1 INTRODUCCIÓN

Si decimos que el propósito de este libro es ayudarlo a aprender a programar, tenemos que ponernos de acuerdo en *¿qué es programar*?

Dicho de modo breve y simple: **programar** es desarrollar un código expresado mediante un conjunto de sentencias en una abstracción (lenguaje de programación) que de algún modo terminan siendo "entendibles " por un aparataje físico concreto (hardware), de modo que al ser procesadas por éste nos dan la "solución " a un problema.

Y aquí los problemas, los lenguajes de programación y el hardware se dan la mano. Los tipos de problemas que nos planteamos solucionar han dependido de las capacidades del hardware, pero a su vez el interés en solucionar nuevos problemas ha incentivado el desarrollo del propio hardware. Y como intermediario, los lenguajes y herramientas de programación para salvar la distancia entre el hardware y los problemas. Tenemos entonces una espiral creciente con cada vez mejores recursos de hardware, lenguajes y herramientas de programación, pero a la vez son más complejos y sofisticados los problemas que queremos resolver.

1.1 De la Máquina de Babbage a la Computación en la Nube

Tal vez la más aceptada como *primera computadora* (aunque esto no pretende ser la punta de la madeja) sea la conocida como *máquina de Babbage*[1] a mediados del siglo XIX. 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. En su época ésta 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 con memoria para guardar los datos y las instrucciones a ejecutar.

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 según algunos, esto es discutible[2].

Pero las instrucciones o *programa* de los ingenios de Babbage quedaban fijos en el hardware de la tal máquina. Es decir, no se podían introducir modificaciones sin de algún modo tener que hacer cambios físicos en el aparataje, lo que limitaba la versatilidad de una tal máquina.

[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

Tuvo que pasar medio siglo hasta finales de los años 40 del siglo XX[3], en que se pudo hablar de máquinas de propósito más general, es decir, máquinas que fuesen capaces de entender y ejecutar un conjunto fijo de instrucciones (para ello disponían de un **procesador**[4]). El gran salto cualitativo fue que si se le indicaban al procesador distintas secuencias de este conjunto de instrucciones (se le llama **programa** a una de estas secuencias), entonces una misma máquina podría servir para realizar acciones diferentes y "solucionar" con ello diferentes problemas. Para que la máquina sepa cuál es la combinación específica de instrucciones que se le pide ejecutar, debe disponer de una **memoria** en la que se "guarda" esta secuencia, de modo que cambiando la secuencia de instrucciones que se pone en la memoria una misma máquina puede solucionar problemas diferentes. Esta memoria sirve a su vez para almacenar los datos iniciales que se procesarán y los que se obtienen como consecuencia del procesamiento.

Esta arquitectura basada en la existencia de un 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.

Los primeros problemas que dieron lugar a estas primeras computadoras o los primeros problemas que pudieron ser resueltos con estas primeras computadoras eran de naturaleza eminentemente numérica. El hardware realizaba cálculos numéricos aprovechando con ello la "velocidad" de procesamiento (que permitía dar respuesta en un tiempo aceptable a cálculos inviables de realizar manualmente) y su "confiabilidad" (porque podían 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 que trabajaban aisladamente hasta este gran ecosistema de disímiles dispositivos interconectados que es hoy Internet, no ha sido más que un proceso en espiral de *nuevos problemas a resolver, entonces se necesitan dispositivos con nuevas y mejores capacidades y se crean nuevas dispositivos 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 (que se trababan) y cinta entintada (que emborronaba el papel). Unos 30 años después disponíamos de pantallas para mostrar textos y software que permitía escribir documentos (y también modificarlos, adaptarlos y enmendarlos con facilidad) pero no se pensaba aún que una foto tomada en un lugar del planeta se pudiese ver con inmediatez en el otro extremo del mismo. ¿Cuáles problemas cree el lector podremos solucionar con las capacidades del hardware y el software dentro de 20 años o qué capacidades de hardware y software se tendrán que desarrollar para resolver qué problemas?

---

[3] Aquí algunos países y algunos historiadores discrepan de quién 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.

[4] Los hoy llamados CPU (del inglés Central Processing Unit)

[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. Estamos hablando de los años de la segunda guerra mundial y de la posterior guerra fría.

1.2 El Espacio del Problema y el Espacio de la Solución

Desarrollar software para la solución de un problema es todo un proceso que culmina en una secuencia de instrucciones en un lenguaje que el hardware sea capaz de procesar.

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 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 más nos planteamos.

Para salvar la distancia entre problemas y hardware se han desarrollado lenguajes y herramientas de ayuda a la programación cada vez de más alto nivel de abstracción 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 hardware, es necesario desarrollar otro software que sirva de "intermediario". Es decir, *desarrollar software* que sirve para *desarrollar software*. 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**), que es cada vez más multivariado y ubicuo.

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 herramientas de software de traducción (tradicionalmente conocidos por *compiladores*[6]), 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" directamente 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.

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

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

máquinas virtuales emulan instrucciones de mayor funcionalidad y expresividad y utilizan "bibliotecas de componentes de software" que completan esa funcionalidad. Con estas máquinas virtuales y bibliotecas se evita repetir la formulación de partes del software. De modo que un CPU puede no tener en su repertorio una instrucción que diga "*cuando se haga clic en este botón de la ventana ésta debe minimizarse de tamaño*"[7]. 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 porque 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 de CPU. La otra ventaja es que el software resultante no estará atado al hardware, es decir que podrá ejecutar sobre diferente hardware, siempre 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 de dispositivos y de la ubicuidad e interconectividad de los mismos, que hace que muchas veces los que desarrollen un software no tengan que saber con exactitud sobre cuál hardware se ejecutará.

1.3 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 requerimientos*
3. *Diseño*
4. *Implementación-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 requerimientos*** se centra especialmente en el software. Cuál es el ámbito de la información en que trabajará el software que se desarrollará, qué funcionalidad se espera de este, cuál debe ser el rendimiento, cómo será explotado, cuáles serán las interfaces de dicho software (sea para interactuar con los usuarios que lo

[7] 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 (display) y un ratón (mouse).

explotarán o a su vez con otros dispositivos y software). Estos requisitos debieran ser especificados en un formalismo o lenguaje de especificación de requerimientos.

En el ***diseño*** se intenta plasmar la arquitectura de partes y funciones, e interrelaciones entre ellas, del producto que se implementará con el objetivo de satisfacer los requerimientos. Se determinará qué partes o componentes ya existen y cómo se reutilizarán y cuáles deberán implementarse.

En la ***implementación*** se escribirá (***programación***) el código (en este libro en el lenguaje C# y con el entorno de desarrollo Visual Studio .NET) que finalmente será "entendible" por el hardware o el framework con su máquina virtual. Proceso en el que por lo general pueden mediar diferentes capas de compilación-traducción que intentan automatizar todo lo que se pueda automatizar. Algunos autores defienden la posición de que la etapa de diseño culmine en un **modelo** y que luego herramientas conocidas como **generadores automáticos de código** transformen ese modelo a código ejecutable.

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.

Por último, tenemos la etapa del *mantenimiento,* que ha sido tal vez históricamente la más marginada y sin embargo la que más costos ha ocasionado[8]. En el software moderno que enfrenta la solución de problemas más complejos (muchos usuarios, distribuidos en diferentes lugares, en escenarios diferentes con interrelación de variados dispositivos, con grandes volúmenes de información a manejar) es prácticamente imposible que en la etapa de "prueba" se corrijan todos los errores. Muchos 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[9]. 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 el proceso de pasar por todas las etapas es en cierto modo un *ciclo* repetitivo.

En el desarrollo de software moderno los lenguajes de programación están integrados a herramientas y entornos de desarrollo[10] que ayudan en este ciclo de desarrollo 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

[8] 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.

[9] 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.

[10] Conocidas en inglés por las siglas IDE (*Integrated Development Environment*)

mismos. Estos entornos de desarrollo disponen 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.

1.4 Por qué .NET y C#

.NET Framework[11] es la plataforma de trabajo que apoyará la ejercitación práctica de los conceptos que se desarrollan en este libro. .NET Framework incluye una tal máquina virtual, conocida como *Common Language Runtime* (CLR) y también una amplia biblioteca de componentes de software denominada *Basic Common Library* (BCL)[12]. Este framework da soporte a varios compiladores de lenguajes de programación de alto nivel que serán traducidos a un lenguaje intermedio conocido como IL[13]. El CLR traduce luego, en el momento de ejecutar una aplicación y de forma transparente al usuario de la aplicación, las instrucciones en IL a instrucciones propias del CPU, utilizando una tecnología conocida por compilación JIT[14].

.NET Framework se sustenta en el **modelo orientado a objetos** e incluye lenguajes de programación como C++, Visual Basic, F# y C# que pueden interoperar y utilizarse entre sí. 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 lenguaje multiparadigma que se sustenta en el paradigma orientado a objetos pero que también tiene del paradigma **funcional** y del paradigma de **tipado dinámico**. Es por ello que es el lenguaje que se usará en este libro para presentar los conceptos de programación y las habilidades que pretenden desarrollarse.

C# es simple y a la vez altamente expresivo a la hora de implementar conceptos modernos de programación. Es un lenguaje vivo y en constante evolución, lanzado en el 2000 y en el momento de cumplir sus 15 años va por su versión 6. Desarrollado y mantenido por un equipo de Microsoft inicialmente liderado por Anders Hejlsberg, creador de los populares Turbo Pascal y Borland Delphi.

Para beneficio del ciclo de desarrollo del software, 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.

[11] Framework se traduce por *marco de trabajo*, pero a lo largo del texto mantendremos el término en inglés.

[12] Cuando comenzamos a escribir este texto, .NET Framework, el CLR, así como un compilador de C#, sólo existían para el sistema operativo Windows de Microsoft y por consiguiente para los hardware sobre los que existe este sistema operativo (la gran mayoría, por cierto). Sin embargo, en el momento de terminación del libro ya Microsoft, en colaboración con Xamarin, ha anunciado que trabaja para llevarlo a los sistemas Linux y Mac y con ello a prácticamente todo el ecosistema de dispositivos.

[13] Siglas del inglés *Intermediate Language*.

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

Como se verá en los siguientes capítulos, en el corazón de un lenguaje orientado a objetos está la forma en que éste maneja el concepto de tipo y da soporte para usar y definir tipos. Con la capacidad de definir nuevos tipos se extiende la expresividad del lenguaje para modelar mejor el problema que se está tratando de resolver. C# ofrece soporte para definir tipos mediante interfaces, clases, delegados y funcionales a la par que ofrece recursos para adaptar, ampliar y modificar los mismos.

Poder detectar estáticamente las inconsistencias en el uso de los tipos permite detectar tempranamente errores en el software, pero poder manejar dinámicamente los tipos permite adaptar el software a situaciones no previstas con antelación y trabajar con nuevas funcionalidades durante la propia ejecución de la aplicación. C# da soporte a ambos enfoques con un balance adecuado del tipado estático, la reflexión (*reflection*) y el tipado dinámico.

Más recientemente, con la apertura del propio código fuente del compilador de C# haciendo éste Open Source, a la par que ofreciendo servicios de compilación que permiten que los propios programas tengan acceso a la información que procesa el compilador, C# abre las puertas al desarrollo de mejores herramientas de ayuda y a que los propios programadores puedan mejorar su productividad y su experiencia de usuario.

Ciertamente, un lenguaje de programación no garantiza un buen desarrollo de software, hace falta de disponer de una buena herramienta de desarrollo, de la aplicación de buenas metodologías, de buenos patrones y de buenas prácticas. C# y Visual Studio dan buen soporte para lograr esto. No obstante si sus jefes, su religión o sus gustos, le llevan en su trabajo profesional a usar otros lenguajes, iniciarse en la programación, o seguir los conceptos de este texto, a través de C# le será de gran utilidad.

1.5 ¿Qué orden seguir en la lectura de este libro?

Si es usted de los que empieza, sugerimos una lectura secuencial de los temas presentados; pero si ya conoce y solo quiere consolidar o ampliar podrá saltarse o leer con más brevedad algunos temas que le sean conocidos, en particular posiblemente aquellos dedicados a los conceptos más básicos que se abordan en los primeros cinco capítulos. Si es Ud. de los que decide pasar por alto algún capítulo, de todos modos le sugerimos que cuando disponga de algo más de tiempo le dé una pasada y explore cómo sus temas han sido presentados, porque posiblemente se encuentre con un enfoque más simple, sencillo, coherente o elegante, ilustrando ejemplos o relacionando los conceptos que ya conocía con otros que tal vez no le sean tan familiares.

1.6 Sobre la notación utilizada en el libro

Los conceptos introducidos se destacarán en **negrita**. Palabras o frases que quieran destacarse se destacarán en *itálica*. Para los listados y segmentos de código se utilizará la fuente `consolas` manteniendo el coloreado que predeterminadamente usa Visual Studio. Los nombres de variables, métodos, tipos, los valores literales, que aparezcan dentro de un texto estarán en fuente `consolas` y con el coloreado que utilice Visual Studio.

Cuando no se quiera cargar el texto de un listado con todo el código, se utilizarán frases en fuente consolas y en itálica; por ejemplo:

```
if (primera vez) … abrir una conexión … .
```

De igual modo, la explicación de un concepto en la descripción de una sintaxis se describirá en consolas, itálica y entre angulares. Por ejemplo, una **instrucción de ciclo foreach** es de la forma `foreach (<Tipo> <variable> in <valor IEnumerable>) <instrucción>.`

Las notas añaden información adicional, o de curiosidad, sobre lo que se está describiendo en el texto, pero no son de imprescindible lectura para la comprensión del mismo.

Algunos términos establecidos que no gozan de una traducción consensuada al español, como los populares array (que algunos traducen por vector, arreglo) y string (cadena, tira), serán mantenidos en inglés.