramas tienen la particularidad de que no muestran las acciones que se realizan sobre cada objeto, ya que solo representan la entidad y se encuentran separadas las acciones de los datos, por lo que nuestro diagrama solo mostraría las clases y sus atributos. Dependiendo del lenguaje utilizado, las clases sí que tendrían operaciones, como por ejemplo los métodos *set* y *get* utilizados para obtener y establecer los atributos de los objetos siempre y cuando los atributos estén encapsulados. En C# realizaríamos esa operación definiendo los atributos como propiedades.

Vamos ahora a utilizar un ejemplo muy sencillo que iremos ampliando según nos surjan las necesidades en la aplicación. En este caso se trata de una libreta de contactos, donde podremos escribir el nombre, email, teléfono y dirección de cada contacto.

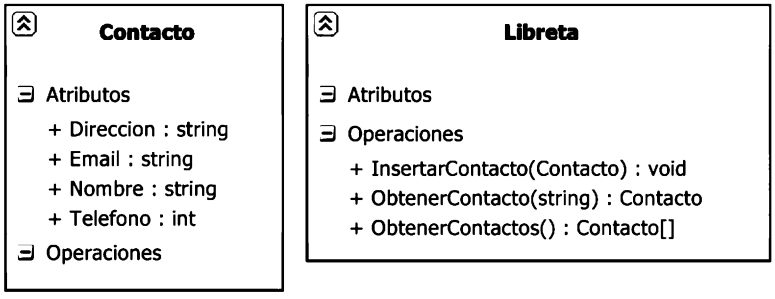


Figura 5.3. *Clases de la aplicación Libreta*

Obervamos que, en este caso, tenemos dos clases, y que, por las características de nuestra aplicación, una de las clases tiene atributos y la otra solo tiene métodos. Eso es debido a que nuestra clase Libreta albergará los datos, y esos datos son de contactos, por lo que al tener una única libreta para nuestros contactos, y teniendo en cuenta que la libreta solo tiene datos de los contactos, la libreta no tiene información adicional de la contenida en los contactos y los contactos no tienen ninguna operación que realizar sobre ellos. Faltaría la creación de contactos, pero de ese

asunto se encarga el constructor de la clase Contacto, por lo que no es necesario especificarlo, ya que es intrínseco a la propia clase.

Notación

Como hemos dicho al principio del capítulo, los diagramas UML responden a un estándar global prefijado de antemano, por lo que las notaciones son estandarizadas y es necesario conocerlas para que nuestros diagramas sean entendidos por todos.

En primer lugar, debemos percatarnos de que las clases se definen mediante cuadrados, y dichos cuadrados se dividen en tres segmentos horizontales.

Primero tendríamos el nombre de la clase y por tanto de nuestro objeto, seguidamente irían los atributos y, al final, los métodos de la clase. Es importante resaltar que si alguna clase no posee atributos propios o métodos propios, se deberá seguir dibujando el segmento que le corresponde, incluso si está vacío.

Vemos además que la especificación de los atributos y los métodos no se termina simplemente escribiendo su nombre, sino que llevan un carácter como prefijo y un tipo de dato separado por dos puntos. Además, los métodos pueden llevar un tipo de dato en los paréntesis. El carácter que precede al nombre del atributo o método se corresponde con su grado de comunicación.

- + : Público, el elemento será visible tanto desde dentro como desde fuera de la clase.
- - : Privado, el elemento de la clase solo será visible desde la propia clase.
- # : Protegido, el elemento no será accesible desde fuera de la clase, pero podrá ser manipulado por los demás métodos de la clase o de las subclases.

Después del nombre del método o atributo, tendríamos un tipo de dato de devolución separado por ":", de este modo podemos saber de qué tipo de dato estamos hablando y cuál será el valor de retorno, tanto de los atributos como de los métodos, es decir, cuando accedamos a alguno de los elementos de clase, obtendremos un tipo de dato (si procede, los métodos void no tienen valor de retorno), el tipo de dato bien puede ser propio del lenguaje o plataforma desde la que trabajamos o, como podemos ver en el ejemplo de la libreta, puede ser un objeto de nuestra propia aplicación.

Por último, tendríamos el tipo de dato que se encuentra entre paréntesis, como bien habréis podido adivinar, se trata del tipo de dato que espera nuestro método como parámetro, al igual que con los valores de retorno pueden ser de cualquier tipo de dato, ya sea del lenguaje o propio.

5.2.2 RELACIONES

Si analizamos con atención el diagrama de la libreta, podríamos percatarnos de que algo falla, a nuestro diagrama le falta algo, hemos especificado que la libreta contiene contactos, pero en ningún sitio lo hemos especificado, bien podríamos decir que la libreta tendría una lista de contactos, pero ésta no aparece por ningún lado. Realmente no se trata de un error, ni de un fallo en el diseño de la aplicación, se trata de evitar información redundante.

En el diagrama de clases, además de dibujar y representar las clases con sus atributos y métodos, también se representan las relaciones y, por ese motivo, esa lista quedaría implícita en la relación entre las clases.

Cardinalidad	Significado
1	Uno y solo uno
0..1	Cero o uno
X..Y	Desde X hasta Y
*	Cero o Varios
0..*	Cero o Varios
1..*	Uno o Varios

Las relaciones se representan con flechas con una forma determinada, y, además, poseen una *cardinalidad*. La cardinalidad es un número o símbolo que representa al número de elementos de cada clase en cada relación, pudiendo usar el comodín "*" para definir un número indeterminado de elementos.

Asociación

La asociación es la más básica de las relaciones, no tiene un tipo definido y puede ser tanto una composición como una agregación, además una asociación podría implicar únicamente un uso del objeto asociado. Se representan mediante una flecha simple que también puede tener una cardinalidad.

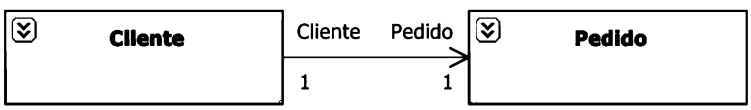


Figura 5.4. *Relación de asociación*

Composición

La composición define los componentes de los que se compone otra clase, se define además que la clase que contiene la composición no tiene sentido de existencia si la(s) agregada(s) desaparece(n). Mediante esta relación, denotada por una flecha con un rombo relleno en una de sus puntas, se indica la presencia de las clases origen en la clase destino, donde la clase destino es apuntada por el rombo de la relación.

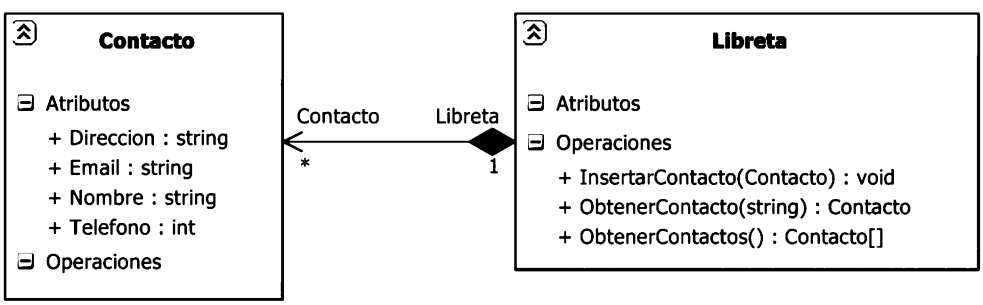


Figura 5.5. *Relación de composición entre Libreta y Contacto*

Ahora ya tenemos representada en nuestra libreta a la lista de contactos. Como vemos, al utilizar la cardinalidad uno a muchos (1 - *), donde el * se encuetra en la clase Contacto y el 1 en la clase Libreta, se especifica que una Libreta se compone de muchos Contactos.

ACTIVIDADES 5.1

- Nuestra libreta se complica, ahora se quiere poder añadir grupos de contactos. Los grupos de contactos contendrán contactos y la libreta contendrá tanto grupos de contactos como contactos. Los contactos, además, no necesitan estar en un grupo de contactos para estar en la libreta. Dibuje y represente los cambios mencionados usando como base el diagrama de clases de ejemplo de Libreta.

Agregación

Nuestra libreta sigue complicándose, ya que ahora se define la posibilidad de tener una cuenta de usuario para acceder a nuestra libreta de contactos. Las cuentas tendrán un nombre de cuenta y un email, y podrán acceder a la libreta de direcciones. Para representar la relación de esta nueva clase, vamos a utilizar una agregación.

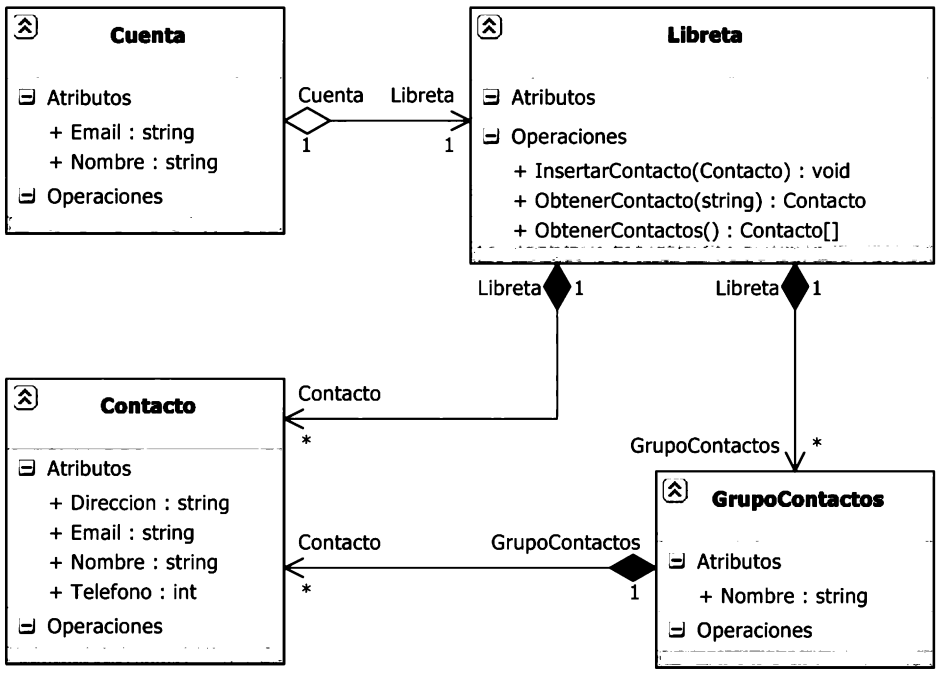


Figura 5.6. *Relación de agregación entre Cuenta y Libreta*

La diferencia principal entre la agregación y la composición reside en que la agregación no implica que la clase agregada necesite una instancia de la otra clase, es decir, la cuenta no necesita tener una instancia de la clase Libreta y libreta tampoco necesita una instancia de la clase Cuenta. Sólo implica que la clase Cuenta accede a la clase Libreta. En ocasiones esa diferencia no resulta tan clara como debería, es por ello que en un primer boceto del diagrama de clases se suelen representar estas relaciones como relaciones de asociación, para más adelante "reforzarlas" mediante la relación apropiada, recordemos que las relaciones de composición y agregación son solo una versión más fuerte y específica de las relaciones de asociación.

Herencia

La herencia es un modo de representar clases y subclases, es decir, clases más específicas de una general, se representan mediante una flecha con una punta triangular vacía.

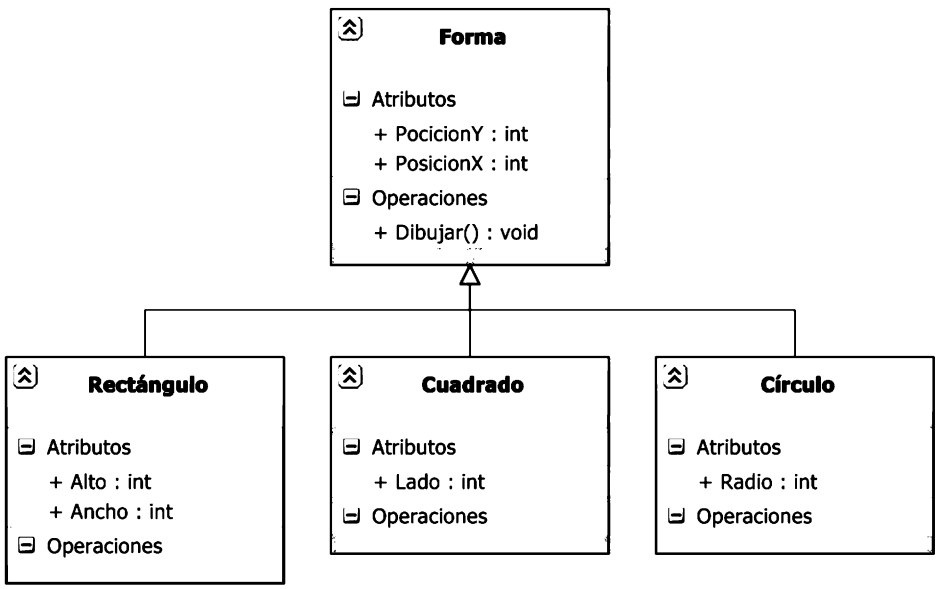


Figura 5.7. *Relación de herencia entre figura y sus derivadas*

Con este diagrama, representamos que Rectángulo, Cuadrado y Círculo son formas y por tanto tienen una posición en el espacio y se pueden dibujar, además, cada una de ellas tienen propiedades que las hacen diferentes a las demás. Con la herencia, nos evitamos además tener que definir los métodos "Dibujar" de las clases heredadas, pues se encuentra implícito en el diagrama que cualquiera de ellas puede ser invocada para ser dibujada.

5.3 HERRAMIENTAS

A pesar de que la creación de los diagramas de clase sea una tarea sencilla, al ser realizados en la fase de diseño, se pierde mucho tiempo recolocando y pasando a limpio todas las ediciones que hemos realizado a nuestro boceto inicial, para evitar este tipo de problemas se utilizan programas que nos permiten mover las clases y crear las relaciones de una manera muy rápida, sencilla y por supuesto modificable.

5.3.1 HERRAMIENTA DE MODELADO DE VS

Integrado en nuestro entorno de desarrollo, tenemos una potente herramienta CASE (*Computer Aided Software Engineering*) que utilizaremos para crear los diagramas de clases y posteriormente usaremos para modelar diversos tipos de diagramas UML diferentes.

En cualquier proyecto de Visual Studio podemos agregar un diagrama como se haría con cualquier otro componente, pero podemos crear en su defecto un proyecto específico para crear los diferentes modelados que necesitemos. Para aprender el funcionamiento de la herramienta de modelado utilizaremos un proyecto nuevo de modelado. Para ello, iremos a la entrada de menú **Arquitectura > Nuevo Diagrama** para que nos aparezca el cuadro de diálogo de creación de diagramas. Elegiremos la plantilla **Diagrama de clases UML** y haremos clic en **Aceptar**. Nos creará una pantalla en blanco con las instrucciones necesarias para empezar a trabajar.

Tal y como bien nos indica Visual Studio, iremos primero al cuadro de herramientas y elegiremos el elemento **Class**.

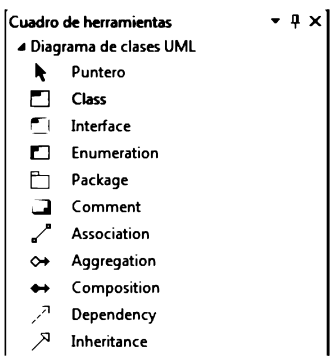


Figura 5.8. *Cuadro de herramientas de modelado de Visual Studio*

Lo añadiremos en cualquier punto de nuestro cuadro de dibujo y veremos que nos aparece un pequeño recuadro con el rótulo *Class1*, que sería el nombre de nuestra clase, la llamaremos *Familia*, repetiremos la operación creando otra clase llamada *Persona*. Deberemos añadir los atributos que correspondan para que el diagrama tenga sentido, es decir, sabemos que una familia es un conjunto de personas.

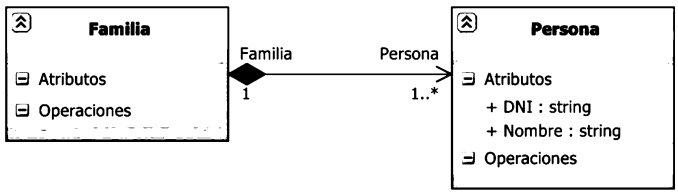


Figura 5.9. *Uso de la herramienta de modelado*

Mientras añadimos los atributos, nos damos cuenta de que Visual Studio nos rellena automáticamente el acceso si no lo hemos definido. Lo mismo ocurre con las operaciones o métodos, que nos escribe los paréntesis si no lo hemos hecho nosotros, pero tenéis que tener cuidado, porque si no escribís los paréntesis pero sí definís el tipo de retorno, os pondrá los paréntesis después del tipo de dato, lo que sabemos que es una notación incorrecta.

Cuando añadáis las relaciones, debéis tener presente que se deben añadir desde la clase contenedora a la clase contenida, mientras que la herencia se hace desde la clase hija a la clase padre.

Vamos a realizar también una relación de herencia, ya que sabemos que una familia se compone de un padre, una madre e hijos, es decir, las personas incluidas en una familia están relacionadas entre sí.

Dejando de lado la poligamia, y la orfandad, suponemos que una familia debe tener un solo un padre y una sola madre, y que ellos pueden tener varios niños en conjunto.

ACTIVIDADES 5.2

- Intente, por su cuenta, realizar el diagrama de lo que hemos explicado de las relaciones familiares antes de proseguir.

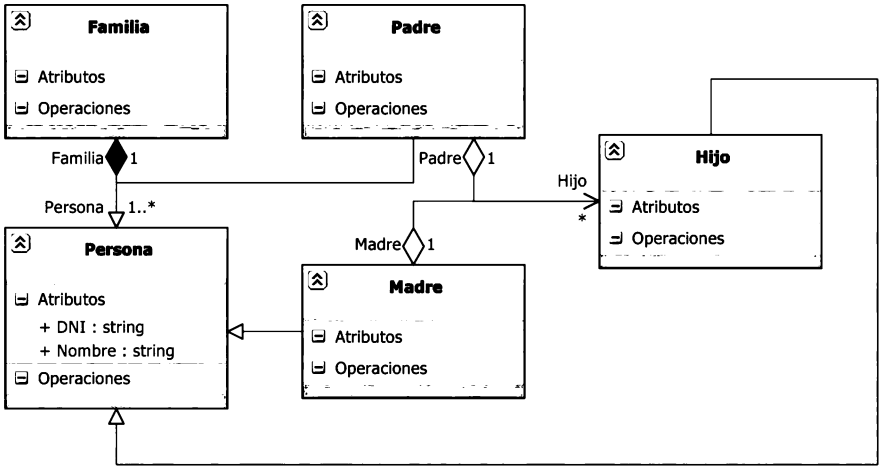


Figura 5.10. *Diagrama de clases final*

Como vemos, el funcionamiento y manejo de la herramienta es muy sencillo, y basta con arrastrar y soltar para tener un diagrama de clases muy sencillo creado en un momento.

Basta con hacer doble clic sobre los diferentes elementos (cardinalidad, rótulos, atributos…) para editarlos, y con un par de movimientos de ratón podemos moverlos a nuestro antojo sin que las relaciones se emborronen o distorsionen.

Generar código a partir de diagramas de clases

Ya hemos visto cómo utilizar la herramienta de modelado para crear proyectos de modelado y así diseñar nuestra aplicación. También hemos hecho notar al principio de este capítulo que los diagramas de clases tienen una relación directa con el código, es decir, se puede crear una traducción exacta de un código a un diagrama de clases y de un diagrama de clases a un código. Esta funcionalidad nos la ofrece la herramienta de modelado, para ello crearemos un nuevo proyecto de aplicación de consola que llamaremos *ClasesyCodigo*. Nos generará la plantilla básica con una clase program y su método *main*. Esa clase no la vamos a tocar, en su lugar vamos a ir añadiendo las clases de nuestro programa, tal y como haríamos normalmente, pero en vez de crear las clases según el método tradicional lo vamos a hacer mediante un diagrama de clases, por ello, una vez creado el proyecto, haremos clic con el botón derecho en él y elegiremos la opción **Agregar > Nuevo elemento**. Elegiremos la plantilla diagrama de clases y nos aparecerá la misma vista de dibujo que hemos visto hasta ahora.

Vamos a modelar una aplicación muy sencilla, donde tendremos solamente dos clases: una clase Operario y una clase Trabajos, donde representamos los trabajos que puede desempeñar un operario. Primero vamos a crear nuestra clase Operario, para ello iremos al cuadro de herramientas del diagrama y arrastraremos el elemento "Clase" hasta el cuadro de dibujo.

Nos aparecerá un cuadro de diálogo donde rellenar los datos necesarios para la clase, en este caso solo tendremos que añadir el nombre de la clase, darle a **Aceptar** y ya tendríamos nuestra clase tanto en el diagrama como en el proyecto. Si abrimos la clase que nos acaba de crear automáticamente, veremos que es una plantilla que solo contiene un constructor vacío, pero nuestro operario sin duda tiene unos atributos. Volvamos al diagrama de clases para añadirle los atributos necesarios, si hacemos clic con el botón derecho en la clase del diagrama, y nos vamos a **Agregar**, veremos que nos aparecen una serie de opciones para agregar a a la clase, un atributo es un campo, por lo que elegiremos la opción **campo**, nos saldrá entonces una entrada en la clase llamada "field" donde podremos escribir el nombre, escribiremos *nombre* y presionaremos **Intro**.

Figura 5.11. *Ventana de creación de clases*

Si ahora nos vamos a la clase *Operario*, veremos que ha creado un atributo privado llamado *Nombre* y vemos también que ese campo es de tipo *int* y eso no es correcto, bien podríamos arreglarlo desde el archivo de la clase o desde el diagrama, vamos a hacerlo desde el diagrama. Si seleccionamos el campo en nuestro diagrama, vemos que en la ventana de propiedades nos aparecen todas las características del campo. Entre ellas se encuentra la propiedad tipo, que podemos modificar a nuestro antojo, en este caso escribiremos *string*, podemos ver como al cambiar el tipo del campo en las propiedades también se ha cambiado en el diagrama. Nuestro código se encuentra ahora completamente enlazado con el diagrama, cualquier cambio en el diagrama modifica el código y cualquier cambio en el código modifica el diagrama. Añadiremos de este modo el atributo dni, que será un entero, por lo que no tendremos que cambiar el tipo. En el campo dni añadiremos también un comentario, escribiremos en la propiedad *Comentario* lo siguiente: "Sólo los números, no se tiene en cuenta la letra", y veremos que dicho comentario se ha convertido en un comentario justo encima del atributo dni. Haremos lo propio con nuestra clase Trabajo con los atributos nombre y descripción, los dos *strings*.

Una vez con nuestras dos clases y antes de relacionarlas, vamos a añadir un método a la clase *Operario* que se llame *TieneElTrabajo*, donde recibirá un trabajo como parámetro y devolverá un booleano como respuesta indicando si tiene o no tiene el trabajo. Al igual que con los atributos, podemos ponerle el nombre y su tipo de devolución desde la ventana de propiedades, pero no podemos añadirle los parámetros. Para añadirle los parámetros vamos a hacer uso de una ventana extremadamente útil en la herramienta de modelado, haremos clic con el botón derecho en el método y elegiremos la opción **Detalles de clase**, veremos una ventana con toda la información de la clase de una manera muy compacta y útil, además de permitirnos añadir o modificar cualquier información sobre nuestros elementos, utilizaremos esta ventana de ahora en adelante para realizar todas estas operaciones. Si expandimos los detalles del atributo haciendo clic en la flecha del método, podremos añadir los parámetros en la caja de texto rotulada como *<agregar parámetro>*, llamaremos al parámetro *trabajo* y elegiremos como tipo nuestra clase *Trabajo*.

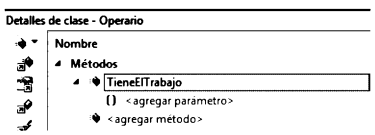


Figura 5.12. *Ventana Detalles de clase*

Si nos fijamos en nuestra clase *Operario*, veremos que nos ha creado un método vacío que simplemente lanza una excepción indicando que no se ha implementado la creación automática del método. Bien, parece que lo tenemos todo listo, solo nos quedaría definir la relación entre operarios y trabajos.

ACTIVIDADES 5.3

- Antes de continuar, razone sobre la relación entre operarios y trabajos. ¿Un operario tendrá muchos trabajos para realizar o un trabajo tendrá muchos operarios que lo pueden realizar? Es decir, tenemos dos opciones, que sea el operario quien tenga una lista de trabajos, o el trabajo el que tenga una lista de operarios que pueden realizarlo. ¿Qué diferencias habría? ¿Hay alguna otra opción?

Vamos a definir la relación indicando que un operario puede realizar uno o varios trabajos, por lo que elegiremos la herramienta de asociación y la arrastraremos desde la clase *Operario* a la clase *Trabajo*. Nos aparecerá una propiedad relacionando las dos clases, por desgracia no podemos definir una cardinalidad, pero podemos hacer otra cosa. Desde la ventana *Detalles de clase* de la clase *Operario* vemos nuestra propiedad *Trabajo*, la vamos a renombrar a *Trabajos* y vamos a cambiarle el tipo (cuando modificamos un tipo en una relación, nos aparece el espacio de nombres al completo) definiendo que es una lista, por lo que cambiaremos el tipo de *Trabajo* a *List<Trabajo>*. Al hacerlo, vemos que la propiedad que representaba la relación ahora se encuentra en una sección del diagrama que se llama *Propiedades*; realmente, la relación existe y es la misma a efectos prácticos, pero no es eso lo que queremos ver, es menos claro, pero podemos arreglarlo. Haremos clic con el botón derecho en la propiedad y elegiremos la opción **Mostrar como relación de colecciones** y volverá a mostrarse automáticamente como la asociación que queremos representar.

Y ya tendríamos nuestra estructura de clases prácticamene terminada, solo nos queda un detalle más. Tenemos nuestros atributos como privados, como tiene que ser, pero no hemos realizado la encapsulación ni nada por el estilo, por lo que no serán accesibles, no pasa nada, es normal, tenemos la estructura, pero no hemos implementado el código para hacerlo funcional, pero tenemos a nuestra disposición una funcionalidad adicional que hace a esta herramienta más útil de lo que aparenta. Podemos utilizar todas las refactorizaciones automáticas de Visual Studio que vimos en el tema anterior directamente desde el diagrama, por lo que si hacemos clic con el botón derecho en un campo y nos vamos al menú **Refactorizar**, vemos que tenemos la opción de encapsular el campo directamente, haremos clic en esa opción y nos aparecerá una ventana para que escribamos el nombre de la propiedad, escogemos la que más nos interesa (por convenio se suele utilizar el mismo nombre que el atributo pero con letra capital) y aplicamos los cambios. Vemos como nuestro campo está perfectamente encapsulado como propiedad. De este modo ya tendríamos nuestro diagrama y nuestra estructura de clases perfectamente realizada, y todo mientras realizábamos el diagrama.

Las posibilidades de esta herramienta son infinitas, aunque sea algo más restrictiva que si simplemente creamos un proyecto de modelado, podemos crear la estructura de clases al mismo tiempo que realizamos el diagrama, lo cual sin duda nos ahorra una gran cantidad de tiempo.

Ingeniería inversa

Visual Studio nos permite realizar una ingeniería inversa de un proyecto con su herramienta de modelado, pero, ¿qué es la ingeniería inversa en el contexto de los diagramas de clase? La respuesta es bien sencilla: consiste en obtener de forma automática el diagrama de clases de un proyecto mediante su código.

El procedimiento de esta poderosa herramienta es muy sencillo, lo único que tenemos que hacer es seleccionar un proyecto de los que hemos realizado hasta ahora, donde hayamos utilizado en código relaciones de asociación o herencia. Una vez abierto, haremos clic con el botón derecho en el proyecto y elegiremos la opción **Ver diagrama de clases**, automáticamente, Visual Studio nos creará un diagrama con las clases utilizadas en el proyecto, aunque sin relacionar, nosotros solo tendremos que establecer correctamente las relaciones entre las clases para dejarlo correctamente.

Es también un dato interesante la posibilidad de ver la clase base de una clase, como la de un WebService o de un formulario WinForms, solo tenemos que hacer clic con el botón derecho en la clase de la que queremos sacar la derivada y elegir la opción **Mostrar clase base**, automáticamente, nos creará y relacionará la clase base en nuestro diagrama de clases.

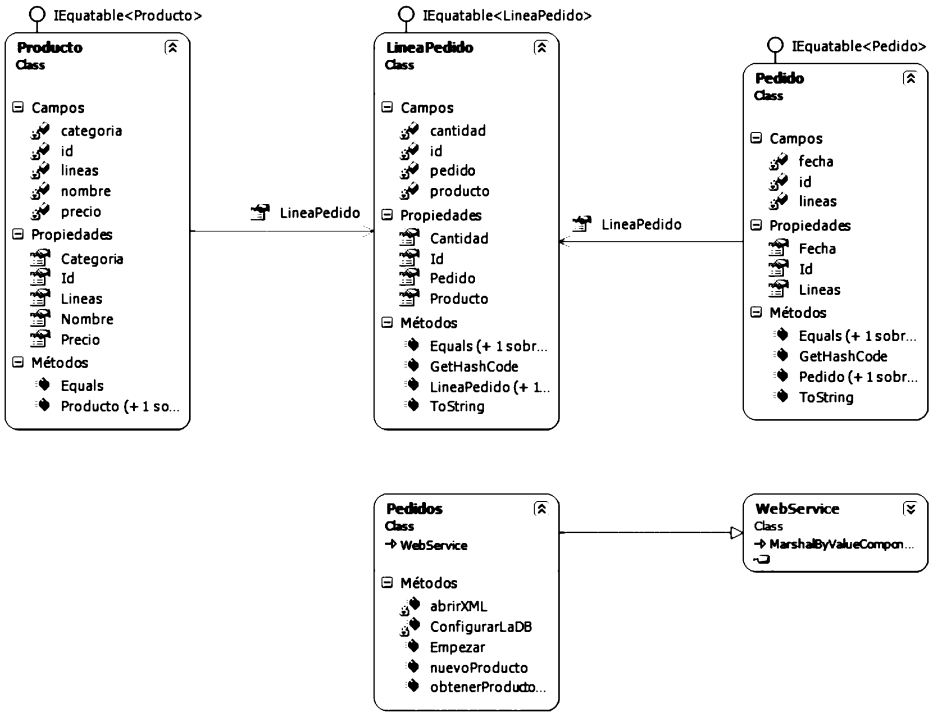


Figura 5.13. *Ingeniería inversa con Visual Studio*

5.3.2 UMLPAD

UMLPad es una herramienta muy sencilla y ligera que nos permite crear diagramas de clases. Es una herramienta externa y portable y su manejo es muy sencillo, muy similar al de la herramienta de modelado de Visual Studio.

El primer paso sería descargar el programa, el cual podemos encontrar en su web oficial:

http://web.tiscali.it/ggbhome/

Una vez descargado, no necesita instalación, es una carpeta comprimida que contiene el ejecutable del programa. Cuando lo ejecutamos, nos aparece una sencilla ventana dividida en dos. A la izquierda tenemos nuestro árbol de diagramas, que se actualizará según vayamos creando diferentes diagramas. En la parte derecha, tenemos nuestro cuadro de dibujo, donde incluiremos los elementos de nuestro diagrama. En la parte superior, tenemos además una barra de herramientas donde podemos elegir los elementos que vamos a añadir.

Para añadir una clase, bastaría con hacer clic en el elemento **Clase** y luego hacer clic donde queramos para que se posicione, y del mismo modo que hemos hecho hasta ahora, relacionar las clases seleccionando y arrastrando.

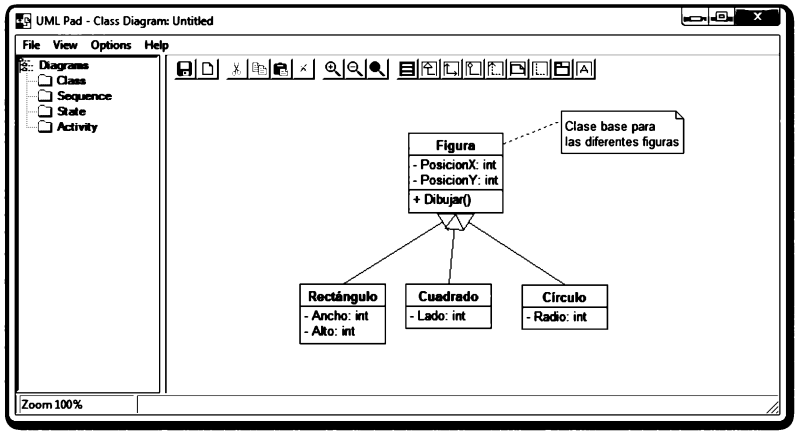


Figura 5.14. *UMLPad*

Sin embargo, vemos que no reacciona a nuestros dobles clics para editarlo, para editar un elemento y agregarle los valores, seleccionaremos un elemento (su borde se volverá de color rojo) y haremos clic con el botón derecho eligiendo la opción **Edit Object** en el menú contextual que aparece.

En la ventana que nos aparece, podemos darle nombre a nuestra clase, establecer si es virtual o si es una interfaz, y la lista desplegable **Constraint** nos permite establecer si dispone de algún tipo de restricción. En esa ventana, vemos además otras dos pestañas: **Attributes** y **Operations**. Desde esas pestañas añadiremos o eliminaremos los atributos y métodos a nuestras clases. El procedimiento es muy sencillo, primero escribimos el nombre y hacemos clic en el botón marcado con una V de color rojo, que nos añadirá el atributo a la lista y nos dejará rellenar el resto de datos al atributo. Podemos definirle el tipo de dato mediante una lista desplegable, su valor por defecto, sus restricciones y su nivel de acceso, entre otras opciones.

Para crear las operaciones, el método es muy similar, solo que en este caso podemos definir el valor de retorno y además tenemos una pestaña para añadir los parámetros que serán definidos del mismo modo que lo hicimos con los atributos.

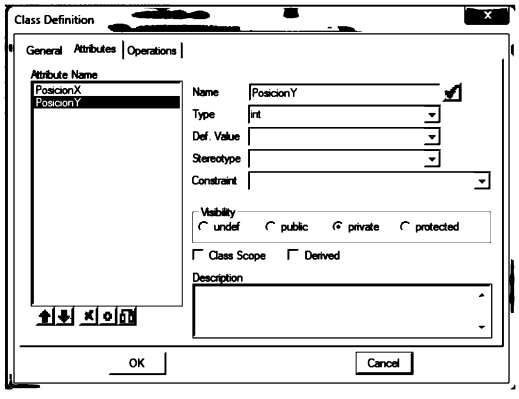


Figura 5.15. *Editar elemento con UMLPad*

Una vez que hayamos terminado de añadir los atributos y métodos, saldremos de esa ventana dando al botón **OK** y veremos los cambios que hemos realizado en el dibujo, podemos moverlo y relacionarlo sin problemas antes o después de realizar esa operación.

ACTIVIDADES 5.4

- Realice diagramas de clases de los proyectos anteriores en los que hemos trabajado sin utilizar ingeniería inversa con cualquiera de las herramientas propuestas.
- Realice ahora ingeniería inversa en dichos proyectos, compare los resultados, analice las diferencias que encuentre.
- Mediante la opción **Mostrar clase base** saque toda la jerarquía de clases de un formulario WinForms.

RESUMEN DEL CAPÍTULO

En este capítulo hemos visto los diagramas UML, hemos aprendido qué es un estándar unificado de reglas y notaciones para realizar diagramas y nos hemos centrado en los diagramas de clase, que son la piedra angular del diseño de software.

Mediante el uso de herramientas propias del entorno de desarrollo y externas, hemos realizado diagramas de una forma rápida y directa, aprendiendo con el uso de las aplicaciones la metodología que hay que usar cuando diseñamos una aplicación.

Los diagramas de clase son fundamentales y no podemos dejarlos de lado por su sencillez. Son una excelente guía cuando desarrollamos, tienen especial utilidad a la hora de mapear gráficamente la relación entre las diferentes entidades de nuestro proyecto y, mediante el diagrama, podemos darnos cuenta y ver más claro quién posee a quién corrigiendo nuestro primer análisis del sistema.

EJERCICIOS PROPUESTOS

1. Utilice la herramienta de modelado de Visual Studio para crear un diagrama de clases destinado a establecer un modelo de las entidades de una aplicación, dicha aplicación tendrá como objetivo realizar matrículas en una carrera universitaria. La Universidad está organizada habitualmente por facultades y son éstas las que tienen un abanico de carreras para escoger. Por otro lado, tenemos al alumno, que podrá realizar una matrícula en una carrera con un número determinado de asignaturas. Debemos tener en cuenta que una facultad tiene varias carreras, que una carrera tiene varias asignaturas y que la matrícula contiene información sobre la carrera y las asignaturas de las que se matricule.

2. Utilizando el diagrama resultante del ejercicio anterior, establezca un tipo a las asignaturas para que puedan ser consideradas como Troncales, Optativas y Obligatorias. ¿Cómo afecta ese cambio al diagrama de clases? Razone su respuesta.

3. Realice un diagrama de clases de un banco. El banco tendrá varios clientes y varias cuentas. A su vez, un cliente puede tener varias cuentas.

4. Utilizando el diagrama del ejemplo anterior, se pide que en el banco, además de cuentas corrientes, se puedan establecer planes de pensiones y planes de inversiones. Represente el diagrama resultante.

5. Utilizando el diagrama de clases de la Figura 5.13, desarrolle la aplicación que defina ese diagrama, después realice una ingeniería inversa y compare los resultados.

6. Descargue la extensión yUml e investigue sobre su funcionalidad. ¿Qué diferencias existen con el funcionamiento de la ingeniería inversa de la herramienta de modelado de Visual Studio?

TEST DE CONOCIMIENTOS

1 ¿Para qué se utilizan los diagramas de clase?

a) Representan el funcionamiento de un programa.
b) Establecen una relación entre la interfaz y las entidades.
c) Representan las relaciones entre las entidades de una aplicación sin prestar atención a las acciones.
d) Ninguna de las respuestas anteriores es correcta.

2 ¿Cómo podemos entender los diagramas UML utilizados por diferentes diseñadores?

a) Se utilizan notas aclaratorias en cada diagrama explicando la notación utilizada.
b) Los diagramas UML son un estándar entendible por todos.
c) Se utilizan herramientas que pueden interpretar diagramas si se establecen las reglas y notación del diagrama.
d) No se puede, los diagramas UML son únicos por cada proyecto y persona.

3 ¿Qué elementos del diagrama de clases representan el ciclo de vida de un programa?

a) Asociación.
b) Agregación.
c) Cualquiera de las respuestas anteriores.
d) Ninguna de las respuestas anteriores es correcta, el diagrama de clases no representa el ciclo de vida.

4 Se utiliza la ingeniería inversa para:

a) Obtener un diagrama de clases a partir del código.
b) Obtener código a partir de un diagrama de clases.
c) Obtener las reglas de notación de un diagrama.
d) Ninguna de las respuestas anteriores es correcta.

5 ¿Cuál de los siguientes elementos se utiliza para representar relaciones entre clases?

a) Herencia.
b) Composición.
c) Agregación.
d) Todas las respuestas anteriores son correctas.

6 La herencia múltiple no podemos representarla en los diagramas de clases, ¿por qué?

a) Hay lenguajes que solo permiten emular la herencia múltiple mediante interfaces.
b) La relación de herencia múltiple no existe en UML.
c) La herencia múltiple sí puede ser representada en UML.
d) Para evitar la redundancia del diagrama, se realiza mediante anotaciones en el propio diagrama.

6 Diseño orientado a objetos. Diagramas de comportamiento

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer los diferentes diagramas de comportamiento.
- ✓ Aprender a elegir y a usar los diferentes diagramas de comportamiento de una manera correcta y apropiada.
- ✓ Aprender el manejo de la herramienta de modelado de Visual Studio para cada diagrama.

6.1 TIPOS Y CAMPO DE APLICACIÓN

Existen diversos tipos de diagramas de comportamiento y, al igual que en el caso de los diagramas de clase del capítulo anterior, también son diagramas basados en el estándar unificado UML. Los diagramas de comportamiento son un modo de representar gráficamente los procesos y formas de uso de un programa, con ellos visualizamos los aspectos dinámicos de un sistema, como el flujo de mensajes a lo largo del tiempo, el movimiento físico de los componentes en una red o los diferentes estados y operaciones que transcurren en el ciclo de vida de un programa.

Si en los diagramas de clases veíamos cómo se definían y especificaban las diferentes clases, sus operaciones y relaciones, los diagramas de comportamiento nos dirán cómo interactúan a lo largo del tiempo dichas clases y operaciones.

Al tratarse de un tipo totalmente diferente de diagramas, no es necesario que lleven una relación directa con el diagrama de clases de un sistema, es más, podría no aparecer ningún nombre del diagrama de clases en un diagrama de comportamiento. No son diagramas estructurales, son diagramas conceptuales de manejo, de flujo y de la secuencia de un programa, por lo que es muy posible que dependiendo de la profundidad y extensión de un diagrama no aparezcan algunas de las entidades definidas en el diagrama de clases.

Cabe destacar que todos los diagramas de comportamiento se utilizan de un modo jerárquico, es decir, en principio, sea el diagrama que sea, se realiza un diagrama sencillo con pocas "etapas", donde vemos el funcionamiento del sistema, separándolo en diferentes secciones que posteriormente tendrán un diagrama más detallado y específico. Dependiendo de las circunstancias que rodeen el desarrollo del diagrama, podría ser más práctico realizar un diagrama detallado de las pequeñas partes y luego unirlas en un diagrama más general, no obstante, esta práctica no es muy habitual, ya que se puede perder la coherencia del sistema al desarrollar las partes pequeñas sin que sean parte de un todo mientras se realizan.

Durante este capítulo iremos viendo los diferentes tipos de diagramas de comportamiento más relevantes, explicando cuál sería su correcto uso.

6.2 DIAGRAMAS DE ACTIVIDAD

Los diagramas de actividad son los diagramas de comportamiento más sencillos y fáciles de comprender. Representan los flujos de trabajo del sistema desde su inicio hasta el fin con las operaciones y componentes del sistema.

Este tipo de diagramas tienen un gran parecido con los clásicos diagramas de flujo que seguramente hayáis visto con anterioridad y con una notación muy similar. Los diagramas de actividad tienen unas características muy concretas y restrictivas, se componen de tres elementos: **estados**, **transiciones** y **nodos**.

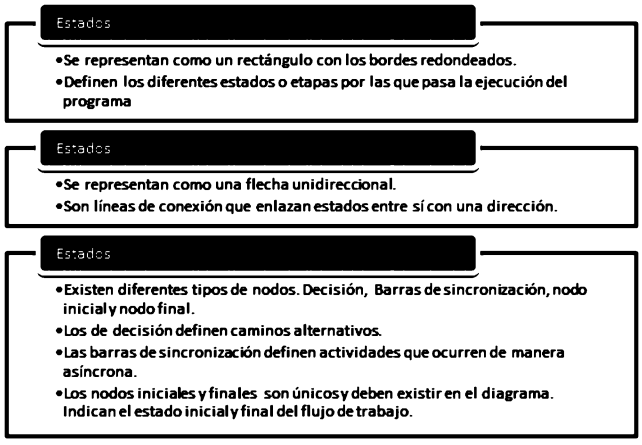


Figura 6.1. *Elementos de un diagrama de actividad*

Las reglas son muy sencillas, siempre debe haber un único estado inicial y un único estado final, todas las operaciones, transiciones y procesos ocurren entre esos dos puntos. Las transiciones se realizan entre estados y pueden tener nodos de por medio. Para aprender tanto su notación como su utilidad, vamos a mostrar un ejemplo sencillo en donde utilizamos todos los elementos de los que se puede componer nuestro diagrama de actividad.

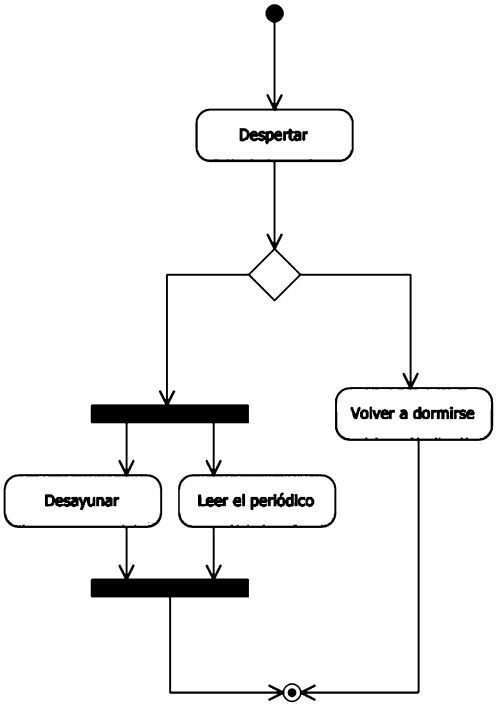


Figura 6.2. *Diagrama de actividad de despertarse*

Como vemos, en el ejemplo hemos utilizado un nodo inicial (representado por un punto), un nodo final (representado con un punto rodeado por una circunferencia), unas flechas de transición entre estados, un nodo de decisión (representado por un rombo) y dos barras de sincronización (representadas por una línea gruesa de color negro). Al igual que los nodos de bifurcación, las barras de sincronización pueden unir transiciones y separarlas, en este caso hemos usado los dos tipos de barras, ya que después de desayunar y leer el periódico debemos unir nuestro flujo de transición al realizarse de manera asíncrona.

Para modelar estos diagramas podemos utilizar la herramienta de modelado de Visual Studio, esta herramienta dispone de una plantilla específica para los diagramas de actividad con sus herramientas personalizadas. Para crear un diagrama de actividad, lo haremos como hicimos con el diagrama de clases, iremos a la entrada de menú **Arquitectura > Nuevo Diagrama** y elegiremos en este caso la plantilla **Diagrama de actividades UML**.

Veremos un cuadro de herramientas distinto al que conocemos donde se encuentran los elementos que vamos a utilizar en los diagramas de actividad.

Utilizaremos el elemento rotulado como **Initial Node** para poner el nodo inicial desde donde empezará nuestro flujo de trabajo, como vemos, tenemos también el nodo final rotulado como **Activity Final Node** que, como hemos visto, utilizaremos para definir el final de nuestra actividad.

El elemento **Action** dibuja nuestros estados, y luego tendríamos las bifurcaciones. Las bifurcaciones las tenemos de dos tipos, las que separan y las que unen, básicamente tendríamos un **Decision node**, que recibe una flecha de transición y pueden salir una o más de una, representando una decisión. Lo contrario ocurre con el **Merge Node**, donde recibe más de una transición y solo sale una transición de él.

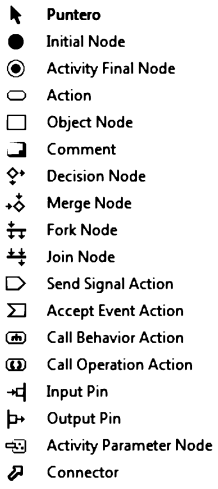


Figura 6.3. *Cuadro de herramientas de actividad*

Tendríamos un comportamiento análogo con las barras de sincronización, donde el **Fork Node** separaría transiciones y el **Join Node** las uniría.

Finalmente, tendríamos el elemento **Connector**, que, como su propio nombre indica, sirve de conexión entre los elementos. Representa las flechas de transición.

ACTIVIDADES 6.1

- Realice el diagrama de la Figura 6.2 por su cuenta mediante la herramienta de modelado de Visual Studio.
- Represente mediante diagramas de actividad el flujo de trabajo de proyectos anteriores.

6.3 DIAGRAMAS DE CASOS DE USO

Los diagramas de casos de uso representan cómo interactúan los diferentes actores en un sistema para cada caso de uso. Es decir, definen qué acciones puede realizar cada actor dentro de un sistema. Cada acción está representada de un modo muy simple por un rótulo que representa el caso de uso de la operación en cuestión.

Un modo de ver los casos de uso dentro de una aplicación serían los diferentes roles o permisos de los usuarios que tienen acceso a dicha aplicación, de este modo, dentro de un banco, los casos de uso no son los mismos (al menos no todos) si el usuario es el gerente o es un cajero.

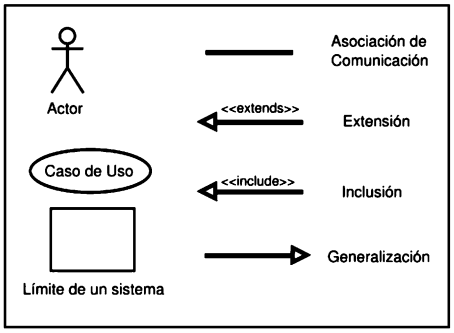


Figura 6.4. *Elementos de un diagrama de casos de uso*

Las relaciones que se pueden realizar entre casos de uso y los actores son realmente muy parecidas, son del mismo tipo pero restrictivas entre sí, es decir, una relación de extensión es una asociación de comunicación específica, al igual que todas las demás, por lo que podríamos generar un diagrama de casos de uso utilizando solamente la asociación de comunicación. En concreto, las relaciones de extensión e inclusión tienen una particularidad añadida: ese tipo de relaciones solo pueden darse entre casos de uso, es decir, es una interacción independiente del actor, que se produce por otro caso de uso.

- **Extensión**: esta relación implica que un caso de uso puede extender a otro, es decir, que el comportamiento del caso extendido se utiliza en otro caso de uso.
- **Inclusión**: similar a la extensión, esta relación implica que un caso de uso se incluye en otro, pero con una dirección determinada. Generalmente, este caso de uso suele provenir de un caso de uso de otro actor, mientras que el de extensión puede no venir de un actor concreto.

- **Generalización**: es un modo de representar la herencia de casos de uso, es decir, un caso de uso puede ser genérico, pero tener formas más específicas del mismo (como el ejemplo que vimos de Formas y las diferentes formas: cuadrado, rectángulo y círculo).
- **Límite de un sistema**: se utiliza para separar los diferentes sistemas dentro de un diagrama de casos de uso. En caso de que solo haya un sistema, no es necesario establecer el límite de un sistema.

Si utilizamos estos conceptos para modelar los casos de uso de una máquina expendedora de cafés, tendremos dos actores: el cliente que va a comprar a la máquina y la propia máquina, podemos especificar diversos casos de uso para cada uno de los actores que intervienen en el sistema. Sin duda, varios de los casos de uso estarán relacionados entre sí, sobre todo los que son propios de la máquina y los que son del cliente.

ACTIVIDADES 6.2

- Intente realizar por su cuenta el diagrama de casos de uso de la máquina expendedora de café con los casos de uso "Escoger producto" y "Entregar producto".

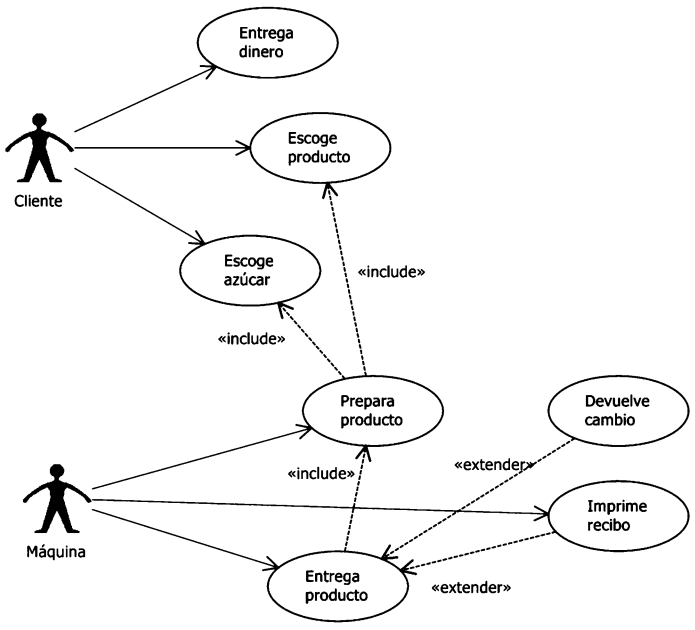


Figura 6.5. *Diagrama de casos de uso de la máquina expendedora de café*

En este diagrama podemos ver mejor las relaciones de extensión y de inclusión, que en la descripción podrían resultar un poco confusas. Devuelve cambio e imprime recibo extendienden a Entrega producto porque son casos de uso que se realizan cuando se entrega el producto, mientras que los casos de uso de escoger azúcar y escoger producto son parte del caso de uso de preparar producto.

Visto de otro modo, la máquina necesita que se haya escogido el producto y la cantidad de azúcar para preparar el producto, y la máquina devuelve el cambio e imprime el recibo cuando entrega el producto.

En la herramienta de modelado de Visual Studio podríamos crear este mismo diagrama utilizando los elementos de su cuadro de herramientas. Tal y como hemos hecho en anteriores ocasiones, crearíamos un nuevo diagrama de casos de uso utilizando la plantilla disponible en el IDE añadiendo sus elementos al cuadro de dibujo de la herramienta.

6.4 DIAGRAMAS DE SECUENCIA

Los diagramas de secuencia modelan la secuencia lógica a través del tiempo de los mensajes entre instancias. Se podría definir como la pila de llamadas resultante de realizar las diferentes operaciones, forman un mapeado de la traza de llamadas que se realizan cuando un *participante* realiza una acción.

El diagrama de secuencia se estructura mediante *líneas de vida*, que a su vez representan a un *participante* en el sistema, ya sea un actor o cualquier otro elemento que participe en el transcurso secuencial de nuestro programa. La parte más importante del diagrama es la línea de vida, esa línea representa una secuencia, una cronología que nos permite visualizar con un rápido vistazo las interacciones, mensajes y *participantes*, así como el número de ellos a través de toda la *vida* del programa, desde que se ejecuta hasta que se cierra.

Como ya hemos comentado, en el diagrama de secuencia se visualizan los diferentes mensajes que se realizan entre los objetos, éstos pueden ser de dos tipos:

- **Síncronos**: se corresponden con llamadas a métodos del objeto que recibe el mensaje. El objeto que envía el mensaje queda bloqueado hasta que termina la llamada. Este tipo de mensajes se representan con flechas con la cabeza llena.
- **Asíncronos**: estos mensajes terminan inmediatamente, y crean un nuevo hilo de ejecución dentro de la secuencia. Se representan con flechas con la cabeza abierta. Al igual que los mesajes síncronos, también se representa la respuesta con una flecha discontinua.

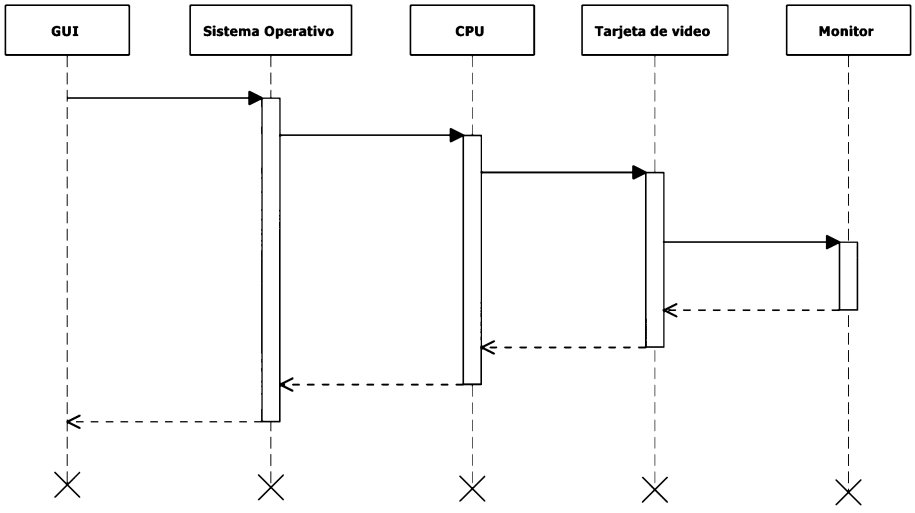


Figura 6.6. *Diagrama que representa las relaciones de la IGU*

Vemos en el ejemplo anterior la notación de las líneas de vida, los participantes y los hilos de ejecución. Cada participante debe tener su propia línea de vida y cada hilo de ejecución se representa como un rectángulo sombreado a lo largo de la línea de vida.

Este ejemplo es muy sencillo y básicamente hemos aprendido la notación y poco más, pero vayamos con algo más complejo y entendible conceptualmente, utilicemos el ejemplo de la máquina expendedora de los casos de uso.

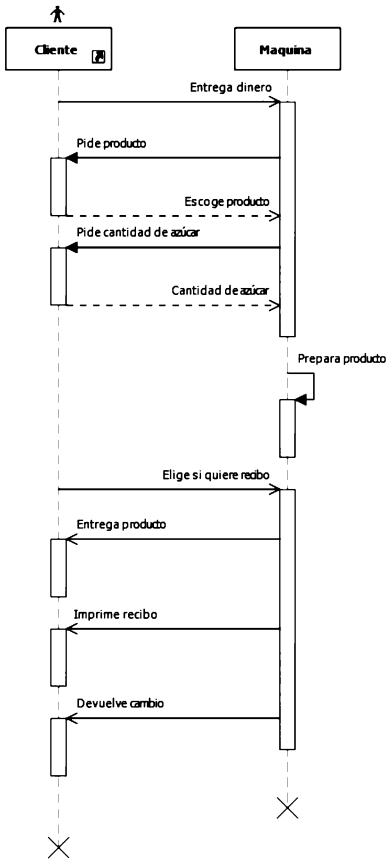


Figura 6.7. *Diagrama de secuencia de una máquina de café*

En este diagrama vemos que se han usado los mensajes asíncronos y los mensajes síncronos, básicamente observamos que las opciones que no requieren respuesta se han representado con llamadas asíncronas, aunque no se tiene por qué corresponder de ese modo siempre. Básicamente, la máquina tiene tres períodos de proceso: cuando le pide los datos al cliente, cuando procesa la solicitud y cuando devuelve los resultados (en este caso un café, el cambio y un recibo) al cliente.

Observamos que la máquina realiza una petición a sí misma y una vez finalizado el proceso de selección del producto, invoca una de sus acciones para realizar la solicitud del cliente.

También vemos que, a pesar de que la opción de imprimir el recibo es una opción, ésta no se encuentra representada, el diagrama de secuencia no representa opciones y esto es algo muy importante que hay que tener en cuenta. El diagrama de secuencia solo muestra la secuencia de los procedimientos que se realizan, ya sea entre instancias de distinto participante o sobre el mismo.

La utilización de la herramienta de modelado para diseñar estos diagramas es igual de sencilla que en el resto de diagramas, simplemente hay que escoger la herramienta del cuadro de herramientas y arrastrarla entre los elementos que queremos relacionar. En este caso concreto, hay una funcionalidad añadida que no se encuentra implícita en los elementos del cuadro de herramientas. Para juntar las diferentes llamadas en un mismo hilo de proceso, hay que arrastrar (seleccionándola previamente) la flecha que indica el mensaje hacia la posición donde se encuentra el hilo de proceso al que queremos añadir el mensaje.

ACTIVIDADES 6.3

- Amplíe el diagrama de secuencia de la máquina de café añadiendo un participante más, un operador. El operador se encargará de reponer las existencias de café y de azúcar, realizar pruebas de funcionamiento y retirar el dinero recaudado por la máquina.

6.4.1 INGENIERÍA INVERSA

Al igual que con los diagramas de clases, podemos conseguir el diagrama de secuencia de un método concreto de manera automática mediante técnicas de ingeniería inversa disponibles en la herramienta de modelado de Visual Studio, para ello, vamos a realizar una sencilla aplicación que añade productos a un carrito de la compra. Para realizar nuestra estructura de clases, vamos a utilizar el mismo método que vimos en el Apartado 5.3.1 y crearemos las clases desde la herramienta de modelado, tendremos dos clases, *Carrito* y *Producto*, primero crearemos nuestro proyecto (una aplicación de consola por ejemplo) y crearemos las clases desde el diagrama, donde los atributos de *Carrito* serían *numPedido* y los atributos de *Producto* serían *nombre* y *precio*.

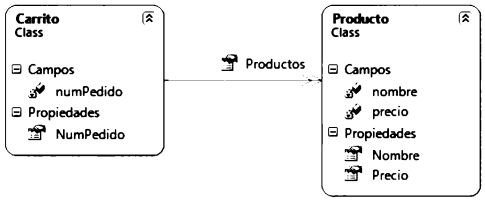


Figura 6.8. *Diagrama de clases*

Ahora añadiremos los métodos y funcionalidades que necesitamos para que sea funcional, añadiremos un contador en nuestro método *main* para llevar un contador de los carritos de compra que creemos y crearemos los métodos *CrearCarrito*, *CrearProducto* y *ListarProductos*.

No debéis olvidar tampoco que una vez creados y encapsulados los campos, y creada nuestra colección de asociación, deberemos eliminar el lanzamiento de la excepción *NotImplementedException* y deberemos inicializar la colección en el constructor de la clase *Carrito* para que la propiedad *Productos* sea una lista vacía y no una referencia nula.

EJEMPLO 6.1a

CREARCARRITO

```
private static Carrito CrearCarrito(int numPedido){
            Carrito carrito = new Carrito();
            carrito.NumPedido = numPedido;

            return carrito;
        }
```

EJEMPLO 6.1b

CREARPRODUCTO

```
private static Producto CrearProducto(string nombre,double precio)
        {
            Producto producto = new Producto();
            producto.Nombre = nombre;
            producto.Precio = precio;

            return producto;
        }
```

EJEMPLO 6.1c

LISTARPRODUCTOS

```
private static string ListarProductos(Carrito carrito)
{
string resultado = "Productos del carrito con N° de pedido: " +
       carrito.NumPedido +"\n";

      resultado += carrito.Productos.Select(p => p.Nombre + " Precio: " +
              p.Precio.ToString()).Aggregate((a,b)=> a+"\n"+b);

            return resultado;
 }
```

EJEMPLO 6.1d

MAIN

```
static void Main(string[] args)
        {
            int contadorPedidos = 1;

            Carrito carrito1 = CrearCarrito(contadorPedidos++);
            Carrito carrito2 = CrearCarrito(contadorPedidos++);

            Producto producto1 = CrearProducto("Leche", 2.0);
            Producto producto2 = CrearProducto("Pan", 0.6);
            Producto producto3 = CrearProducto("Huevos", 1.0);
            Producto producto4 = CrearProducto("Cereales", 1.25);

            carrito1.Productos.Add(producto1);
            carrito1.Productos.Add(producto4);

            carrito2.Productos.Add(producto2);

            System.Console.WriteLine(ListarProductos(carrito1));
            System.Console.WriteLine(ListarProductos(carrito2));
        }
```

Con nuestra aplicación funcionando correctamente, ha llegado el momento de obtener nuestro diagrama de secuencia. Podemos crear el diagrama de secuencia de cualquier método de nuestro código, para observar las diferencias en cada método, vamos a generar un diagrama por cada uno de los métodos implementados en nuestra clase principal excluyendo el método *main*. Lo único que tenemos que hacer es botón derecho en cualquier método y seleccionar la opción **Generar diagrama de secuencia**, una vez seleccionada, nos saldrá un cuadro de diálogo con las opciones de nuestro diagrama de secuencia, el cual dejaremos con los valores por defecto.

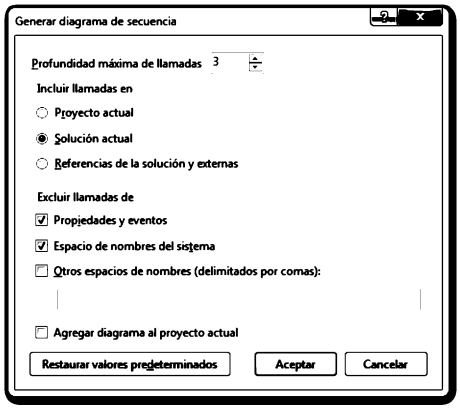


Figura 6.9. *Opciones del diagrama de secuencia*

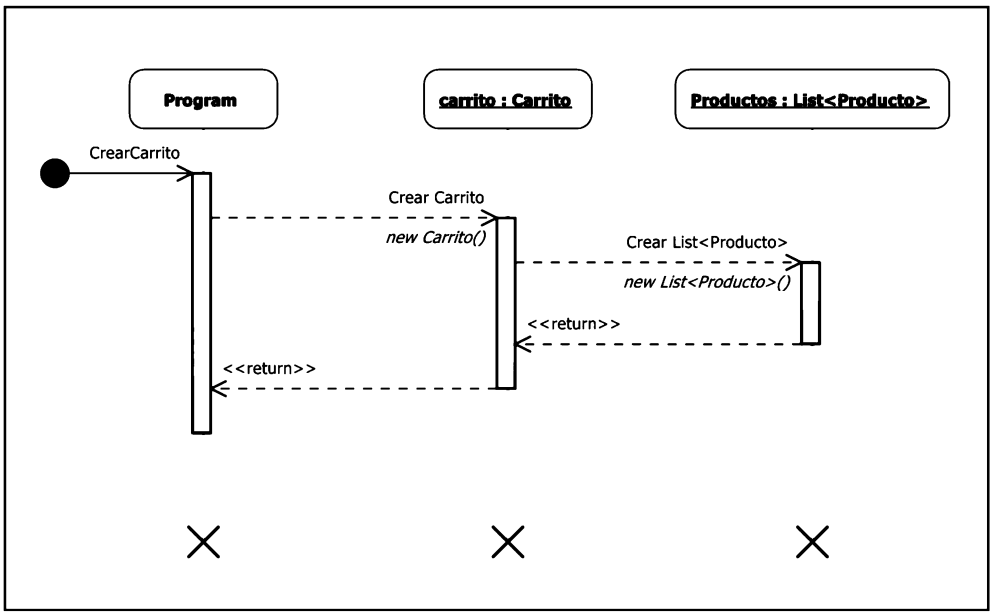


Figura 6.10. *Diagrama de secuencia del método CrearCarrito*

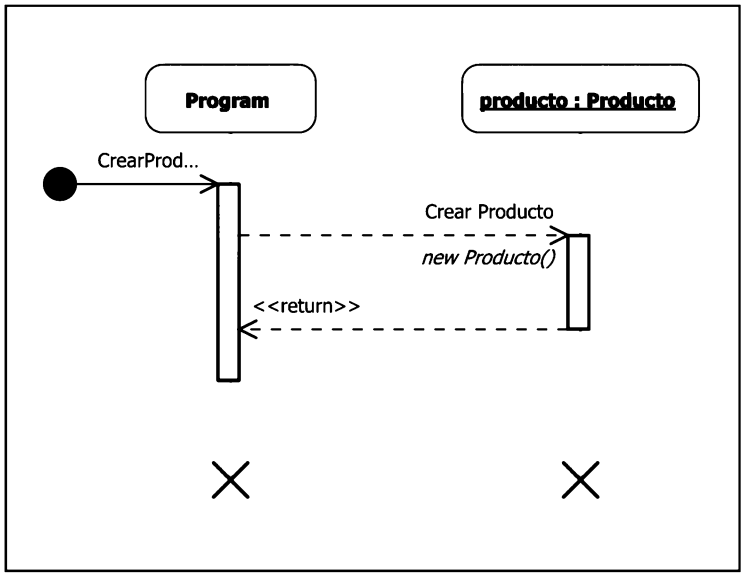


Figura 6.11. *Diagrama de secuencia del método CrearProducto*

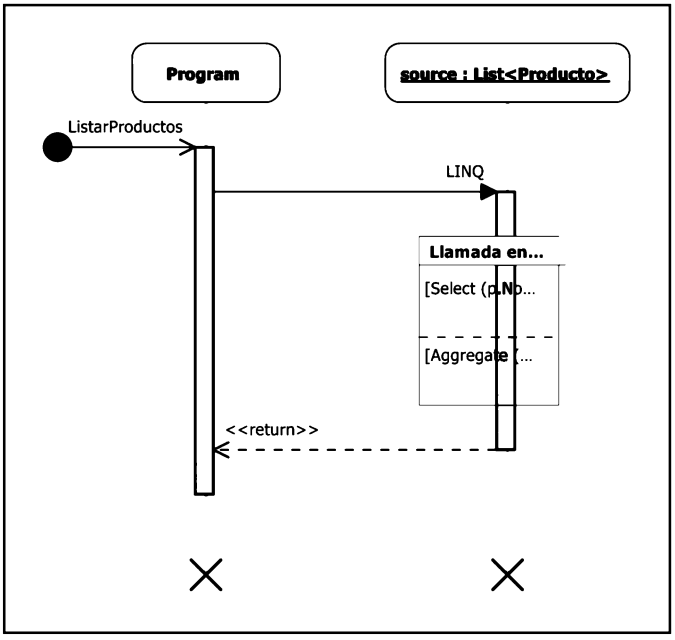


Figura 6.12. *Diagrama de secuencia del método ListarProductos*

Podemos observar en el último diagrama cómo se ha realizado la llamada a LINQ que hemos usado para sacar formateada la información de nuestra colección. LINQ es una poderosa herramienta para el manejo de colecciónes, hemos utilizado su potencial para proyectar sus elementos obteniendo una lista con los *strings* formateados y luego utilizando una concatenación de sus elementos, por ese motivo aparecen esas interacciones en la llamada en diferido que hemos realizado a la hora de crear nuestro listado de los productos.

Como vemos, el procedimiento es sumamente sencillo, mediante unos sencillos clics hemos obtenido el diagrama de secuencia de nuestro programa de forma automática, permitiendo analizarse de forma independiente para cada método de nuestro programa.

Con este sencillo programa no hemos visto lo que ocurre cuando hay llamadas a otros métodos dentro de nuestro proyecto, pero eso tiene fácil solución, vamos a añadir un método *ConvertirEnCadena* a nuestra clase *Producto*, ese método devolverá un *string* que tendrá un formato como el que hemos formado antes.

EJEMPLO 6.2a

PRODUCTO: CONVERTIRENCADENA

```
public string ConvertirEnCadena()
        {
            return nombre + " Precio: " + precio.ToString();
        }
```

Modificaremos también nuestro método *ListarProductos*, haciendo uso del método *ConvertirEnCadena* que acabamos de crear.

Como hemos visto, simplemente hemos sustituido en el método de proyección de LINQ donde antes convertíamos la información del producto en una cadena, para utilizar en su lugar el método de la clase *Producto* que se encarga de esta tarea que acabamos de crear.

EJEMPLO 6.2b

LISTARPRODUCTOS

```
private static string ListarProductos(Carrito carrito)
{
string resultado = "Productos del carrito con N° de pedido: " +
       carrito.NumPedido +"\n";

resultado += carrito.Productos.Select(p => p.ConvertirEnCadena())
.Aggregate((a,b)=> a+"\n"+b);

            return resultado;
 }
```

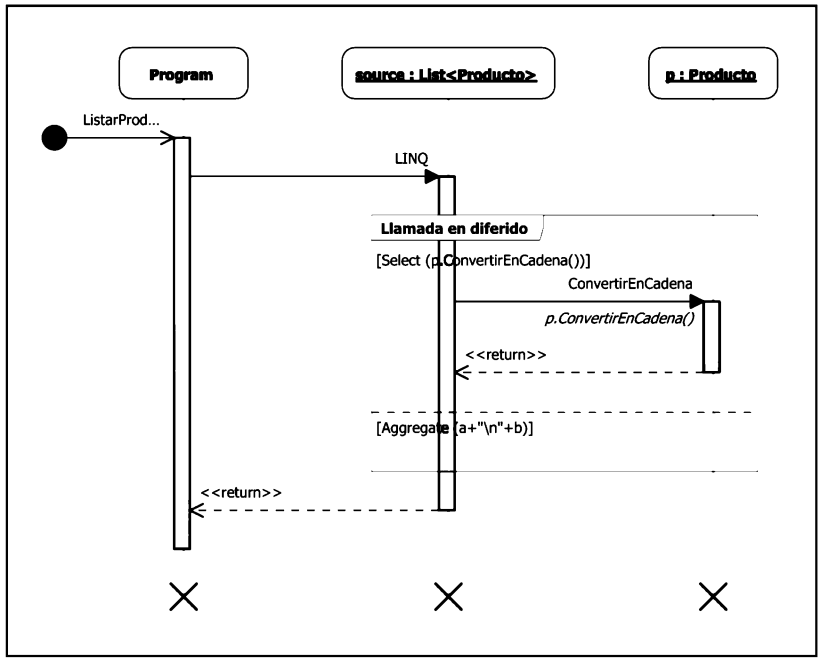


Figura 6.13. *Diagrama de secuencia del método ListarProductos*

En este nuevo diagrama, podemos ver como la llamada en diferido en donde se realiza el uso de la proyección del método Select llama a su vez al método convertir cadena de la clase *Producto*.

Con esto habríamos terminado con el uso de la generación automática de diagramas de secuencia, y hemos aprendido un nuevo concepto: las llamadas en diferido que se realizan al utilizar métodos y extensiones externas a nuestro propio código.

ACTIVIDADES 6.4

- Modifique el método *ListarProductos* evitando el uso de LINQ, utilice cualquier estructura de iteración para construir la cadena y vuelva a generar el diagrama. Reflexione sobre los resultados.
- Genere el diagrama de secuencia del método *main* de nuestra aplicación mediante la generación automática de Visual Studio.

RESUMEN DEL CAPÍTULO

En este capítulo hemos visto un tipo diferente de diagramas UML, los diagramas de comportamiento. Estos diagramas representan un proceso de ejecución del programa, ya sea mediante la especificación de los actores y los casos de uso, del flujo de trabajo de la aplicación o mediante una representación de la pila de llamadas entre los diferentes participantes del sistema.

Hemos seguido utilizando la herramienta de modelado de Visual Studio, que trae consigo las plantillas necesarias para modelar los diagramas correcta y apropiadamente.

Ahora somos capaces de diseñar una aplicación mediante los diagramas UML de forma completa, primero diseñando la estructura de clases mediante los diagramas vistos en el capítulo anterior y luego especificando cómo se va a trabajar con la aplicación, quiénes podrán realizar las diferentes acciones y cuál será la relación de mensajes y operaciones entre todos los participantes, internos y externos, de nuestra aplicación.

EJERCICIOS PROPUESTOS

1. Se desea diseñar el software necesario para una red bancaria provista de cajeros automáticos (ATM), que serán compartidos por un consorcio de bancos. Cada banco dispone de una serie de servidores, provistos de software propio, que llevan la información sobre sus cuentas y procesan las transacciones que actúan sobre dichas cuentas. Estos servidores están conectados a las estaciones de cajero, que son propiedad del banco y en las que operan cajeros humanos, que pueden crear cuentas e introducir transacciones sobre ellas. Los cajeros automáticos aceptan tarjetas de crédito, interaccionan con el usuario, se comunican con un ordenador central para llevar a cabo las transacciones, entregan dinero en efectivo al usuario e imprimen recibos. El sistema llevará el registro de las transacciones efectuadas, cumplirá características aceptables de seguridad y manejará accesos concurrentes a la misma cuenta. El coste de desarrollo de la parte compartida del sistema se dividirá entre los bancos que forman parte del consorcio en función del número de clientes provistos de tarjetas de crédito. Para poder realizar el diagrama de secuencia de esta aplicación, será necesario realizar dos diagramas de secuencia: el diagrama de secuencia relativo a *Validar tarjeta y clave* y el de *Retirar efectivo*.

2. Realice el diagrama de casos de uso para un gestor de incidencias que utilice los siguientes roles de usuario: Cliente, un usuario que podrá crear incidencias y ver el estado en el que se encuentran; Técnico, un usuario que realizará intervenciones para solucionar la incidencia, y Coordinador, que asignará incidencias a un técnico.

3. Utilizando el enunciado y el diagrama resultante del ejercicio anterior, realice un diagrama de actividad que represente el flujo de trabajo de una incidencia desde su creación hasta que se finaliza. Se definen además tres estados básicos para la incidencia: Abierta, En proceso y Cerrada.

4. El restaurante de comida rápida YMeVoyALaRuina quiere una aplicación que le permita gestionar los pedidos creados por sus clientes permitiéndoles llevar un seguimiento de su pedido y modificar los pedidos antes de que se repartan. Además, quiere llevar un seguimiento de los pedidos repartidos por sus empleados mediante informes que generará la propia aplicación. Realice el diagrama de casos de uso y de secuencia para el restaurante y un diagrama de actividad para sus pedidos.

TEST DE CONOCIMIENTOS

1 ¿Qué representa el flujo de trabajo de un programa?

a) El diagrama de actividad.
b) El diagrama de casos de uso.
c) El diagrama de secuencia.
d) Cualquier diagrama de comportamiento.

2 ¿Cuántos nodos iniciales podemos tener en un diagrama de actividad?

a) Uno.
b) Dos.
c) Tantos como se quiera.
d) Ninguna de las respuestas anteriores es correcta.

3 ¿Cómo representamos las decisiones en un diagrama de secuencia?

a) Las decisiones se usan en los diagramas de actividad.
b) Mediante un rombo en la línea de vida.
c) Ninguna de las respuestas anteriores es correcta.
d) Todas las respuestas anteriores son correctas.

4 La relación de inclusión en los casos de uso representa:

a) Una asociación de comunicación.
b) Una relación entre casos de uso.
c) Un caso de uso incluido en otro.
d) Todas las respuestas anteriores son correctas.

5 ¿Cuál de estas afirmaciones es verdadera?

a) Los diagramas de secuencia y de casos de uso se pueden transformar los unos en los otros mediante reglas de equivalencia.
b) Los diagramas de comportamiento representan la estructura interna del programa.
c) Solo puede haber un actor por cada subsistema en el diagrama de casos de uso.
d) El diagrama de secuencia representa una cronología de llamadas de un sistema.

7 ¡Ponlo en práctica!

OBJETIVOS DEL CAPÍTULO

- ✓ Preparar al alumno para diseñar y codificar proyectos.
- ✓ Realizar una evaluación global de los conocimientos del alumno y la capacidad del mismo para aplicar todo lo aprendido.
- ✓ Aprender el manejo de la herramienta de modelado de Visual Studio para cada diagrama.

7.1 NUESTRO PROYECTO

Para terminar con el propósito del libro, vamos a realizar un proyecto final que incluya todos los conceptos usables del contenido del libro. Para ello os guiaremos por una serie de procesos utilizando el modelo en cascada que vimos en el primer capítulo.

No llevaremos el proceso en cascada de un modo estricto, pues el objetivo es ofrecer una mayor libertad en la práctica que se va a llevar a cabo.

7.2 PLANTEAMIENTO

Primero debemos plantearnos qué es lo que queremos hacer, al final de este capítulo se os ofrece un enunciado para utilizarlo como proyecto en caso de que no queráis elegir otro proyecto.

En principio, cualquier proyecto es válido, siempre y cuando cumpla con unos requisitos de dificultad y lleve una funcionalidad coherente, presumiblemente se os asignará un tutor para guiaros y aconsejaros sobre el proyecto que queréis realizar. En cualquier caso, siempre podéis usar el enunciado propuesto como ejemplo para compararlo con el proyecto que hayáis elegido realizar.

Huelga decir que todas las actividades y tareas sobre las que se habla deberán estar contenidas y documentadas en el proyecto que vayáis a realizar.

7.2.1 DISEÑO CONCEPTUAL

Primero deberemos realizar un diseño conceptual, es decir, deberéis formaros una imagen mental de lo que debe hacer la aplicación, reflexionar sobre ello y ponerlo sobre el papel, ésta debería ser una tarea previa a cualquier otra, con ella, definiréis los requisitos y las operaciones básicas del programa, además de hacer añadidos y anotaciones que os serán útiles más adelante. En caso de que hayáis escogido como proyecto el enunciado propuesto, esta etapa será muy corta, reduciéndose a realizar una serie de anotaciones sobre el enunciado.

7.2.2 MODELADO COMPLETO

El proyecto deberá tener un modelado completo, es decir, deberéis realizar los diagramas de clase, actividad, casos de uso y de secuencia necesarios. Deberéis traducir el diseño conceptual en modo de diagramas como hemos aprendido en los capítulos previos.

El primer paso sería realizar el diagrama de clases, no tengáis prisa, primero definid las clases, sin atributos ni operaciones, luego realizad una pasada para rellenarlo con los atributos que creáis necesarios y luego una pasada con las operaciones de dichas clases.

Seguramente, cuando escribáis los métodos de las clases surjan atributos nuevos necesarios para la aplicación, no os preocupéis, es algo muy habitual, estamos diseñando y la tarea de diseño es en muchas ocasiones recurrente hasta llegar al modelado final.

7.3 ¿QUÉ TIPO DE PROYECTO ES?

Deberéis plantearos de qué tipo de proyecto estamos hablando y cómo podríamos atacar su desarrollo. Quizás el propio proyecto defina un único modo de realizarlo o quizás queramos utilizar unos recursos concretos que definan el tipo de proyecto.

7.3.1 TIPOS DE PROYECTO

Los tipos de proyecto que podríamos elegir son muy variados, hoy en día .NET ofrece un abanico de posibilidades de desarrollo abrumador, además de ofrecer excelentes plantillas para facilitar el uso de algunos patrones de diseño.

A pesar de ello, se explican de un modo general los diferentes tipos de proyecto y sus características.

Aplicación de consola

Nuestro proyecto será una aplicación de consola si no necesitamos o queremos que se haga uso de una interfaz de usuario destinada a la web o al escritorio. Por ejemplo, si nuestro proyecto se encarga de manejar el funcionamiento de un ascensor, donde la interfaz de usuario es solo física, definitivamente nuestro proyecto sería una aplicación de consola.

También sería posible que nuestro proyecto no esté destinado a equipos que posean una interfaz gráfica poderosa, como podrían ser equipos antiguos o servidores dedicados que solo necesitan poder interpretar comandos.

También sería factible que nuestro proyecto sea parte de un programa más grande, el cual necesite de una consola de comandos, como podría ser el propio sistema operativo o algún videojuego.

Aplicación de escritorio

Si nuestra aplicación está destinada a ser usada mediante la ejecución en un equipo, como por ejemplo un equipo doméstico, y necesita de una interfaz gráfica, estaríamos hablando de una aplicación de escritorio o de formularios.

La mayoría de las aplicaciones responden a estas características, por lo que lo más probable es que nuestra aplicación sea de escritorio. Hemos visto durante el transcurso del libro ese tipo de aplicaciones y sabemos cómo utilizarlas, por lo que no debería suponernos ningún problema realizar una aplicación de estas características.

Aplicación Web

Una aplicación web está destinada a ser desplegada y visible desde cualquier parte que tenga visibilidad sobre la aplicación. Bien podría ser publicada en Internet o en una intranet, en cuyo caso solo sería accesible desde la red local.

Podemos definir que nuestra aplicación necesita ser web si necesitamos que sea accesible desde cualquier parte y, además, se necesite que varios usuarios puedan acceder a la aplicación.

Servicio Web

Un servicio web es una aplicación web destinada a ser consumida por un cliente, a diferencia de la aplicación web, el servicio web no es accedido por un usuario final, es una aplicación cliente la que consume los servicios del servicio y realiza operaciones con él.

7.4 DOCUMENTACIÓN

Nuestro proyecto deberá estar debidamente documentado, deberíamos tener una documentación técnica destinada a la lectura de otros desarrolladores, partícipes o no en el proyecto, y una documentación con el propósito de ser mostrada al usuario final para que aprenda a manejar la aplicación.

7.5 OPCIONAL: INSTALACIÓN Y DISTRIBUCIÓN

De manera opcional, se podrá crear un instalador o un sistema de distribución del proyecto creado. Esta tarea no está especificada ni comentada en los contenidos del libro, por lo que, en caso de realizarse, se deberá realizar una documentación previa para llevarla a cabo.

Si se decide realizar esta tarea, se debe tener en cuenta que un usuario final debería ser capaz de realizarla sin necesidad de una ayuda externa y debería instalarse o desplegarse de manera autónoma sin problemas.

7.6 NOTAS

Para terminar con el capítulo y, por ende, con el libro, vamos a comentar un poco algunas cuestiones sobre el proyecto final.

Todas las tareas que hemos descrito para llevar el proyecto a buen término deberán ser realizadas con las herramientas que hemos visto en el libro.

Además, el proyecto deberá estar debidamente refactorizado y deberá usar al menos un patrón de diseño reconocible. También se deberán crear pruebas unitarias en el proyecto, por lo que al menos dos de las clases utilizadas deberán tener su *clase test* asociada.

Como comentario adicional, el proyecto deberá encontrarse en un repositorio del control de versiones, ya sea en un servidor local o en un servidor externo.

7.7 PROYECTO PROPUESTO

El supermercado SuperCart necesita informatizar su servicio de pedidos a domicilio. Necesita una aplicación que sea accesible tanto por los *clientes* como por el *administrador* para realizar todas las gestiones necesarias y mantener el sistema actualizado.

Este programa consta de *dos líneas de acción*, dependiendo del *tipo de usuario* que utilice el sistema.

- Se necesitará tener un listado de todos los clientes, pudiendo añadirlos, modificarlos y borrarlos. Dicho listado de clientes se guardará también en un fichero con el siguiente formato:

  id@dni@usuario@nombre y apellidos@dirección@contraseña@admin

EJEMPLO 7.1

1@72141790J@abc@Zanahoria Fundidodehierroson@Calle falsa 123@contra_segura@0

2@72561892A@nimda@Ook Ook Ook!@Universidad Invisible S-1@nopongonumeritos@0

3@13985722V@admin@Mustrum Ridiculus@Universidad Invisible 3ºB@4815162342@1

4@48151623A@bruja@Tata Ogg@Pais de caliza nº1337@elvelozmurcielagohindu@0

- Se necesitará tener un listado de todas las categorías de productos, pudiendo añadirlas, modificarlas y borrarlas. Dicho listado de categorías se guardará también en un fichero con el siguiente formato:

  id@nombre

EJEMPLO 7.2

1@Carnicería

2@Lácteos

3@Refrescos

4@Limpieza

- Se necesitará tener un listado de todos los productos disponibles, pudiendo añadirlos, modificarlos y borrarlos. Dicho listado de productos se guardará también en un fichero con el siguiente formato:

  id@nombre@id_categoría@precio

EJEMPLO 7.3

1@Pizza Carbonara@6@4,8

2@Yogures activia@2@15,16

3@Lejía conejo@4@23,42

4@Queso de burgos@2@31,41

- Se necesitará tener un listado de las relaciones entre pedidos y productos que se piden. Hay que tener en cuenta que un pedido puede constar de varios productos, por lo que tendremos otro tipo de objetos que utilizaremos de transición, será una relación donde se almacenará el *id* del pedido y el *id* del producto. Dicha relación de pedidos se guardará también en un fichero con el siguiente formato:

  id_pedido@id_producto@cantidad

EJEMPLO 7.4

1@2@1

1@6@16

2@2@4

3@1@1

- Se necesitará tener un listado de todos los pedidos, pudiendo añadirlos, modificarlos y borrarlos. Dicho listado de productos se guardará también en un fichero con el siguiente formato:

  id@id_cliente@fecha

EJEMPLO 7.5

1@2@15/09/2011

2@15@4/8/2011

3@2@1/1/1983

4@8@15/16/2342

Primeramente, el programa deberá solicitar un *nombre de usuario* y una *contraseña*, si la identificación es positiva, se deberán mostrar unas opciones diferentes, dependiendo de si se trata de un *cliente* (*campo admin=0*) o de un *administrador* (*campo admin=1*).

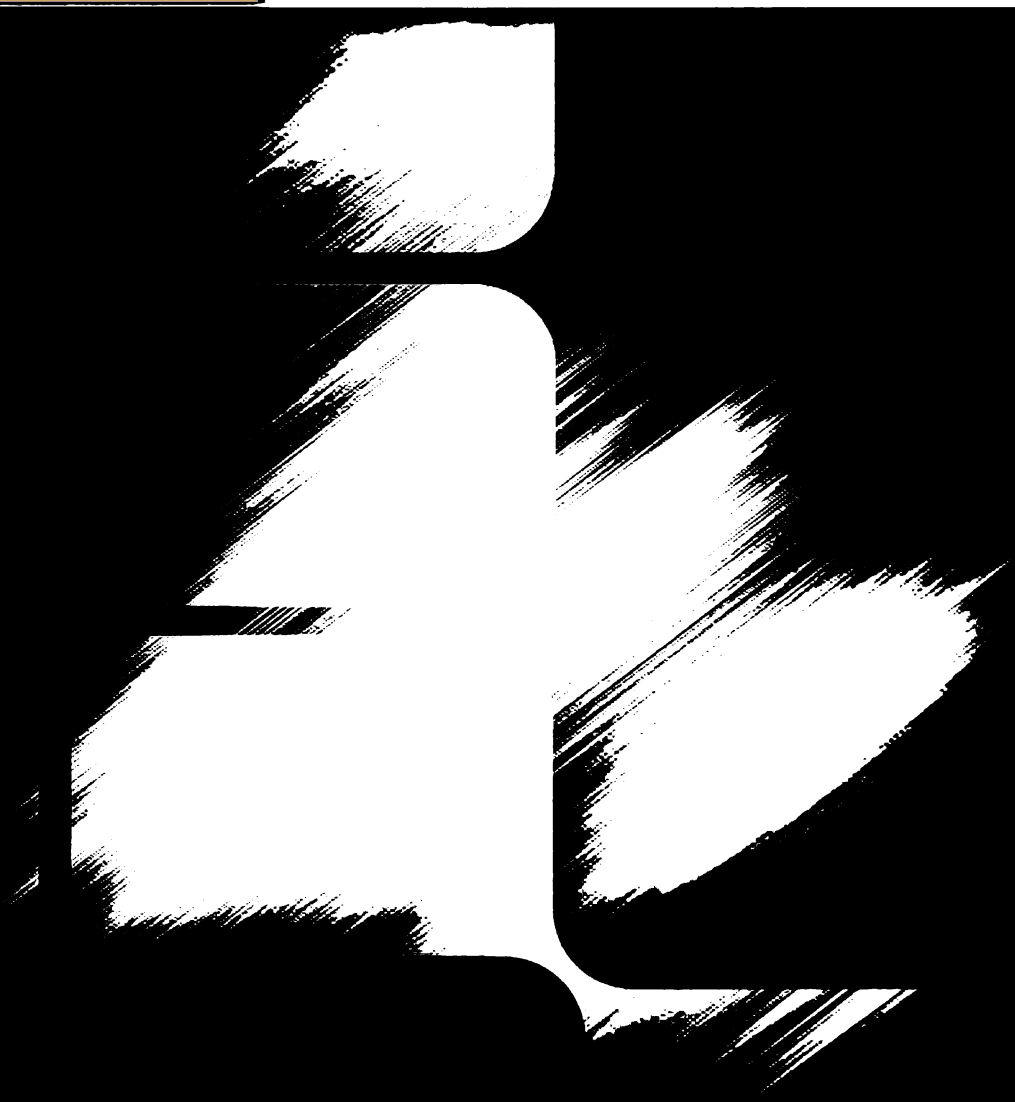
El programa deberá permitir las siguientes operaciones desde la vista del cliente:

- ✓ Modificar su dirección.
- ✓ Ver el listado de productos. *Organizados por categorías*.
- ✓ Añadir productos a un carro de la compra.
- ✓ Ordenar un pedido. *Finalizando el carro de la compra*.
- ✓ Ver su lista de pedidos ordenados.
- ✓ Desconectarse.

El programa deberá permitir las siguientes operaciones desde la vista del administrador:

- ✓ Crear, modificar y borrar usuarios.
- ✓ Crear, modificar y borrar categorías.
- ✓ Crear, modificar y borrar productos.
- ✓ Ver listado de pedidos.
- ✓ Desconectarse.

8 Comentarios y conclusiones



En primer lugar, espero que el contenido del libro sea de verdadera utilidad para todo aquel que lo utilice como manual de referencia o como ayuda al estudio académico.

En un tema tan genérico como son los entornos de desarrollo, es sumamente complicada la realización de un documento que sirva de utilidad para cualquier desarrollador o trabajador en la industria del software, debido a la cantidad ingente de diferentes lenguajes de programación y de los entornos de desarrollo específicos que existen para cada uno de ellos. Por si fuera poco, para cada lenguaje existen una amplia gama de entornos de desarrollo, y para cada entorno de desarrollo hay también una gran cantidad de herramientas y extensiones disponibles que realizan las mismas o semejantes tareas.

Se decidió por el entorno de desarrollo Visual Studio por ser uno de los entornos de desarrollo más completos y profesionales, además de contar con una excelente galería de extensiones listas para ser instaladas e integradas en el sistema.

Como habréis podido comprobar durante la lectura del libro, para cada tarea y contenido se han tenido que elegir unas herramientas específicas para desarrollar su manejo y uso de un modo concreto. A pesar de ello, las herramientas alternativas o con el mismo objetivo suelen tener un funcionamiento muy similar, por lo que una vez que sabemos manejar una herramienta deberíamos ser capaces (quizás con un poco de ayuda de la documentación de la herramienta) de manejar una herramienta con el mismo propósito sin problemas. Un caso claro de esto podría ser el cliente de control de versiones. En nuestro caso hemos usado AnkhSVN, pero el funcionamiento con cualquier otro cliente será muy parecido tanto en funcionalidad como en interfaz.

Este libro incluye además conceptos sobre la programación orientada a objetos que no se especifican ni se definen de un modo completo, ya que se da por sentado que el lector debería conocer esos conceptos de antemano, por ejemplo, se utilizan técnicas de polimorfismo y herencia en las refactorizaciones, pero no se explica qué es una herencia o qué es el polimorfismo ni cómo se deben implementar. Esto se debe a que no es el objetivo de este libro el aportar dichos conocimientos, sino cómo usarlos en nuestro beneficio para una determinada tarea, ya sea refactorizar nuestro código o realizar un diseño y un modelado de un software.

Todas las técnicas y herramientas contenidas en este libro están en constante desarrollo y mejora, por lo que siempre resultará interesante estar pendiente y documentarse sobre los nuevos cambios que se producen constantemente en el mundo de la industria del software. Cabe destacar la importancia de revisar las nuevas extensiones y añadidos que se realizan al estándar de modelado UML, pues diversas extensiones a un diagrama concreto pueden restarle importancia a otros diagramas o incluso resultar redundantes.

En el Capítulo 4, se mencionan de pasada dos herramientas adicionales de asistencia al programador y que ofrecen refactorizaciones potentes y completas adicionales a las disponibles en Visual Studio, quisiera remarcar con especial atención la herramienta ReSharper, la cual uso a título personal y profesional diariamente siendo de gran ayuda y una herramienta sumamente recomendable para cualquier programador.

Para concluir, quisiera remarcar que tengáis presente que no siempre vais a poder utilizar las herramientas a las que estáis acostumbrados, por lo que está bien agradecer la ayuda y asistencia que nos ofrecen, pero no se debe depender de ellas para programar, deberíamos ser capaces de realizar cualquier programa desde un editor de texto plano (como el bloc de notas); los entornos de desarrollo y sus herramientas no hacen un programa mejor, solo facilitan y aceleran el proceso de desarrollo de software.

Índice Alfabético

A

Actualizar ficheros locales a una versión específica, 123
Agregación, 137
Análisis, 19
Análisis de código, 68
Análisis de valores límite, 74
Analizador estático de código, 68
Antipatrones, 36
Aplicación de consola, 169
Aplicación de escritorio, 169
Aplicación web, 169
Arquitectura de software, 22
Asociación, 136

B

Barras de herramientas, 61

C

Cadenas mágicas, 37
Caja blanca, 72
Caja negra, 73
Características, 46
Casos de prueba, 71
Clases, atributos y métodos, 133
Clasificación y características, 14
Cliente de control de versiones: ANKH SVN, 121
Codificación, 20
Código spaguetti, 36
Coherencia, 77
Comentarios de documentación, 125
Compilación, 17
Comportamiento, 30
Composición, 136
Consolidar expresiones condicionales, 95
Consolidar fragmentos duplicados en condicionales, 94
Constructor virtual, 23
Contratos de código, 70
Control de versiones, 118
Copiar & pegar, 38
Creacionales, 23
Criterios de elección de un IDE, 48

D

Decorador, 27
Desarrollo colaborativo, 51
Desarrollo en 3 capas, 38
Descomponer un condicional, 95
Diagramas de actividad, 150
Diagramas de casos de uso, 153
Diagramas de secuencia, 155
Diseño, 19
Diseño conceptual, 168
Diseño de clases en UML, 133
Documentación, 20, 124, 170

E

Edición de programas y generación de ejecutables, 51
Eliminar asignaciones a parámetros, 93
El programa informatico, 12
Encapsular atributo, 99
Encapsular atributo como propiedad, 99
Encapsular colección, 99
Estado, 31
Estructurales, 26
Explotación, 21
Extensiones y herramientas, 46
Extraer clase, 100
Extraer método, 91
Extraer subclase, 100

F

Fachada, 29
Fases de compilación, 18

Fábrica abstracta, 23
Flujo de lava, 37
Forma de ejecución, 16

G

Generar código a partir de diagramas de clases, 141

H

Herencia, 138
Herramienta de modelado de VS, 139
Herramientas, 126, 139
Herramientas de control de versiones, 121
Herramientas de depuración, 66
Herramientas de Visual Studio, 103
Herramientas y disponibilidad, 49

I

Infierno de las dependencias, 37
Ingeniería inversa, 143, 157
Inspecciones, 67
Instalación, 54
Instancia única, 25
IntelliSense, 103
Interacción con el sistema, 12
Introducción a UML, 132
Iterador, 35

L

Lenguaje de programación y framework, 49
Lenguajes de alto nivel, 15
Lenguajes de bajo nivel, 15
Lenguajes de medio nivel, 15
Lenguajes de programación, 14

M

Malos olores, 101
Manejo de excepciones inútil, 37
Mantenimiento, 21
Martillo dorado/Varita mágica, 37
Metodología, 78
Modelado completo, 168
Modelo vista controlador, 39
Modelo vista vistamodelo, 39
Mover método, 93

N

Nivel de abstracción, 15
Notación, 135
Notas, 170
Nuestra elección Visual Studio, 53
Nuestro proyecto, 168
NUnit, 79

O

Objeto compuesto, 28
Obtención de código ejecutable, 17
Opcional: instalación y distribución, 170
Opciones del entorno, 61
Opciones del proyecto, 62
Otras herramientas, 117

P

Paradigma de programación, 16
Partición equivalente, 74
Patrones de desarrollo, 22
Patrones de refactorización más usuales, 91
Personalización y configuración, 48, 59
Planteamiento, 168
Procesos de desarrollo, 19
Proyecto propuesto, 171
Prueba de condiciones, 73
Prueba del camino básico, 72
Pruebas, 20
Pruebas de bucles, 73
Pruebas unitarias, 78
Puntos de ruptura, 66
Puntos de seguimiento, 67

Q

Qué tipo de proyecto es, 169

R

Recorrido por las ventanas y paletas principales, 55
Reemplazar array con objeto, 98
Reemplazar condicional por polimorfismo, 96
Reemplazar datos y valores por objetos, 98
Reemplazar número mágico con constante simbólica, 97
Reemplazar número mágico con método constante, 97
Reemplazar subclases por atributos, 100

Refactorizaciones, 104
Refactorización, 88
Refactorización y pruebas, 102
Reinventar la rueda, 37
Relaciones, 135
Rendimiento, 75
Repositorios, 118
Roles que interactúan en el desarrollo, 21

S

Separar variables temporales, 92
Servicio web, 170
Sistema operativo, 49

T

Tabulación, 90
Team Foundation Server, 121
Tipos de código (fuente, objeto y ejecutable), 17
Tipos de proyecto, 169
Tipos y campo de aplicación, 150

U

UMLPad, 144
Uso básico de un IDE, 51
Uso de comentarios, 124

V

Ventanas, 59
Ventanas: documentos, 59
Ventanas: herramientas, 60
Ventanas de depuración, 59
Visitor, 32
Visual SVN Server, 119

La presente obra está dirigida a los estudiantes de los Ciclos Formativos **Desarrollo de Aplicaciones Multiplataforma** y **Desarrollo de Aplicaciones Web** de Grado Superior, en concreto para el módulo profesional **Entornos de Desarrollo**.

Los contenidos incluidos en este libro abarcan los conceptos básicos de entornos de desarrollo. Se estudian como objetivo principal de la obra los entornos de desarrollo, aprendiendo todas las posibilidades que nos ofrecen. También se aprenderán a utilizar las herramientas disponibles para mejorar y optimizar el proceso de desarrollo de software. Se utiliza como base principal el IDE Visual Studio, uno de los entornos de desarrollo más avanzados y aclamados disponibles en el mercado. Para el correcto entendimiento de todo lo presentado en la obra, se aportan conocimientos de arquitectura de software, que será útil para todo desarrollador. En el transcurso de todo el libro se realiza un enfoque directo, dispuesto para garantizar un aprendizaje rápido y pragmático de todas las herramientas y opciones necesarias para el desarrollo de software, maximizando el uso de los entornos de desarrollo, sin dejar de aportar una base de conocimientos mejorando la calidad del aprendizaje ofrecido.

Los capítulos incluyen actividades y ejemplos con el propósito de facilitar la asimilación de los conocimientos tratados. Así mismo, se incorporan test de conocimientos y ejercicios propuestos con la finalidad de comprobar que los objetivos de cada capítulo se han asimilado correctamente.

Además, reúne los recursos necesarios para incrementar la didáctica del libro, tales como un glosario con los términos informáticos necesarios, bibliografía y documentos para ampliación de los conocimientos.

WWW

En la página web de **Ra-Ma** (**www.ra-ma.es**) se encuentra disponible el material de apoyo y complementario.

9 788499 641690

ra-ma.es

Ra-Ma®