1 INTRODUCCION

Este capítulo es parte de un libro aún preparación. Por esta razón los autores no se hacen responsables de los errores y gazapos que en él puedan encontrarse. Por cortesía han sido puestos a disposición de los estudiantes de la carrera de Ciencia de la Computación. Queda prohibida la comercialización, utilización en otros textos y transferencia a terceros sin el consentimiento expreso de los autores. Los autores agradecen cualquier sugerencia o error que les sea comunicado.

En el prefacio se dice que uno de los objetivos de este libro es que quien intente aprender a programar debiera hacerlo a través de la orientación a objetos y que es una acertada decisión hacerlo con C#, pero *¿qué es programar*?.

Dicho de modo breve y simplón: *programar* es desarrollar un *conjunto de instrucciones* "*entendibles"* por una *computadora* de modo que al ser *procesadas* por ésta nos *resuelvan un problema*.

Y aquí los problemas y las computadoras se dan la mano. Los tipos de problemas que nos planteamos a solucionar con computadoras han dependido de las capacidades de las computadoras, pero a su vez el interés en solucionar nuevos problemas ha incentivado el desarrollo de las capacidades de las computadoras.

7.1 Un Poco De Historia

Tal vez la más aceptada como *primera computadora* (aunque como ya se aclaró arriba esto no pretende ser la punta de la madeja) sea la conocida como *máquina de Babbage*[1] mediados del siglo penúltimo del pasado milenio. Más exactamente Babbage concibió varias máquinas conocidas como *Calculation Engines* las más ambiciosa de ellas fue la última conocida como *Analytical Engine* que nunca se terminó del todo. Esta intentó ser una máquina flexible y poderosa, con un mecanismo de control en tarjetas perforadas (parecido al que se utilizaba en las industrias de hilado de la época), un almacenamiento separado, un conjunto de registros internos y otras características que más tarde reaparecieron en las computadoras modernas de programa almacenado.

Había que dar las instrucciones u operaciones que fueran ejecutadas en las máquinas de Babbage. Una estudiante de matemáticas Ada Lovelace, ayudó a Babbage a "escribir" y probar esas instrucciones, y por ello a menudo se le considera *el primer programador* aunque esto es discutible[2].

Pero las instrucciones o *programa* de los ingenios de Babbage quedaban de cierta manera fijas en la concepción del "aparataje" (*hardware*) de la tal máquina. Es decir, no se podían introducir modificaciones sin de algún modo tener que hacer cambios en el aparataje, lo que limitaba la versatilidad de una tal la máquina[3].

---

[1] Charles Babbage (Inglaterra 1791-1871), http://www.ex.ac.uk/BABBAGE/biograph.html

[2] Augusta Ada Lovelace (Inglaterra 1815 – 1852) http://www.ex.ac.uk/BABBAGE/ada.html

[3] Y tal vez por ello la programación no fue aún la importante fuente de empleo que es hoy ✍.

Tuvo que pasar medio siglo para que no fuera sino hasta finales de los años 40 del siglo XX[4] en que pudiéramos hablar de máquinas de propósito más general, es decir máquinas que fuesen capaces de entender y *ejecutar* (procesar, realizar, llevar a cabo) un conjunto fijo de instrucciones (para ello disponían de un *procesador*) pero que suministrando distintas secuencias de ese conjunto de instrucciones (se le llama *programa* a una de estas secuencias) una misma máquina podría servir para la solución de diferentes problemas. Para indicarle a la máquina cuál es la combinación específica de instrucciones que debe ejecutar ésta dispone de una *memoria*, de modo que cambiando la secuencia de instrucciones que se pone en la memoria se pueda cambiar el efecto de la acción realizada por la máquina y solucionar con ello diferentes tipos de problemas. Esta memoria sirve a su vez para almacenar los datos que se procesarán y los datos intermedios producto de la ejecución de las instrucciones de la secuencia o programa, y que serán necesarios para la ejecución de otras instrucciones de la propia secuencia.

Esta arquitectura basada en procesador o "ejecutor" de una secuencia de instrucciones y memoria o "almacenador" de las propias instrucciones y de los datos, es la que ha caracterizado esencialmente a las *computadoras* hasta hoy día. Este concepto de *programa almacenado* en la memoria se mantiene hasta hoy y fue el punto de partida al trabajo de desarrollar software y a la forma en que se solucionan problemas con las computadoras.

Los primeros problemas que dieron lugar a estas primeras computadoras o los primeros problemas que pudieron ser resueltos con estas primeras computadoras (el perro que se muerde la cola) eran de naturaleza eminentemente numérica, es decir realizar cálculos numéricos aprovechando con ello la "velocidad" de procesamiento de esas computadoras (que permitía dar respuesta en un tiempo aceptable) y su "confiabilidad" (que incluye la disposición a repetir con menos fallos, y sin protestar, el cálculo monótono o preciso que el humano no es capaz de realizar)[5]. Lo que ha ocurrido desde aquellas primeras computadoras hasta esa "gran computadora" que es hoy Internet, no ha sido más que un proceso en espiral de *nuevos problemas a resolver* entonces *nuevas computadoras con nuevas capacidades a crear* y de *se crean nuevas computadoras con nuevas capacidades* entonces nuevos *problemas a resolver*.

El concepto de qué es un problema a resolver con computadoras, es relativo. Los físicos, matemáticos e ingenieros que programaban y usaban esas computadoras de los años 40 y 50 no se planteaban como problema a resolver el que se pudiese escribir un texto con una computadora en lugar de con una máquina de escribir de teclas y cinta que se gastaba y enredaba. Allá los escritores que emborronasen cuartillas y gastasen papel, o que renunciasen a enmendar sus erratas y a mejorar su estilo. Para los que, unos 20 años después, desarrollaron las computadoras y el software que permitía escribir documentos (y también modificarlos, adaptarlos y enmendarlos con facilidad), no constituía un problema que un periódico editado en Londres pudiese ser leído al instante, con imágenes a todo color y sin mediación de papel, en casi todos los confines del planeta, algo que hoy (unos otros 20 años después) disfrutamos sin asombro. ¿Cuáles problemas cree el lector podremos solucionar con las capacidades de las computadoras y del software de los próximos 20 años o qué capacidades se tendrán que desarrollar para resolver qué nuevos tipos de

[4] Aquí algunos países y algunos historiadores discrepan de quien merece la autoría de determinados aportes, de modo que como este libro no tiene espacio para abundar en ello se ha preferido no ofrecer nombres. El lector más interesado debe remitirse a la literatura sobre estos temas.

[5] Hay que reconocer que lamentablemente muchos de los problemas que dieron lugar o apoyaron la creación de estas "primeras" computadoras eran de carácter militar (como el de calcular la trayectoria de misiles). Estamos hablando de los años de la segunda guerra mundial y de la posterior guerra fría.

problemas?. Pronostique un nuevo problema que requiera de una nueva capacidad o una nueva capacidad que de lugar a un nuevo problema.

7.2 Conceptos Básicos

Sobre la arquitectura básica de *procesador* + *memoria* se han ido desarrollando equipamientos adicionales para mejorar y facilitar las vías y formas en que se suministran las instrucciones y los datos a procesar así como las formas en que se expresan los resultados[6], además de cómo aumentar las capacidades de almacenamiento de la información. Este equipamiento en el que hoy podemos mencionar a discos (de variado tipo), monitores, teclados, ratones, reproductores de sonido, impresoras y los más variados, sofisticados y especializados dispositivos, ha sido condicionado por (o ha condicionado a) los tipos de problemas o necesidades a resolver con las computadoras en un proceso en espiral de desarrollo.

Al conjunto de todos estos dispositivos, por ser "tangibles" y "físicos", es a lo que se le denomina *hardware*[7]. A las combinaciones de de instrucciones que se forman para que al ser ejecutadas por ese hardware nos resuelvan los problemas, y que no son tan "tangibles", se le denomina *software*[8].

De modo que al conjunto de hojas de papel encuadernadas, que tienen símbolos impresos en tinta sobre ellas, y que llamamos libro, puede considerársele "hardware". Pero a la novela que puede leerse en ese libro puede considerársele "software".

Sobre esta arquitectura de *procesador* + *memoria* + *dispositivos* se puede ejecutar un conjunto de acciones básicas que se denomina *lenguaje del procesador*[9]. Y a todo esto se le suele agrupar bajo la sombrilla del término *computadora*[10].

Desarrollar software es todo un proceso que de alguna forma culmina en una secuencia de instrucciones en uno de estos lenguajes que pueden ser procesados por la computadora.

En todos estos años han habido avances inmensos en cuanto a la velocidad de procesamiento, confiabilidad de ese procesamiento, reducción del consumo de energía y de la disipación de calor, reducción del tamaño de los dispositivos y reducción de los costos de fabricación, sin embargo, para sustentar esos avances, el repertorio de instrucciones o lenguaje de máquina que las los procesadores entienden y ejecutan de manera directa con su hardware se ha mantenido relativamente simple y no se ha desarrollado al mismo ritmo. Por simple, estos repertorios son *fáciles* de aprender; pero fácil de aprender no quiere decir *fácil* para solucionar los problemas más *difíciles* que cada vez mas nos planteamos.

---

[6] Inicialmente fueron denominados *periféricos* porque se ubicaban en la "periferia" o fuera del aparato principal al que se le llamaba computadora. Este término es mantenido aún hoy por muchos aunque podamos tener "encerrados" una misma "caja" monitor, bocinas de sonido, teclado, memoria, unidades de disco, procesador.

[7] En español podría decirse "*aparatura*", "*aparataje*" pero ya está acuñado el término en inglés *hardware*..

[8] El término *software* del inglés *soft* (blando) se ha adoptado en contraposición del término *hard* (duro). No hay un equivalente aceptado por consenso en español.

[9] También se dice en lenguaje coloquial *lenguaje de máquina*.

[10] También llamado *computador* u *ordenador* en otros países de habla hispana pero nosotros preferimos usar el femenino.

Para salvar este escollo se han desarrollado, desde las primeras computadoras, *lenguajes* o *repertorios de instrucciones de mas alto nivel* que si bien pueden ser menos simples de aprender, ni son reconocidos y ejecutados de manera directa por un procesador, son más expresivos y por tanto nos ayudan a resolver de manera *más fácil* los problemas *más difíciles*.

Pero para que el software que se exprese en estos lenguajes de mayor abstracción culmine en instrucciones que sean de las directamente ejecutadas por el procesador, es necesario desarrollar otro software que sirva de "intermediario". Es decir *desarrollar software* que sirve para *desarrollar software*[11]. De modo que pueden existir distintas "capas de software" intermediarias entre el contexto del problema que se quiere solucionar (lo que se denomina *espacio del problema*) y el hardware que finalmente va a realizar el procesamiento físico para hacer perceptible la solución a dicho problema (lo que se denomina *espacio de la solución*).

El acercar estos extremos se ataca por dos vías, de "arriba hacia abajo" y de "abajo hacia arriba". De *arriba hacia abajo* porque a partir del problema se utilizan formalismos y lenguajes suficientemente expresivos que faciliten encontrar y formular la solución para luego con la ayuda de software de traducción (tradicionalmente conocidos por *compiladores*[12]) traducir lo expresado en el nivel superior de abstracción hacia una capa inferior de software. Se repite entonces un proceso similar en cada capa o nivel hasta que finalmente se llegue a una capaz de ser "entendida" por el hardware. De *abajo hacia arriba* porque sobre el hardware se añaden capas de software que amplían de manera *virtual* estas capacidades del hardware ofreciendo instrucciones y funcionalidades de mayor generalidad que son emuladas por las capas más inferiores, de modo que en el proceso *arriba hacia abajo* no haya que llegar "tan abajo".

Se dice *virtual* porque esta ampliación de las capacidades del hardware no significa añadir ninguna componente física sino que se basa en añadir componentes de software. De ahí que a este hardware unido con las capacidades añadidas por software, y que emulan a instrucciones de mayor abstracción, se le suele llamar *máquina virtual*. Estas máquinas virtuales emulan instrucciones de mayor funcionalidad y expresividad y utilizan "bibliotecas de componentes de software" que completan esa funcionalidad. Con estas bibliotecas se evita repetir la formulación de partes del software. De modo que un procesador puede no tener una instrucción para indicarle directamente, y de manera fácil, al procesador *poner en rojo un punto en las coordenadas x, y del monitor*, ni mucho menos para decir algo como *cuando se haga click en este botón de la ventana ésta debe minimizarse de tamaño*[13]. Sin embargo, esto puede ser simple de expresar en la máquina virtual y su biblioteca acompañante, que harán el trabajo "sucio" de llevarlo a las elementales instrucciones comprensibles por el procesador.

La existencia de una tal máquina virtual facilita y hace más productivo el proceso de desarrollar el software aporque acorta la distancia entre el problema a resolver y el hardware sobre el que se procesará. Pero tiene además varias ventajas adicionales. Una de estas ventajas es que el software final es más confiable porque el desarrollador no tiene que utilizar esas grandes secuencias de instrucciones cercanas al hardware para lograr un determinado efecto (como ese de

[11] Y vaya con los energúmenos que dicen que para practicar esta profesión de desarrollar software no se necesita saber de recursividad , si es que la propia naturaleza de la profesión es recursiva!.

[12] Del inglés *compiler*, aunque compilar en español mas que traducir (que es como se usa el término en este caso) es recopilar, reunir.

[13] De hecho los conceptos de *botón* y de *ventana* son a su vez conceptos virtuales recreados por software, ya que lo que tenemos realmente por hardware son un monitor y un ratón.

minimizar una ventana de tamaño). La otra ventaja es que el software resultante no estará atado al hardware, es decir que podrá ejecutar sobre diferentes hardware, siempre claro está que para cada hardware se haya desarrollado una máquina virtual similar. Esto que se conoce como *independencia del hardware* o *portabilidad* del software, es prácticamente una característica imprescindible del software moderno debido a la gran diversidad del hardware existente y a la alta interconexión del mismo, que hace que muchas veces los que desarrollen un software no tengan que saber con exactitud sobre cuál hardware se ejecutará.

La plataforma .NET sobre la que se apoyará la ejercitación práctica, y el lenguaje C# que se utilizará en este libro, se sustentan en una tal máquina virtual, conocida en este caso como *Common Language Runtime* (CLR) y que está acompañada de una amplia biblioteca denominada *Basic Common Library* (BCL)[14]. El software de una aplicación que se escriba en C# será traducido por un compilador a un lenguaje intermedio conocido como IL[15], luego el CLR al ejecutar la aplicación traducirá este IL a instrucciones propias del procesador utilizando una tecnología conocida por compilación JIT[16]. El propio CLR controlará la ejecución de las instrucciones resultantes y utilizará las componentes de la BCL.

La plataforma .NET incluye compiladores para otros muchos lenguajes de programación que también se traducen a IL (por ejemplo C++, Visual Basic y Java) y que pueden interoperar y utilizarse entre sí. La plataforma .NET se sustenta en el modelo orientado a objetos. Es el método de diseño y programación orientado a objetos el que hace de columna vertebral del presente libro.

El lenguaje C# ha sabido recoger buena parte de lo mejor de otros lenguajes precedentes sin con ello afectar su simplicidad y puede considerarse un paradigma en el método de programación orientado a objetos. No obstante, como se dice en el prefacio, se sale del alcance de este libro, y no forma parte de sus objetivos, estudiar C# en profundidad. Lo que interesa es utilizar C# como vehículo propicio para introducir los conceptos fundamentales de la programación y de la orientación a objetos y para crear buenos hábitos y habilidades de programación. Con ello el lector interesado por otros lenguajes (o que las circunstancias le exijan trabajar con otros lenguajes) debe ser capaz de expresar estos conceptos y habilidades en dichos lenguajes[17].

7.3 Etapas Del Proceso De Desarrollo De Software

El proceso de desarrollo de un software va desde el planteamiento del problema hasta la explotación de su solución computacional al ser ejecutado sobre un determinado hardware el software desarrollado. Estas etapas suelen clasificarse en

1. *Planteamiento del problema y análisis del sistema*
2. *Análisis de requisitos*

[14] En el momento de escritura del presente libro, el CLR y el BCL de .NET así como un compilador de C# sólo existen sobre el sistema operativo Windows de Microsoft y por consiguiente para los hardware sobre los que existe este sistema operativo (la gran mayoría por cierto). Pero están en desarrollo implementaciones para el mundo Unix que ha reconocido las grandes bondades de esta plataforma.

[15] Siglas del inglés *Intermediate Language*.

[16] Siglas del inglés *Just In Time* que se puede traducir al español en algo como *en el momento preciso*.

[17] Lo cual requerirá de mayor o menor esfuerzo en dependencia del lenguaje. Esto por supuesto será más fácil para lenguajes que sean soportados por la plataforma .NET (lo que es de hecho un conjunto creciente) entre otras cosas por el uso común que hacen de la BCL.

3. *Diseño*
4. *Programación*
5. *Prueba*
6. *Mantenimiento*

En el ***planteamiento del problema y análisis del sistema*** se analiza el problema a resolver así como la relación del software con otros elementos, tales como hardware, datos, personal y otras disciplinas que intervendrán o con las que hay que relacionarse en el proceso. Por ejemplo para el desarrollo de un software para comercio electrónico hay que tener en cuenta infraestructura de almacenamiento, distribución y transportación, formas de pago, elementos de seguridad, etc.

El *análisis de los requisitos* se centra especialmente en el software. Cuál es el ámbito de la información en que trabajará el software, qué funcionalidad se espera del software, cuál debe ser el rendimiento, cómo será explotado el software, cuáles serán las interfaces del software (sea para interactuar con los usuarios que lo explotarán o a su vez con otro software). Estos requisitos debieran ser especificados en un formalismo o lenguaje de especificación.

En el *diseño* se intentan llevar los requisitos a una "representación" del software en términos de organizar los datos y las acciones (lo que se suele denominar arquitectura del software), determinar las distintas componentes que participarán y las interfaces entre ellas. En la organización de las acciones se conciben los *algoritmos* que se necesiten para expresar en más detalle esas acciones.

Dicho de modo informal y simple un algoritmo es una secuencia finita de acciones que deben ser "realizables" y que deben llevar a que el algoritmo termine en un tiempo finito.

En la *programación* se escribe (también se dice codifica) las representaciones de los datos y los algoritmos en un lenguaje de programación que sea "entendible" por el hardware o la "máquina virtual" (por lo general porque median en este proceso diferentes capas de compilación o traducción). Es la programación la que elimina las comillas para garantizar que las acciones del algoritmo diseñado sean "realizables".

Una vez que se tiene el programa sin errores detectados en la fase de anterior de programación-compilación (lo que se conoce como *errores de compilación*) es que se debe entonces probar que el programa no tenga *errores de* ejecución y que produzca los resultados esperados según los requisitos establecidos. Para ello se trata de ejecutar el programa con datos de prueba que provoquen la ejecución de todos los caminos u opciones contempladas en el programa y analizar los resultados o efectos que esto produce.

El *mantenimiento* ha sido tal vez históricamente la etapa más marginada y sin embargo la que más costos ha ocasionado[18]. En el software moderno que es cada vez más complejo y está destinado a la solución de problemas más complejos (muchos usuarios posibles, escenarios diferentes de uso, interrelación con muchos dispositivos, gran volumen de información a manejar), por ello es prácticamente imposible que en la etapa de prueba se corrijan todos los errores. Muchos

[18] Tal vez el término *mantenimiento* no es el más exacto porque el software, a diferencia de otros dispositivos más tangibles, no se desgasta por su uso continuado. Puede que sea más correcto decir *modificación*, *cambio*. Sin embargo, mantenimiento es el término que ha quedado.

errores surgirán cuando ya el software esté en explotación por los clientes y habrá que corregirlos. Nuevos requerimientos aparecerán que provocarán la necesidad de hacer modificaciones, adaptaciones o ampliaciones[19]. Se dice que del software que se termina, mucho no cumple con los requerimientos iniciales y que del que cumple con los requerimientos iniciales buena parte ya es obsoleto porque han aparecido nuevos requerimientos.

De modo que este proceso es en cierto modo un *ciclo* porque el mantenimiento lleva a empezar de nuevo en alguna de las etapas anteriores. Por ello se le suele denominar *ciclo de vida del software*.

En el desarrollo de software moderno los lenguajes de programación están integrados a herramientas o entornos de desarrollo[20] que ayudan en este ciclo de vida del software. Estas herramientas integran al compilador con *editores de texto* que facilitan la escritura del texto de los programas y la realización de cambios en estos textos una vez detectados errores por el compilador. A su vez estas herramientas incluyen también *depuradores* que facilitan la prueba de los programas "visualizando" la ejecución de los mismos. Estas herramientas disponen también de *inspectores o navegadores* que permiten explorar las bibliotecas de piezas de software disponibles inspeccionando sus características y funcionalidades para poder incorporar estas piezas al software que se está programando.

7.4 El Método Orientado A Objetos Y C#

Este libro se centra fundamentalmente en la etapa de *diseño* y la *programación* mediante el método orientado a objetos y usando un lenguaje de programación que favorezca la aplicación de este método, lo que se conoce como *lenguaje de programación orientado a objetos*.

Para beneficio del ciclo de vida del software expuesto anteriormente, el método orientado a objetos (o más simplemente la *orientación a objetos*) tiene tres aportes importantes. Sus conceptos y técnicas favorecen un mejor modelado de la realidad y por tanto hacen al diseño y a la programación más expresivos y simples con vistas a plasmar los requerimientos, características e interrelaciones del problema a resolver. Por otra parte la orientación a objetos favorece la reutilización del software al disponer de recursos para integrar en el software en construcción piezas de software ya existentes y probadas; lo que por consiguiente disminuye la cantidad de errores y aumenta la confiabilidad. Por último con la orientación a objetos, gracias a la herencia y el polimorfismo, se puede con facilidad ampliar, adaptar o especializar piezas de software ya existentes (como se verá en el capítulo 2 estas "piezas" de software se denominan clases) sin con ello poner en riesgo la funcionalidad previa de éstas.

El lenguaje C# con el que se trabajará en este libro es simple y a la vez altamente expresivo a la hora de implementar conceptos modernos de programación. C# incluye soporte completo a la programación orientada a objetos tomando lo mejor de lenguajes precedentes como Delphi, C++, Visual Basic y Java.

C# fue desarrollado por un pequeño team de Microsoft liderado por Anders Hejlsberg creador de los populares Turbo Pascal y Borland Delphi.

---

[19] Es poco probable que después de construido un puente de cuatro sendas se le pida al constructor añadirle dos nuevas sendas. Sin embargo, demandas equivalentes se le piden con frecuencia a los desarrolladores de software.

[20] Conocidas en ingles por las siglas IDE (*Integrated Development Environment*)

Como se verá en los siguientes capítulos, en el corazón de un lenguaje orientado a objetos está la forma en que éste soporta definir y trabajar con clases. Con las clases se definen nuevos tipos que permiten extender el lenguaje para modelar mejor el problema que se está tratando de resolver. C# contiene todos los ingredientes para declarar nuevas clases con sus métodos y propiedades, y para la implementación los tres pilares del la programación orientada a objetos: el encapsulado, la herencia y el polimorfismo.

En C# todo lo que pertenece a la definición de una clase está dentro de la declaración de ésta. La definición de una clase no necesita de ficheros de encabezado o de ficheros de lenguajes de definición de interfaces[21]. C# incluye un nuevo estilo de documentación basado en el estándar XML[22] y que simplifica grandemente la documentación de una aplicación y permite integrarla a las múltiples herramientas que en el software moderno trabajan sobre XML.

Más allá de su suporte a la orientación a objetos C# incluye también soporte a características orientadas a componentes tales como los *eventos*. C# tiene una construcción denominada *atributos* que permite incluir información declarativa asociada a una clase. Esta información se almacena junto con el código resultante de la compilación de la clase y forma parte de lo que se denomina *metadato* de una componente.

El resultado de la compilación de una o varias clases en C# queda en *ensamblados* (*assemblys*). Estos están almacenados en archivos con extensión `.EXE` o `.DLL` En un ensamblado están los metadatos que describen a las clases y a sus métodos, propiedades, elementos de seguridad. De este modo el resultado de la compilación es una unidad autocontenida que puede ser reusada, transportada y dotada de mecanismos de seguridad. De este modo que un software (como puede ser una herramienta integrada de desarrollo) que sepa (y tenga los derechos para) cómo “leer” los metadatos y el código que están en un ensamblado no necesite de ninguna otra información para usar las clases que están en éste. Un programa escrito en C# puede a su vez en ejecución inspeccionar sus propios metadatos y de los ensamblados que utiliza, gracias a recursos conocidos como reflexión (*reflection*).

7.5 De Los Mainframes A Internet

Los primeros modelos de computadoras de los años 50s y 60s e incluso los 70s estaban constituidos por la máquina o computadora principal (*mainframe*) que contenía al procesador y a la memoria, que por lo general se encerraban en una misma caja. Adicionalmente esto estaba complementado con voluminosos dispositivos periféricos, como los magnéticos para almacenar información, o las grandes impresoras y lectoras de tarjetas perforadas de cartulina. Todo este equipamiento era muy costoso y por lo general ocupaba habitaciones completas, tenía que estar bien climatizado pues eran grandes disipadores de calor y “celosamente” custodiado.

Muy pocos programadores tenían el privilegio de trabajar directamente con la máquina, otros tenían que conformarse con dejar sus trabajos para que operadores fueran los que interactuaran con la máquina posiblemente para que tal vez tres días después su programa fuese rechazado porque faltaba un punto y coma en una instrucción del programa.

[21] Conocidos en inglés por las siglas IDL (*Interface Definition Languege*)

[22] e**X**tended **M**arkup **L**anguage

Se escribían las instrucciones en un lenguaje de programación para luego hacérselas llegar al compilador (por mediación tal vez de operadores humanos), corregir los errores de sintaxis detectados por el compilador y así hasta que no tuviera errores de sintaxis. Luego intentar la ejecución del programa traducido (de nuevo por mediación tal vez de operadores humanos) probando con datos para verificar si se cumplían los requerimientos del problema. Una vez considerado el programa como depurado o puesto a punto este se ponía en explotación por los usuarios finales puede que utilizando su propio equipamiento que debía ser similar al sobre el que se desarrolló o posiblemente utilizando el mismo equipamiento sobre el que se desarrolló.

El proceso de desarrollar software así era muy lento e improductivo. Esto condicionaba el tipo y la cantidad de problemas a plantearse a resolver.

Ya a finales de los 60s y en los 70s se desarrollaron las minicomputadoras que eran de menor costo y disminuían el volumen que ocupaban. Esto vino aparejado con mejor software para soportar el proceso de desarrollo y mayores posibilidades de acceso al equipamiento tanto de parte de los desarrolladores como de parte de los usuarios. Mayor productividad de desarrollo y mayor accesibilidad entonces mas variedad de problemas a resolver.

Pero no fue hasta finales de los 70s y principios de los 80s en que los adelantos en la microelectrónica y la aparición de las *microcomputadoras* provocaron una sacudida sísmica en la forma de desarrollar y utilizar el software. Por primera vez tanto desarrolladores como usuarios pudieron poner la computadora (y sus "periféricos") sobre su mesa o escritorio (de ahí el término *desktop computer*). Una explosión de usuarios y una explosión en la variedad de problemas y aplicaciones (ahora sí cada escritor podía escribir su novela usando la computadora para escribir y corregir su texto de forma simple y visual).

Aunque la programación orientada a objetos puede decirse que nació a finales de los 60s con el lenguaje SIMULA 67 fue a la par de la explosión de las microcomputadoras que se popularizaron implementaciones de lenguajes orientados a objeto como Smalltalk, C++ y Turbo Pascal.

De ahí para acá la historia debe ser más conocida para el lector. Con dos PCs aislados y en diferentes lugares, dos usuarios diferentes no pueden estar reservando a la vez asientos en un mismo vuelo de avión, al menos sin correr el riesgo de que uno de los dos se quede sin viajar. De modo que poco tiempo después nos "inventamos" la "conveniencia" o la "necesidad" de conectar las computadoras entre ellas para que se pudieran comunicar y surgieron entonces las *redes* de computadoras.

Las primeras redes eran redes locales, por lo general cubrían un área no muy grande en una misma institución. El equipamiento utilizado era bastante homogéneo y las velocidades de transmisión aún insuficientes para enviar imágenes, sonidos, etc. Pero ya los usuarios y los desarrolladores se podían enviar mensajes y usuarios en distintos puntos de la red podían compartir información. Los dos usuarios queriendo reservar boletos en una misma aerolínea podían ahora estar comunicándose con el software que hiciera de central de reservas ejecutando en un tercer punto de la red. Surge entonces el modelo conocido como *cliente-servidor*. Un software "cliente" ejecutando en una computadora de la red solicita un servicio a un software "servidor" preparado para brindar ese tipo de servicios y probablemente ejecutando en otro computadora de la red (es decir la ejecución de determinada acción posiblemente para que se le regrese determinada información). Nuevos problemas a resolver como el de que dos usuarios reales conectados al servidor podían disputar una partida de bridge con dos usuarios virtuales (es decir emulados por software) ejecutados en el servidor. Este tipo de software que trabaja en el modelo cliente-servidor requiere de conocimientos y coordinación muy específicas entre el que desarrolla el software del cliente y el que desarrolla el software del servidor (muchas veces un mismo equipo de programadores usando el mismo lenguaje de programación y las mismas herramientas).

Pero las computadoras siguieron conectándose y las redes ampliándose. Se empezaron a crear redes de redes en lo que ha devenido en esa telaraña[23] de computadoras que es Internet. Esta gran interconexión con la mayor capacidad y velocidad de transmisión hacen ahora posible que un nuevo problema a resolver sea el de ver (con todo y fotos a color) un periódico editado en la otra cara del planeta. Si obviamos los aspectos de seguridad y protección de la información (que introducen otro nivel de complejidad en el software), los nuevos problemas planteados al software son relativamente simples. Hay que disponer de un software servidor, que está donde radica la información o contenidos, que organice y envíe la información a aquellos softwares clientes que la solicitan (el software cliente que "visualice" al usuario humano dicha información). Pero a diferencia de la arquitectura cliente servidor, explicada anteriormente, para redes locales, ahora queremos que el equipamiento y software disponible en el cliente pueda ser totalmente diferente del utilizado en el servidor. El problema es simple ya que todo parece resolverse definiendo estándares de comunicación que todos los servidores respeten y que los softwares clientes (navegadores) sepan visualizar[24].

Pero aquí no termina el asunto pues resulta que ahora nos planteamos "nuevos problemas". Queremos que el cliente no esté necesariamente conectado mediante una computadora de sobremesa tradicional, sino que puede estar en un teléfono móvil, o con una computadora de bolsillo, o que el cliente sea otro software empotrado en un equipo domestico como un refrigerador! El concepto de navegador debe variar porque las posibilidades y requerimientos de "visualización" y dónde están los clientes puede variar. Es decir el cliente no tiene que ser ahora un humano usando el navegador Internet Explorer o Netscape, sino que el cliente puede ser a su vez otro software. También el concepto de la información que se trasmite por la red debe variar porque ésta no sólo será un contenido pasivo que está almacenado en el servidor. La información puede ser "creada" a la medida según lo que haya solicitado dinámicamente el software cliente, es decir la solicitud que hace el software cliente puede requerir de un procesamiento a realizar en el servidor, que a su vez puede requerir un procesamiento a otro servidor. El software que va a realizar este procesamiento puede requerir de parámetros que le deba enviar el software cliente. Un determinado servicio o componente de software no tiene que ser físicamente replicado en cada cliente sino que puede ser ofrecido desde Internet. Esta forma de ofrecer servicios se ha denominado *servicio web*.

Tienen que haber entonces lenguajes y herramientas de software para desarrollar los servicios, para desarrollar las aplicaciones que usen esos servicios y que pueden comunicarle al servidor los parámetros que exigen los servicios, así como para incorporar a la aplicación los resultados devueltos por tales servicios. Pero también debe haber software para dar a conocer los servicios que se ofrecen y para investigar cuáles son los servicios que se ofrecen. En fin, una nueva forma de plantearnos problemas y abordar sus soluciones.

Estamos ante una sacudida sísmica similar a la de la aparición de las microcomputadoras pero como ahora todo está en permanente cambio es imposible pedir una tregua para observar. Estamos en los umbrales de convertir a Internet en una gran computadora virtual y necesitamos modelos, lenguajes y herramientas de software para esa computadora y para abordar la nueva concepción de problemas y aplicaciones que se nos caen encima. Es éste el contexto para el que se ofrece la tecnología .NET. Como se dice en el prefacio, este libro no pretende estudiar la tecnología .NET, ni pueden resolverse aquí por razones de espacio y tiempo problemas como los arriba

[23] Telaraña en inglés es *web* de ahí el uso del término y de la trilogía *www* iniciales de las palabras *world wide web* o gran telaraña mundial.

[24] Eso es por ejemplo el protocolo de transmisión http y el lenguaje HTML para representar la información.

sugeridos. Sin embargo, el uso del lenguaje C# que es el lenguaje paradigma de .NET y muy apropiado para presentar los conceptos de la programación y de la orientación a objetos (que de hecho están a su vez en el propio "genoma" de .NET) deben servirle al lector de buen punto de partida para cuando tenga que enfrentar otros estudios.

7.6 Disección De Los Primeros Programas En C#

Para seguir la tradición de los libros de programación analicemos el clásico programa de bienvenida, con ello se introducirán algunas características de C# así como de su sintaxis y notación.

```
class MiPrimerPrograma{
  static void Main(){
    // Se imprime un mensaje de saludo
    System.Console.WriteLine("Bienvenido a C#");
  }
}
```

Si se compila y ejecuta este programa se mostrará la frase `Bienvenido a C#` en la consola o monitor.

(gráfico de la consola)

Este programa define una única clase de nombre `MiPrimerPrograma`. Para ello se ha utilizado la palabra reservada `class` seguida del nombre que quiere dar a la clase. Una palabra reservada o palabra clave[25] es una palabra con un uso específico dentro del lenguaje y que no puede utilizarse para otro fin, en este ejemplo `static` y `void` son también palabras reservadas. Las palabras reservadas se irán introduciendo a lo largo del texto según se utilicen para expresar los diferentes recursos de programación que se utilizarán de C#.

La definición de una clase va encerrada entre las llaves `{` y `}`.

Esta clase consta de un método de nombre `Main` que esté precedido de las palabras `static` y `void`. El significado de la palabra reservada `static` se explica en el capítulo 2. La palabra reservada `void` significa que el método `Main` no devuelve de manera directa ningún resultado. El método `Main` está seguido de los paréntesis `(` y `)`, el espacio entre los paréntesis está dedicado para indicar los parámetros utilizados por el método, pero en este caso no se utiliza ningún parámetro. Una aplicación que contenga una clase con un método `static void Main()` significa que la ejecución de la aplicación comenzará por ejecutar a dicho método. Por convenio el compilador de C# dejará el resultado de la compilación de la tal clase en un fichero con extensión `.EXE`.

A su vez la implementación del método (lo que se conoce también como *cuerpo del método*) va encerrada entre dos llaves `{` y `}`. La línea

```
// Se imprime un mensaje de saludo
```

que está dentro del método es un *comentario*. Todo texto que comience con un par de diagonales (*slash*) `//` es un comentario. Un comentario es una nota al programador y no tiene que

[25] Por el término en inglés por *key word*.

ver con el funcionamiento del programa, no es traducido por el compilador de C# a ninguna instrucción ejecutable.

Un comentario que comienza con `//` termina en el final de la línea. De modo que

```
//Si esto pretende ser un comentario será un error ya que el compilador de C#
pretende interpretar esta segunda línea como una instrucción de C#
```

Si un comentario ocupa más de una línea puede escribirse

```
// Este comentario ocupa
// dos líneas
```

Hay una forma más cómoda para expresar comentarios de más de una línea, se empieza el comentario con los símbolos `/*` y se termina con `*/`

```
/* Este comentario ocupa mas de una línea y
correcto */
```

La siguiente línea del método `Main` es

```
System.Console.WriteLine("Bienvenido a C#");
```

Este `MiPrimerPrograma` es un ejemplo de programa por *consola*. Una aplicación de consola no hace uso de ninguna interfaz gráfica, es decir no hay íconos visuales como botones, barras de *scroll*, etc. Esta consola muestra una ventana de la forma

(gráfico)

La entrada y salida de información por esta consola es en forma de texto. `System.Console` es el nombre de una clase de la biblioteca BCL. `System` es el nombre de un espacio de nombres (*namespace*). Los *namespaces* se utilizan para agrupar los nombres de varias clases (estos se estudian con más detalle en el capítulo 2). `Console` es el nombre de una clase en `System` que es la que se utiliza cuando se quiere hacer entrada y salida de texto por la consola.

`WriteLine` es el nombre de un método en la clase `Console` que escribe por la consola el texto formado por una secuencia de caracteres. La secuencia de caracteres `"Bienvenido a C#"` que aparece dentro de los paréntesis `(` y `)` es un objeto de tipo `string` (el tipo `string` se estudiará en el capítulo 4 XXX) que es el tipo de parámetro que hay que pasar al método `WriteLine`. Toda secuencia de caracteres que se quiera indicar como un texto debe encerrarse entre comillas dobles `" "` (las comillas no forman parte del texto, solo sirven para que C# distinga el texto del resto del programa).

El operador punto (`.`) se usa para marcar un ámbito dentro del cual queremos referirnos a un elemento de ese ámbito. Así `System.Console` indica que hablamos del `Console` en el ámbito de `System` (lo que permite que pudiera haber otro `Console` dentro de otro ámbito). A su vez `System.Console.WriteLine` se refiere al método `WriteLine` dentro de la clase `System.Console`.

C# es *case-sensitive* es decir sensible a la diferencia entre mayúsculas y minúsculas. Esto significa que `WriteLine` no es lo mismo que `writeLine`. Si en este programa se hubiese escrito `writeLine` esto reportaría error de compilación porque dentro de la clase `System.Console` no existe un tal método `writeLine`.

Para evitar confusiones se sugiere al lector utilizar un convenio para nombrar a sus entidades (variables, parámetros, clases, métodos, propiedades). Para mayor uniformidad en este libro se

utilizará el mismo convenio adoptado por la biblioteca BCL. Las variables y parámetros se escribirán comenzando en minúscula `variable`, si se escribe un nombre compuesto el segundo nombre empezará en mayúscula `otraVariable`. Los nombres de métodos, propiedades, clases empezarán con mayúscula, como por ejemplo `MiPrimerPrograma` y `WriteLine`.

Si el lector ya intentó probar este primer programa antes de leer este párrafo posiblemente esté algo decepcionado. La salida de los resultados por la consola debe haber ocurrido con tal velocidad que probablemente no haya alcanzado a leer el mensaje de bienvenida. Este inconveniente se solucionará rápidamente a continuación con nuestro segundo programa. Este le va a pedir al usuario que teclee su nombre para que reciba un saludo personalizado. El programa se acercará más a los problemas reales: recibir datos de entrada, procesarlos y dar los resultados como salida:

```
1  class MiSegundoPrograma{
2    static void Main(){
3    System.Console.WriteLine("Bienvenido a C#");
4    System.Console.WriteLine("Por favor, teclee su nombre y pulse Enter");
5    //Leer el texto que teclee el usuario y guardarlo en la variable s
6    string s = System.Console.ReadLine();
7    System.Console.WriteLine("Mucho gusto " + s +
8                             "Esperamos nos acompañes hasta el final");
9    //Esperar por una entrada cualquiera para darle tiempo a leer
10   System.Console.ReadLine();
11 }
```

Fíjese en la línea 6, la operación `System.Console.ReadLine()` invoca a otro método de `System.Console` que se utiliza para "leer" una línea de texto que el usuario teclee. En este caso el cursor quedará parpadeando en la consola y el programa no continuará su ejecución hasta que el usuario no teclee un texto terminando por un cambio de línea (al pulsar Enter)). El texto leído se "guarda" en una variable `s` para utilizarlo después. Para que sirva para guardar un texto esta variable tiene que haberse declarado de tipo `string` (en C# todas las variables tienen que declararse con algún tipo).

En la líneas 7 y 8 se vuelve a escribir un texto hacia la consola por medio de `System.Console.WriteLine`. El texto que se manda a escribir está formado por la concatenación de tres textos: el texto `"Mucho gusto "` el texto que está guardado en la variable `s` y el texto `"Esperamos nos acompañes hasta el final"`. El símbolo de suma `+` se utiliza también en C# para concatenar textos. La acción se ha separado en dos líneas por las limitaciones del ancho de la página en que Ud. está leyendo esto, pero para C# podía haberse escrito en una única línea.

Al final, línea 10, se tiene un nuevo `System.Console.ReadLine()`. Note que en este caso el resultado de la lectura no se guarda en ninguna variable porque no nos interesa. El único objetivo de usar `ReadLine` aquí es esperar a que el usuario teclee Enter y dar con ello la posibilidad de ver los resultados escritos con anterioridad.

La clase `System.Console` y los métodos `WriteLine` y `ReadLine` nos acompañarán en múltiples ejemplos a partir de ahora.

7.7 Notación Utilizada En Este Libro

Como se utilizó en la sección anterior cuando se muestren ejemplos en C# y cuando dentro del texto se haga mención a nombres utilizados en los ejemplos, a palabras reservadas de C# o a nombres utilizados en la biblioteca BCL se usará el formato `courier new`.

En lo adelante por razones elementales de espacio y para simplificar la lectura, muchos ejemplos no se mostrarán completos. Se utilizará la notación `...` (puntos suspensivos) para indicar la presencia de un posible texto en C# pero que no es imprescindible para la comprensión del ejemplo que se está explicando. Así por ejemplo si el lector se encuentra con

```
class MiPrimerPrograma{
  static void Main(){
    // Se imprime un mensaje de saludo
    System.Console.WriteLine("Bienvenido a C#");
  }
  ...
}
```

Los puntos `...` indican la presencia de un posible texto dentro de la clase `MiPrimerPrograma`. Pero si se encuentra con

```
class MiPrimerPrograma{
  static void Main(){
    // Se imprime un mensaje de saludo
    System.Console.WriteLine("Bienvenido a C#");
    ...
  }
}
```

Los puntos `...` indicarían en este caso la presencia de un posible texto dentro de la definición del método `Main`.

Note que esto es sólo un convenio de notación. El intento de compilar un texto con estas características será reportado error por el compilador porque no reconoce la secuencia `...` como una construcción sintácticamente correcta en C#.

Este libro no pretende describir de manera formal y completa la sintaxis de C#, para ello el lector interesado puede referirse a la documentación libre disponible en Internet, a la ayuda de la herramienta de desarrollo que esté utilizando (posiblemente Visual Studio .NET).

Los elementos de sintaxis que el lector deba conocer se irán introduciendo a lo largo del texto según se vayan necesitando para explicar los diferentes conceptos que se vayan estudiando. Esto, complementado con los mensajes de error que el compilador le emitirá, deberá ser suficiente para comenzar a programar en C#.

Cuando se exponga algún fragmento de sintaxis de C# se utilizará letra normal en cursiva y encerrada entre los angulares < y > para denotar algún concepto de C# que o es evidente su significación o puede ser explicado en otro lugar del texto. De modo que cuando se dice que una clase en C# se escribe en la forma

```
class <nombre de la clase> {
  <definición de la clase>
}
```

Se quiere decir con ello que lo que aparece en formato `courier new`, es decir `class`, `{` y `}` sea porque se escribe exactamente así en C#, pero que *<nombre de la clase>* y *<definición de la clase>* son elementos de los que o bien basta con una comprensión intuitiva del lector (como es el

caso de <*nombre de la clase*>) o de los que una mayor explicación no se quiera entrar en detalle aquí y se debe encontrar en otro lugar de texto.