Programación orientada a objetos

CD interactivo en esta edición

con C++ y Java

José Luis López Goytia
Ángel Gutiérrez González

Un acercamiento interdisciplinario

Grupo Editorial Patria

Programación orientada a objetos C++ y Java

Un acercamiento interdisciplinario

PRIMERA EDICIÓN EBOOK
MÉXICO, 2014

GRUPO EDITORIAL PATRIA

Programación orientada a objetos C++ y Java

Un acercamiento interdisciplinario

José Luis López Goytia
Ángel Gutiérrez González

Instituto Politécnico Nacional

Para establecer comunicación con nosotros puede hacerlo por:

correo:
Renacimiento 180, Col. San Juan Tlihuaca, Azcapotzalco, 02400, México, D.F.

fax pedidos:
(01 55) 5354 9109 • 5354 9102

e-mail:
info@editorialpatria.com.mx

home page:
www.editorialpatria.com.mx

Dirección editorial: Javier Enrique Callejas
Coordinadora editorial: Estela Delfín Ramírez
Supervisor de preprensa: Gerardo Briones González
Diseño de portada: Juan Bernardo Rosado Solís/Signx
Ilustraciones: Adrian Zamorategui Berber
Fotografías: © Thinkstockphoto

Revisión Técnica:
Fabiola Ocampo Botello
Instituto Politécnico Nacional-ESCOM
Roberto de Luna Caballero
Instituto Politécnico Nacional-ESCOM

Programación orientada a objetos con C++ y Java.
Un acercamiento interdiciplinario

Renacimiento 180, Colonia San Juan Tlihuaca
Azcapotzalco, México D. F.

Miembro de la Cámara Nacional de la Industrial Editorial Mexicana
Registro Núm. 43

ISBN ebook: 978-607-438-933-3

Impreso en México
Printed in Mexico

Primera edición ebook: 2014

Dedicatoria

A mi hija Ameyalli, la más tierna fabricante de lunas.

A mi esposa Claudia Aidee, por su valentía para reconstruir los sueños.

José Luis López Goytia

Tengan fe en sus propósitos y perseveren en ellos con la confianza de hacerlos realidad. Cualquier éxito súmenlo al estímulo de sus esfuerzos, cualquier fracaso anótenlo en el catálogo de sus experiencias; más nunca abandonen sus tareas ni las empobrezcan con su desaliento (I. P. p.36).

A mis amados hijos: José Ángel y Mario Ángel.

A ti, Paty, por tu inagotable paciencia y amor para mantener unida nuestra familia.

El hombre superior está centrado en la justicia, el hombre vulgar en el beneficio (Confucio).

A ti…. Grecia, como una pequeña luz que brilla en la oscuridad.

Ángel Gutiérrez González

Agradecimientos

No quisiéramos dejar pasar la oportunidad de agradecer a los estudiantes y maestros de la Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas (UPIICSA), del Instituto Politécnico Nacional (IPN), por todas las conversaciones y polémicas que hemos tenido acerca de cómo aprender y enseñar desarrollo de software. Algo vamos avanzando, pero estamos lejos de finalizar la labor.

También a Grupo Editorial Patria, por un trabajo tan amable y profesional, entre ellos al corrector de estilo y al revisor técnico, cuyos nombres desconocemos. Pero de manera muy particular a Estela Delfín, nuestra editora, siempre cortés y asertiva; un gusto tratar con personas como ella.

Introducción

¿Por qué otro libro sobre programación estructurada y programación orientada a objetos, si existen varios de excelente calidad? Todos los trabajos que hasta hoy hemos encontrado se dirigen hacia las instrucciones de código y los algoritmos: se explican las sentencias y las soluciones, para luego dar uno o varios ejemplos de uso (el inicio típico es un programa que despliega la frase "Hola, mundo"). En términos generales, este método puede formar buenos codificadores. Sin embargo, se pierden varios puntos esenciales en el camino de formar un verdadero profesional en las áreas de desarrollo de software y sistemas de información, cuyas bases se encuentran dispersas entre libros de programación, ingeniería de software, bases de datos y pedagogía, en unidades de aprendizaje y documentación que rara vez se relacionan explícitamente entre sí.

Esta situación fue detectada desde hace algún tiempo en las asignaturas de Lógica de Programación y Programación Orientada a Objetos de la Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas (UPIICSA) del Instituto Politécnico Nacional (IPN), lugar en donde hemos buscado dar un enfoque integral a la enseñanza de la programación y de donde surge la presente obra. Aunque debemos aclarar que no es un problema local: existen indicios de que es una necesidad nacional e internacional.

Por ello, este libro trata de la programación estructurada y programación orientada a objetos, pero al mismo tiempo reflexiona sobre cómo aprender y enseñar programación. Lo que intentamos es una obra que aborde aspectos básicos de programación, pero que muestre las ligas indispensables hacia la ingeniería de software y señale recomendaciones —siempre expuestas a verificación y crítica— sobre aspectos de enseñanza y aprendizaje. Seamos honestos en este sentido: hay más dudas que certezas. Debemos reconocer que aunque son opiniones con un sustento relativamente firme, pueden coincidir o no con la situación concreta de cada escuela. Nuestra pretensión no es tener una realidad universal, sino mover a la reflexión sobre la enseñanza de la programación, un aspecto por lo común descuidado.

La programación estructurada abarca los capítulos 1 a 3 con base en el lenguaje de programación C. En ellos se establece una metodología de trabajo básica para programación estructurada a fin de que el estudiante codifique de manera disciplinada. La adaptación del método de cascada que proponemos tiene como aspectos principales:

1. Partir de una petición explícita de un usuario final, la cual incluye un ejemplo de cálculo en caso necesario. Esto evita que el estudiante resuelva ambigüedades en los requerimientos. Desde nuestro punto de vista, este es uno de los "pecados capitales" que se suelen cometer en la enseñanza cotidiana de los cursos de programación: el estudiante suele ser su propio usuario, "suplantando" al usuario final, costumbre que suele llevar después al campo profesional.

2. Diseñar un lote de pruebas antes de elaborar el algoritmo, el cual debe considerar tanto las situaciones normales como los casos excepcionales. En este punto se reitera la importancia de las pruebas de escritorio antes de llevar el programa a la codificación y al equipo de cómputo. No es trivial el asunto: la mayoría de los estudiantes reprueba un examen de prueba de escritorio en la primera oportunidad.
3. Conformar la documentación "sobre la marcha", la cual debe entregarse de manera completa: desde el requerimiento hasta la pantalla final. De esta manera se pretende disminuir, aunque sea de manera muy modesta, uno de los problemas más fuertes en el área de sistemas de información: la falta de una documentación completa y actualizada que facilite la operación y el mantenimiento del software.

También es conveniente señalar las mejoras que necesitaría el programa para "crecer" hacia una aplicación profesional, acercándose intuitivamente a temas como portabilidad del código, robustez, eficiencia y facilidad de mantenimiento.

Con este propósito se proporcionan ejemplos diversos y programas completos para cada tema, además de actividades de interpretación de código o seudocódigo para que el estudiante practique las pruebas de escritorio. Una novedad más: a lo largo del libro hay ejercicios de "clínica de programación", en los cuales se parte de la mayor parte del programa para que el alumno lo complete o corrija. Esto tiene la intención de acostumbrarlo a trabajar con la lógica y el estilo de otro programador, situación por demás común en el mercado laboral cuando se le da mantenimiento a sistemas elaborados por otro desarrollador (la refactorización de código).

En los capítulos 4 a 6 se pretende abordar los aspectos principales de la programación orientada a objetos (POO) a través de C++ y Java. Nos atrevimos a entremezclar la parte conceptual —la misma en términos generales—, con programas mostrados en ambos lenguajes.

Aquí se propone que el estudiante perciba que el requerimiento en POO se refiere a la creación de componentes de software (clases) que deben ser probadas de manera independiente (a través de su TestClase) para que se integren en forma posterior al sistema integral. Hacemos "focos" con su respectivo "probador de focos"; la instalación eléctrica completa y la casa vendrán luego. Los diagramas de clase sustituyen en gran parte al seudocódigo o los diagramas de flujo de la programación estructurada.

Por último, en el capítulo 7 se abordan aspectos básicos sobre el desarrollo de software desde el punto de vista metodológico, ejemplificando con casos prácticos que vivimos en nuestra experiencia profesional (como actores o como testigos cercanos). La intención es que el estudiante pueda relacionar de manera natural la ingeniería de software con la programación: no son mundos separados.

Conviene aclarar que este libro puede ser empleado por dos tipos de estudiantes: en primer lugar, quienes comienzan su carrera en informática o disciplinas afines y en un futuro se harán profesionales del área. En segundo término, puede servir para alumnos de otras carreras que llevan un curso introductorio de programación y desean realizar después trabajos interdisciplinarios con el área de desarrollo de sistemas. En cualquier caso, parte del supuesto de que no tienen antecedentes en el área de programación.

Por otra parte, esta obra no está dirigida únicamente a estudiantes; pretendemos que sea un material interesante para egresados y docentes, quienes sin duda encontrarán afirmaciones coincidentes o polémicas que los retarán a reflexionar sobre su campo profesional. Porque es necesario hacer hincapié en que difícilmente se puede fomentar una forma de trabajo disciplinada si el propio docente y el egresado no están convencidos de ello. Digámoslo de manera directa: la mayoría de las ambigüedades de la especificación de requerimientos proviene de la forma en que el profesor o el profesional plantean el problema. Nos hemos topado con tareas de programación que pueden entenderse de cuatro o cinco formas distintas, y profesores y egresados a quienes esta situación se les hace natural. Como maestros, a veces no solemos dar la importancia que se merece a los aspectos metodológicos y esto parece ser una situación a nivel nacional e incluso internacional.

El análisis de las formas más adecuadas para enseñar desarrollo de software es un área que se ha abordado muy poco. En muchos países no existen diplomados ni posgrados sobre la enseñanza de la programación, como en matemáticas, química y física. Por ello, a lo largo del libro se irán expresando "puntos de reflexión" en torno a mejoras didácticas en la enseñanza de la programación, que parten del supuesto de una participación activa de estudiantes y profesores dentro de una experimentación permanente en el aula. Desde luego, estas propuestas deben ser corroboradas en el propio contexto de la cotidianidad, pues la realidad siempre tiene la última palabra.

REFLEXIÓN

Se utilizarán cuadros como este para expresar "puntos de reflexión" sobre la enseñanza del desarrollo de software.

Por otra parte, hemos optado por una estrategia algo "agresiva" para el proceso de enseñanza-aprendizaje. Un contenido esencial que hace hincapié en los aspectos más importantes de cada tema, combinado con reflexiones y ejercicios sugeridos. No aspiramos a ser un manual de referencia, pues hubiéramos necesitado sextuplicar el contenido; por ello muchos temas pueden complementarse con un banco de información innovador de acceso gratuito que hemos construido con base en enlaces de información, al cual puede accederse a través de www.megasinapsis.com.mx.

Esta forma de escribir un libro lleva un concepto implícito: el conocimiento es único, pero se fue dividiendo en especialidades durante los siglos XIX y XX. Esta visión ayudó en muchos casos a éxitos rotundos de la humanidad, como el caso de la creación de esquemas de vacunación; no obstante, además del conocimiento especializado en una rama del conocimiento, se requiere un trabajo interdisciplinario para abordar problemas en donde intervienen factores múltiples, para los cuales es indispensable un enfoque holístico. Hasta hoy no hemos hallado un libro sobre programación estructurada y programación orientada a objetos que hubiera sido trabajado con esta perspectiva; de allí el atrevimiento de intentar acercarnos a este objetivo.

Una bella definición de ciencia es que es la búsqueda de conocimientos confiables y verificables acerca de la realidad, incluidos nosotros mismos. Y debemos admitirlo: la enseñanza del desarrollo de software tiene claroscuros que deben ser analizados. Esperamos que este libro contribuya, aunque sea en forma modesta, a llenar varios vacíos.

Contenido

Capítulo 1

Los primeros elementos para comenzar a programar en C

1.1 Una metodología para programación estructurada

Quizá la primera forma en que se desarrolla un programa es tener una idea general de lo que debe hacer y luego programarlo. Cuando al fin hace lo que deseamos —y después de unas pruebas muy generales— se presenta al profesor, quien detecta errores y nos pide modificaciones. A la desvelada de sábado y domingo se añade la del miércoles para, al fin, presentarlo en forma correcta (véase figura 1.1).

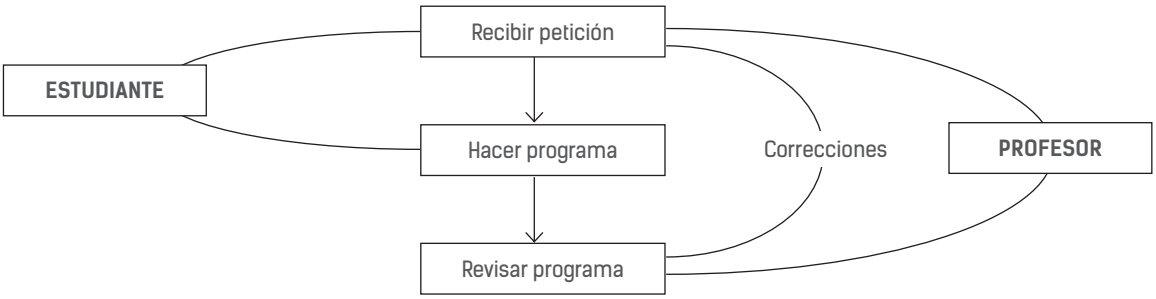


Figura 1.1 La programación sin aplicar metodologías de desarrollo.

Ahora cambiemos un poco los sustantivos y observemos el resultado.

Quizá la primera forma en que se desarrolla un sistema es tener una idea muy general de lo que debe hacer y luego desarrollarlo. Cuando al fin hace lo que deseamos —y después de unas pruebas muy generales— se presenta a un directivo de la empresa, quien detecta errores y nos pide modificaciones. A los seis meses de trabajo se añaden otros tres, para al fin presentarlo en forma correcta.

En definitiva, trabajar sin una metodología adecuada difícilmente puede llevar a buen término un desarrollo de software. Solo una labor realizada con calidad en cada uno de sus pasos y con base en una metodología acorde al proyecto —y seguida en forma disciplinada— puede producir un sistema confiable y correcto. En caso contrario, las quejas de los usuarios serán frecuentes y será conveniente tener siempre nuestro curriculum vitae bien preparado, por si es necesario buscar otro trabajo.

Una metodología es una guía general de trabajo que, llevada de manera disciplinada, trata de garantizar la construcción de un software de calidad. Lo que sirva para ese propósito debe fomentarse; si alguna tarea o documentación no contribuye a esa meta, debe adecuarse o eliminarse por completo.

La metodología vigente más conocida es el método de cascada, el cual consta de varios pasos que deben ejecutarse en forma secuencial (véase cuadro 1.1). Tomamos las etapas de uno de los mejores libros sobre gestión de proyectos informáticos que hemos conocido.[1]

Cuadro 1.1 Las etapas del método de cascada

Etapa	Objetivo
Análisis de requerimientos	Establecer sin ambigüedades los requerimientos del sistema a realizar.
Diseño	Para un sistema: diseño general del sistema a nivel interno, que comprende estructura de bloques y base de datos. En un curso introductorio: elaboración del algoritmo o seudocódigo.
Codificación y depuración	Realización del software en el lenguaje de programación elegido.
Pruebas	Pruebas del software para constatar que funciona en forma correcta.
Mantenimiento	Modificaciones al software cuando ya está en funcionamiento.

[1] McConnell, Steve. *Desarrollo y gestión de proyectos informáticos*. McGraw-Hill, España, 1998, pp. 158-159.

El método de cascada ha recibido cuestionamientos serios. Entre los más significativos se encuentran: la falta de una etapa previa al análisis en la cual se autoriza el proyecto y que el software se pruebe hasta el final. Sin embargo, todavía es vigente para proyectos de corta duración y cuyos requerimientos no cambian a lo largo del mismo. De hecho, todas las metodologías actuales lo retoman en gran medida, y le hacen modificaciones prácticas y conceptuales.

En nuestro caso, será nuestra base, aunque agregaremos algunos elementos de otras metodologías, como el diseño guiado por pruebas de Programación Extrema. Para metodologías de mayor alcance pueden consultarse Proceso Unificado, Scrumo Programación Extrema.

Aparte de la metodología, conviene aclarar que siempre debe comenzarse por definir los requerimientos (la salida esperada), después qué datos se requieren para llegar a ese resultado (la entrada requerida), y al final el algoritmo y el código que harán la transformación de esos datos de entrada en la información de salida (véase figura 1.2).

Figura 1.2 El proceso de elaboración de un software se da a partir de los requerimientos (salida), para después definir los datos (entrada), y por último el algoritmo y el código (proceso).

A continuación describiremos en forma breve nuestro método de trabajo y brindaremos un ejemplo sencillo (no comenzaremos con "hola mundo"). La intención es que desde el inicio se familiarice con este a fin de que lo siga desde su primer programa, sobre todo en la definición sin ambigüedades del requerimiento (véanse figura 1.3 y cuadro 1.2). Sería recomendable que reprodujera el ejemplo en el equipo de cómputo con ayuda de un profesor o un amigo.

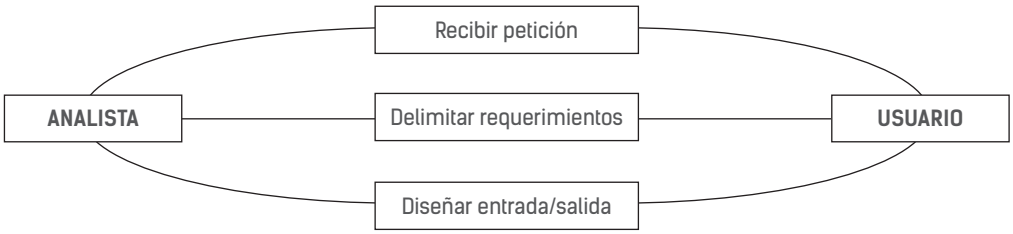


Figura 1.3 Definición de requerimientos en un curso introductorio de programación.

Al principio podrá sentirse desorientado. No se preocupe, tendrá varias formas para no perderse:

1. La descripción general que se brindará para los distintos elementos en esta introducción.
2. La explicación detallada en el momento de abordar cada tema en los siguientes capítulos.
3. La guía del profesor.

Cuadro 1.2 Adaptación del método de cascada a un curso introductorio de programación	
Análisis	Requerimiento del problema. Diseño de tabla de pruebas.
Diseño	Diseño de la pantalla con ejemplos reales. Seudocódigo o equivalente.
Codificación	Codificación en el lenguaje de programación con buenas prácticas de codificación.
Pruebas	Aplicación del lote de pruebas.
Documentación a entregar	Requerimiento, código, tabla de pruebas aplicadas y la pantalla del programa ejecutándose.

REFLEXIÓN 1.1

¿Qué estrategia utilizar en la enseñanza de la programación?

Existen distintos puntos de vista para la enseñanza de la programación. El primero propone que el estudiante trate los lenguajes de programación como conceptos algorítmicos, por lo cual sus ejemplos solo los da en seudocódigo. El segundo intenta ir al código de inmediato, y deja de lado los ejercicios centrados en el diseño de la solución.

Ambos tienen su razón de ser y sus riesgos. No puede lograrse la solución de problemas de mediana complejidad sin el planteamiento abstracto del algoritmo, expresado con algún tiempo de diagrama o seudocódigo. Por otra parte, si no se lleva la solución planteada a código, no hay aplicación concreta y no podemos estar seguros por completo de que nuestros algoritmos en verdad funcionan.

Nosotros nos inclinamos por un método mixto: llevar a los estudiantes a un primer ejemplo que involucre un pequeño código tan pronto como sea posible, a fin de que perciban mejor el contexto de trabajo. Luego, en cada rubro combinar ejercicios de codificación con otros sobre la lógica del algoritmo y pruebas de escritorio para tratar que el alumno pueda visualizar los conceptos sin importar el lenguaje en que se codifique al final.

Esta propuesta reconoce en forma implícita que deben diversificarse las estrategias de enseñanza. No existe un método único para aprender.

Paso 1. Especificación de requerimientos del usuario. En este paso el estudiante debe aclarar cualquier duda del requerimiento, incluso la realización de cálculos, la delimitación del problema, la aclaración de ambigüedades y la ubicación de casos especiales. De preferencia, deben ser explícitas las validaciones que el programa debe realizar (por ejemplo: que exista la fecha indicada) y aquellas que por ser un curso introductorio pueden omitirse (como la validación del tipo de dato al capturar).

REFLEXIÓN 1.2

Que el programador no sea su propio usuario

Una de las prácticas más comunes en los cursos de programación es que el estudiante define los requerimientos a partir de una idea general. ¡No es conveniente! Es primordial que el desarrollador tenga la perspectiva de que trabaja para un usuario final (o para un conjunto de usuarios finales en caso de un software de uso general). Lo más probable es que no sea posible recurrir a un usuario final real en un curso introductorio, pero el profesor u otro compañero puede hacer las veces de usuario final. Lo más importante es que quien codifique no defina los requerimientos y el programa terminado sea sometido a pruebas por otro estudiante o por el profesor mismo.

Paso 2. Diseño preliminar de la pantalla que se obtendrá al finalizar el programa. Esta pantalla tiene dos finalidades: verificar que no existen dudas sobre el requerimiento y tener una visión del resultado final esperado. Siempre debe hacerse con datos de ejemplo que correspondan a posibles datos reales del usuario. Los datos que provengan de operaciones deben calcularse a partir de dichos datos. Una recomendación final: utilizar una forma visual que se asemeje lo más posible a la forma en que se verá el programa terminado y que sea factible de programar.

REFLEXIÓN 1.3

Antes que nada, resolver dudas sobre el requerimiento

No deben dejarse ambigüedades en los requerimientos al empezar la parte del diseño del algoritmo, pues por lo normal eso traerá como consecuencia una gran cantidad de reprocesos innecesarios. Por ejemplo, en muchos casos el software creado nunca es utilizado por el usuario final y esto se debe en gran parte a errores al establecer los requerimientos.

A un grupo de estudiantes se les planteó: "Realice un programa que reciba los tres lados de un triángulo y despliegue su área." 45% lo hizo con (base * altura) / 2, lo cual no correspondía a la especificación solicitada. Ese porcentaje bajó a 8 en otro grupo al cual se les dio la fórmula y un ejemplo de cálculo desde el inicio.

Paso 3. Diseño del lote de pruebas que se aplicará al terminar el programa. Este lote de pruebas debe cubrir los diferentes casos típicos y especiales a los cuales se puede enfrentar el programa. Debe establecerse antes de comenzar el diseño del algoritmo.

REFLEXIÓN 1.4

¿Hasta dónde llevar las validaciones en un curso introductorio?

Las pruebas de un programa deben incluir consistencia de tipos de datos. Por ejemplo: avisar que el usuario tecleó una letra cuando se solicitó un número. En un curso introductorio como este, se puede suponer que el usuario capturará de manera adecuada el tipo de información requerida. Sin embargo, en asignaturas posteriores deberá llenarse este vacío.

A partir del tema de condicionales, conviene que el estudiante también valide la coherencia de los datos. Por ejemplo: antes de obtener el factorial de un número, que se verifique que este no sea negativo (no existen factoriales de números negativos).

Paso 4. Elaborar el algoritmo. Conviene que el estudiante elabore el bosquejo de la solución bajo alguna modalidad (seudocódigo, diagrama de flujo, etc.). El propósito es que se centre en la lógica del algoritmo y por el momento no piense en la sintaxis.

El seudocódigo se construye con las palabras elegidas por el estudiante. Por ejemplo: leer, desplegar, si, etc. No debieran establecerse reglas demasiado rígidas, aunque sí tiene que reflejar las estructuras de condicionales y ciclos que maneja la programación estructurada.

En cuanto al diagrama de flujo, este puede realizarse "a mano alzada" pero conforme a las reglas que rigen al mismo. Debe cuidarse que no consuma demasiado tiempo.

REFLEXIÓN 1.5

Sobre el seudocódigo y los diagramas de flujo

El seudocódigo es muy útil para representar al algoritmo, pues refleja con cierta libertad el lenguaje en que se codificará. Aunque debe cuidarse que no se aleje demasiado de las posibilidades de la programación estructurada, a tal punto que sea imposible codificar lo escrito. Además, verificar que sea correcto a través de las pruebas de escritorio correspondientes.

Por su parte, el diagrama de flujo sigue reglas claras y por lo mismo es más útil desde el punto de vista pedagógico. Aunque consume más tiempo hacer el diagrama de flujo que el mismo código y por ello casi no se emplea a nivel profesional.

No existe solución idónea. Cada docente debe tratar de aprovechar las ventajas de cada opción y tratar al mismo tiempo de minimizar sus inconvenientes. De cualquier manera, es conveniente que los estudiantes sepan elaborar, interpretar y verificar tanto el seudocódigo como el diagrama de flujo de un programa.

Paso 5. Elaborar el código. Se realizará la codificación si se toma como base el algoritmo diseñado y buenas prácticas de programación (alineación del código, uso de comentarios, etc.). Al terminar, se aplicará el lote de pruebas que se determinó en el paso 3. Recuerde: el programa no está terminado hasta no haber pasado todas las pruebas (y aun así, podría haber errores no descubiertos).

REFLEXIÓN 1.6

¡Desde el inicio, las buenas prácticas de programación!

El curso introductorio de programación es el comienzo para formar una arquitectura de software. Así que desde el inicio se tiene que insistir en un desarrollo de calidad, que incluya la parte metodológica (trabajar con base en requerimientos, pruebas al código, documentación, etc.) como la codificación (nombres de variables, alineación, etc.). No debemos dejar para después las buenas prácticas de programación.

Paso 6. Conjuntar la documentación. Se conjunta la documentación del programa que se entregará: requerimiento, código, tabla de pruebas y pantalla final. Conviene aclarar que algunas actividades no se reflejan en la documentación final del sistema, solo ayudan a avanzar más rápido y evitar reprocesos. En nuestro caso, el diseño preliminar de la pantalla y el diseño del algoritmo no se incorporarán a la documentación final.

REFLEXIÓN 1.7

¿Qué documentación pedir en los programas?

Una de las carencias más fuertes en el entorno docente es un análisis de la documentación a pedir en los cursos introductorios de programación. Por lo común se solicita solo el código fuente y con ello se pierde de manera implícita la liga entre los requerimientos, el código fuente y las pruebas. Desde el inicio debe acostumbrarse al estudiante a documentar en forma completa un programa: cuando menos el requerimiento, el código y las pruebas a que fue sometido.

Nuestro primer ejemplo: el promedio de dos números

Aquí va nuestro primer ejemplo. Es muy probable que sienta que es un problema relativamente artificial y que ningún usuario real solicitaría algo tan sencillo. Tiene razón, pero se comienza con él para efectos didácticos.

Paso 1. Requerimientos del usuario.
Hacer un programa que reciba dos números y despliegue su promedio a un decimal.

Paso 2. Diseño preliminar de la pantalla que se obtendrá al finalizar el programa.

```
Bienvenido.
Este programa obtiene el promedio de dos números.
Teclee sus datos separados por un espacio.
6 9
El promedio es 7.5
Oprima cualquier tecla para terminar…
```

Paso 3. Diseño del lote de pruebas que se aplicará al terminar el programa.

Datos del usuario	Resultado esperado
6 9	El promedio es 7.5
6 m	En nuestro caso, el programa "tronará". En cursos posteriores deberá validarse que el tipo de dato sea correcto.

Paso 4. Elaborar el algoritmo.

```
desplegar el texto "Bienvenido"
definir 3 variables de tipo decimal: dato1, dato2 y promedio
solicitarlos 2 datos
promedio = (dato1 + dato2) / 2
desplegar promedio
despliega "Oprima cualquier tecla para terminar…"
```

Paso 5. Elaborar el código.

```
// Programa 1.1: recibe 2 datos y devuelve su promedio.
#include <conio.h>
#include <stdio.h>
int main() {
   float dato1, dato2, promedio;
   printf("Bienvenido.\n\n");
   printf("Este programa obtiene el promedio de dos numeros.\n");[2]
   printf("Teclee sus datos separados por un espacio.\n");
   scanf("%f %f", &dato1, &dato2);
   promedio = (dato1 + dato2) / 2;
   printf("El promedio es%4.1f ", promedio);
   printf("\n\n Oprima cualquier tecla para terminar...");
   getch();
}
```

Paso 6. Conjuntar la documentación a entregar.

Requerimiento:

Hacer un programa que reciba dos números y despliegue su promedio a un decimal.

Código:

```
// Programa 1.1: recibe 2 datos y devuelve su promedio.
#include <conio.h>
#include <stdio.h>
int main() {
float dato1, dato2, promedio;
   printf("Bienvenido.\n\n");
   printf("Este programa obtiene el promedio de dos numeros.\n");
   printf("Teclee sus datos separados por un espacio.\n");
   scanf("%f %f", &dato1, &dato2);
   promedio = (dato1 + dato2) / 2;
   printf("El promedio es %4.1f ", promedio);
   printf("\n\n Oprima cualquier tecla para terminar...");
   getch();
}
```

[2] Existe un problema de compatibilidad para el despliegue de los acentos y no aparecerán correctamente en la pantalla final. Aunque puede solucionarse si se despliega el caracter ASCII correspondiente, distrae la atención para quien inicia en la programación. No existe por el momento la solución ideal.

Pruebas aplicadas:

Datos del usuario	Resultado esperado
6 9	El promedio es 7.5

Pantalla final:

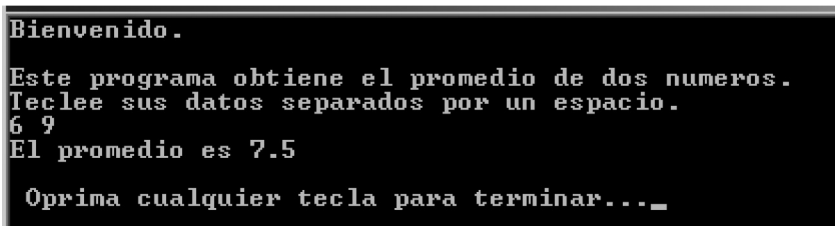


Figura 1.4

Explicación de las instrucciones empleadas en el primer ejemplo

Desmenucemos el código de nuestro primer ejemplo, con la advertencia que solo se dará una explicación introductoria de cada tema. Esperamos que sea suficiente para que el lector no se sienta perdido; conforme se avance en el libro se ahondará en cada uno de los conceptos.

```
// Programa que recibe 2 datos y devuelve su promedio.
```

Esta línea es un comentario que describe el propósito del programa. Los comentarios son palabras de orientación hacia otros programadores; no son tomados en cuenta por el compilador. Puede utilizarse el operador //al inicio de cada línea, o bien, los operadores /* y */ para iniciar y finalizar los comentarios, respectivamente, sin importar las líneas que se abarquen. Cabe aclarar que /* y */ es la forma original de especificar comentarios, mientras que // fue una aportación de C++.

```
#include <conio.h>
   #include<stdio.h
```

Indican las librerías que se utilizarán en el programa. Una librería es un conjunto de subrutinas precompiladas listas para poderse utilizar. `scanf` y `getch` se encuentran en las librerías `<conio.h>` o `<stdio.h>`, según el compilador, por lo cual se mantiene la costumbre de poner siempre ambas librerías.

```
main() {....}
```

Enmarca a todas las instrucciones del programa principal.

```
float dato1, dato2, promedio;
```

Declara tres variables de tipo flotante (decimal), llamadas dato1, dato2 y promedio, respectivamente. Las variables pueden cambiar su valor a lo largo del programa y en la mayoría de los casos se recomienda que solo almacenen datos del tipo indicado.

```
printf("Bienvenido.\n\n");
printf("El promedio es %4.1f", promedio);
```

Son instrucciones para desplegar. \n equivale a un salto de línea. `%f` señala que se desplegará el valor de una variable de tipo flotante. Al poner `.1` entre el `%` y la `f` se indica que el valor se redondeará a un decimal. El 4 significa que se reservan cuatro espacios para desplegar el valor de la variable.

```
scanf("%f %f", &dato1, &dato2);
```

Permite leer datos del teclado. `%f` señala que es información de tipo flotante (decimal). Observe que el nombre de cada variable está antecedido por un `&`.

```
promedio = (dato1 + dato2) / 2;
```

Realiza la operación mostrada y el resultado lo almacena en la variable promedio. Los paréntesis son indispensables, como en álgebra, aunque en programación no se utilizan los corchetes para este tipo de instrucciones.

```
getch();
```

Permite detener la pantalla hasta que se oprima cualquier tecla. Si no se pone en el programa, la pantalla desaparecerá en Dev C++ tan rápido que no se podrá ver el resultado, aunque debemos aclarar que no es la única forma de solucionar este problema. En ZinjaI esta instrucción no es necesaria, ni en Code::-Blocks.

Cabe aclarar que técnicamente la instrucción `getch()` permite obtener un carácter tecleado por el usuario sin que este se muestre en la pantalla, por eso es especialmente útil para programar juegos. Nosotros aprovechamos sus características para solucionar un problema que es muy común para quienes inician en lenguaje C.

EJERCICIOS SUGERIDOS

Los primeros ejercicios propuestos

He aquí algunos ejercicios sugeridos, que deben realizarse con un proceso de entrada, cálculo y salida sencillos (sin condicionales ni ciclos). Lo más importante: debe seguirse el método de trabajo sugerido.

- Realice un programa que reciba el lado de un cuadrado y calcule su área.
- Realice un programa que reciba el diámetro de un círculo y calcule su área.
- Realice un programa que reciba un número y despliegue su raíz cuadrada.
- Una tienda departamental promueve descuento sobre descuento en algunas temporadas. Calcula el primer precio y, al monto con ese descuento, le aplica un segundo descuento. Por ejemplo: si a una prenda cuyo costo original es de 100 pesos le colocara un 40% más 20% adicional, su precio final sería 48 pesos. Realice un programa que permita ingresar el monto original y ambos descuentos, y como resultado arroje el precio final.

1.2 ¿Qué hace un compilador?

Un programa (código fuente) está escrito en un lenguaje de programación. Dicho código fuente se traduce a un lenguaje binario entendible para la computadora (código máquina) a través de un software de propósito específico llamado compilador. En la gran mayoría de ocasiones se enlazan fragmentos de código (subrutinas) que resultan apropiados para nuestro programa y fueron previamente escritos y compilados por otros programadores (véase figura 1.5). Todo programa debe ser compilado antes de poder ejecutarse.

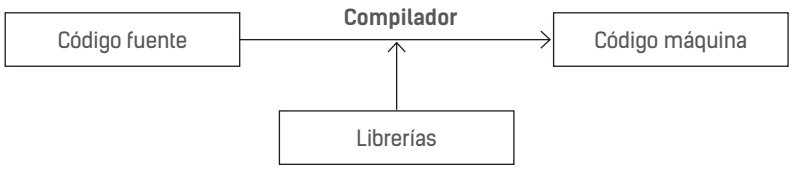


Figura 1.5 El propósito de un compilador.

Algunos lenguajes de programación se manejan a través de intérpretes, los cuales traducen y ejecutan línea por línea el programa (por ejemplo, JavaScript). En el caso del lenguaje C se traduce el código completo y, en caso de que no haya error, se ejecuta completo.

Un programa en código máquina ya puede ser ejecutado por el tipo de computadora y entorno para el cual fue compilado. No correrá fuera de ese entorno. Por citar un ejemplo: un programa ejecutable para Windows no correrá en computadoras Apple. En muchas ocasiones, el software no trabaja en versiones diferentes del mismo sistema operativo (es posible que un código para Windows '95 no trabajará en Windows XP o versiones posteriores).

Algo similar puede ocurrir con las versiones del compilador: pueden existir pequeñas diferencias entre las versiones de un compilador y, en consecuencia, un programa que trabaja de manera impecable en una versión puede marcar pequeños errores en otra (aunque por lo normal se corrigen rápido). Como es lógico, las diferencias suelen ser mayores de un compilador a otro.

Existen instrucciones que reconocerán todos los compiladores (`printf`, `scanf`, etc.). Sin embargo, habrá otras muy específicas. Por ejemplo: `gotoxy`, que solo se encuentra en el compilador Turbo C, de Borland.

¿Qué pasaría si se perdiera el programa ejecutable? Se tendría que sacar una copia de respaldo o, en su defecto, recompilar el programa a partir del código fuente. ¿Qué sucedería si se tuviera el programa en código máquina, pero no el código fuente? Se podría trabajar con él, pero no se hacerle ninguna modificación. Por lo regular, los programas que se venden de manera comercial no entregan el código fuente (Windows, videojuegos, etc.), aunque algunos que llevan la tendencia del software libre sí permiten descargarlo.[3]

¿Y si tenemos el código fuente, pero no estamos seguros de que es la última versión? Sería un riesgo realizar cualquier modificación al sistema porque el resultado sería impredecible. Moraleja: guarde el código fuente como suele hacerse con las garantías de los artículos electrodomésticos. Al paso de los meses o de los años se llegan a necesitar.

Para lograr sus tareas el compilador realiza algunas actividades:[4]

- Revisa que no se introduzcan elementos desconocidos en el programa (análisis léxico). Por ejemplo: variables que no fueron declaradas con anterioridad.

  Observe que `dato1`, `dato2` y `promedio` de nuestro primer ejemplo son definidas como variables de tipo flotante (decimal) antes de ser utilizadas en cualquier instrucción, como `scanf` o `printf`. En el lenguaje C no pueden utilizarse elementos que no hayan sido especificados antes. Por lógica, si cometemos un error "de dedo" en la declaración o al usarlas, el compilador lo marcará como un elemento desconocido.

  Las reglas sintácticas principales se mencionarán en temas posteriores.
- Revisa que las sentencias escritas estén construidas conforme a las reglas establecidas por el compilador (análisis sintáctico).

[3] El compilador Dev C++ permite la descarga del código fuente, pero su análisis queda fuera del alcance de este libro. Se deja ese ejercicio para otras materias, como compiladores.

[4] Quien desee adentrarse en el proceso puede revisar capítulos introductorios en libros dedicados al tema de compiladores.

Todas las instrucciones tienen que obedecer reglas sintácticas. Si estas no se siguen, el programa marcará un error. Por ejemplo, la multiplicación se identifica con un asterisco (*), el cual no puede omitirse como en algunas expresiones matemáticas.

- Convierte el código fuente a lenguaje máquina, al enlazarlo antes con las rutinas que fueron elaboradas por otros programadores y están contenidas en las librerías correspondientes.

  Cada librería reúne subrutinas con un propósito en común; cada subrutina es un subprograma que realiza una función específica, puede o no recibir datos desde fuera de la rutina (llamados parámetros) y devolver o no algún dato.

  Para usar las subrutinas se le debe avisar al compilador a través de la palabra `include`, al inicio del programa. Algunas subrutinas se encuentran en diferentes librerías, según el compilador y la versión. Por lo común, `scanf` y `printf` están en la librería `stdio.h`, mientras que `getch` se halla en `conio.h`. De ahí el comienzo de todos los programas del libro:

```
#include <stdio.h>
#include <conio.h>
```

Cabe aclarar que el orden es indistinto.

Existen rutinas que solo obedecen a determinados compiladores. Cabe destacar `gotoxy` y `clrscr`, que funcionan para Turbo C, de Borland, y no están incluidos en Dev C++.

A lo largo del libro se mencionarán las librerías que correspondan a los temas tratados.

REFLEXIÓN 1.8

La frecuente confusión entre los compiladores de C y C++

Los compiladores de C++ están preparados para programación orientada a objetos, aunque pueden ejecutar los códigos del lenguaje C. Las instrucciones `cin` y `cout` solo las tomará un compilador de C++. En nuestros cursos, observamos a muchos estudiantes que creen que por emplear `cin` y `cout` mientras trabajan, utilizan programación orientada a objetos. Si se habla de manera estricta, se utiliza programación orientada a objetos al emplear clases, encapsulamiento, herencia, polimorfismo o sobrecarga.

Los posibles compiladores a utilizar

Existen varios compiladores aplicables a un curso introductorio de lenguaje C, estos son: Dev C++, ZinjaI, Turbo C y Code::Blocks (este último puede descargarse gratuitamente de http://www.codeblocks.org). Es conveniente destacar que también existen compiladores para C compatibles con la plataforma Android.

Dev C++ es gratuito y de fácil manejo. Este software puede descargarse del siguiente sitio web http://www.bloodshed.net/dev/devcpp.html. En mayo de 2013 se encontraba disponible la versión 4.9.9.2 en versión beta.

Para observar los resultados de un programa debe compilarse (`Compile`) y después ejecutarse (`Run`) desde la opción `Execute` del propio compilador, como se muestra en la figura 1.6. En este curso no es recomendable compilar proyectos completos, salvo en los proyectos finales.

Dev C++ es un compilador que trabaja de manera correcta, aunque en ocasiones presenta problemas de portabilidad con versiones recientes de Windows. Algunas son muy difíciles de resolver.

Otro compilador aplicable a nuestro curso es **ZinjaI**, que cuenta con una producción automática de diagramas de flujo y alineación del código. Más compatible con las últimas versiones de Windows y más atractivo en lo visual, pero menos eficiente en el tiempo de compilación. En mayo de 2013 la última versión estable anunciada era al 30 de diciembre de 2012.

Al igual que en Dev C++, se sugiere no trabajar al inicio con proyectos, sino con archivos independientes (véase figura 1.7).

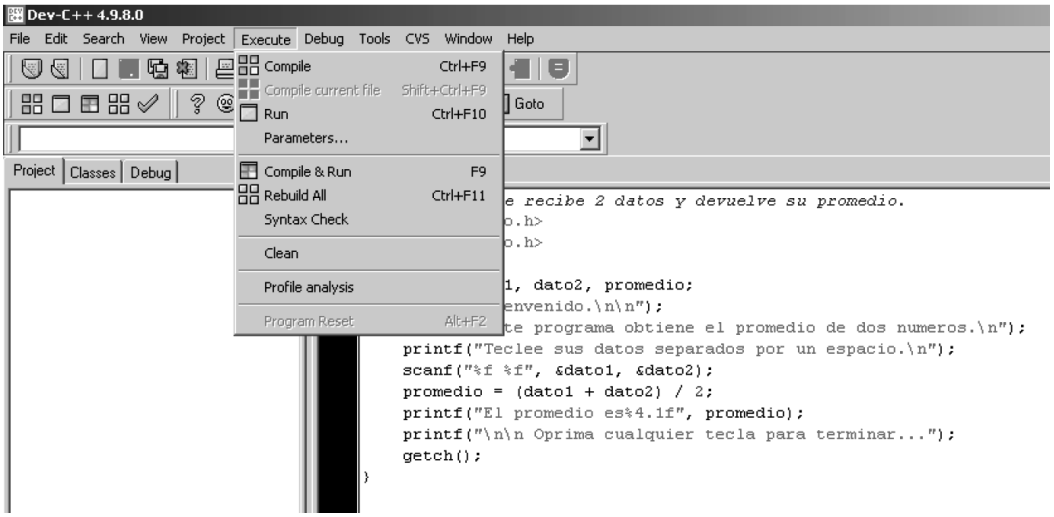


Figura 1.6 Compilación y ejecución en Dev C++.

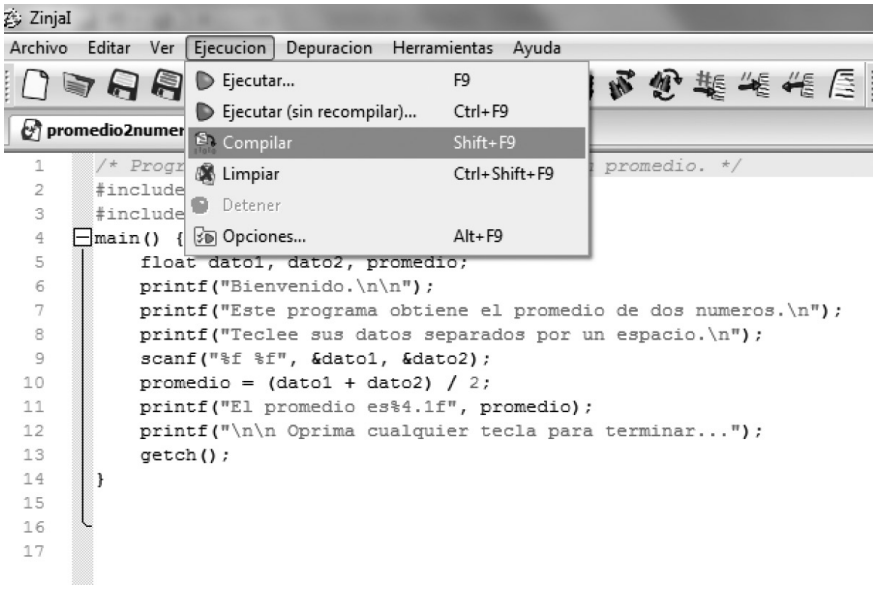


Figura 1.7 Compilación y ejecución en Zinjal.

1.3 ¿Por qué el lenguaje C para un primer curso de programación?

Existe discrepancia acerca del lenguaje más adecuado para empezar a programar. Las propuestas abarcan C, Pascal y Java, entre otras. La decisión se vuelve más difícil porque no es predecible hacia dónde se dirigen los lenguajes de programación (véase figura 1.8). Java y C conservan el más alto nivel de penetración, aunque Java disminuyó su participación. Por su parte, Microsoft —con C# y Visual Basic—, así como PHP para ambientes web conservan un nicho significativo. Conviene aclarar que SQL predomina en el campo de las bases de datos.

Pascal tiene su ventaja en ser un lenguaje muy enfocado a los cimientos pedagógicos, aunque casi no se utiliza en el campo profesional. Java es muy utilizado, pero su orientación a objetos se explotaría muy poco en un curso introductorio. El lenguaje C tiene un sitio envidiable en el mercado, en particular en código embebido; no obstante, su sintaxis en muchas ocasiones confunde a los estudiantes.

Nos inclinamos hacia el lenguaje C, sin dejar de reconocer que también Pascal o Java podrían ser opciones viables. Sin importar la elección, hay que hacer hincapié en la lógica de programación, la metodología de trabajo y los principios básicos de la programación estructurada. En forma curiosa, existen rubros que suelen descuidarse en muchos materiales introductorios sobre el tema de programación, que hacen hincapié solo en las características técnicas del lenguaje.

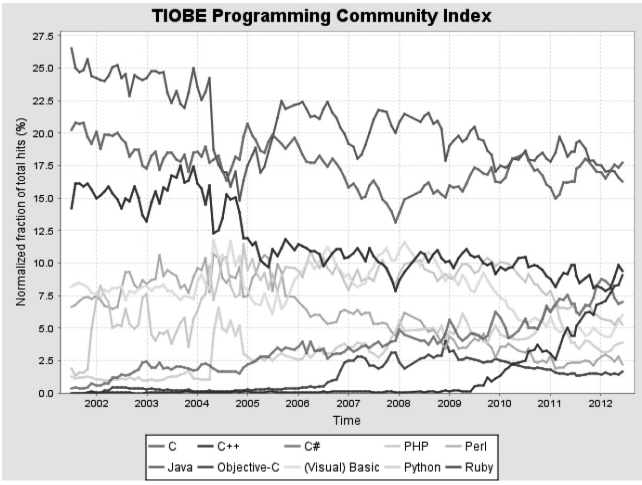


Figura 1.8 Tendencias de los lenguajes de programación.[5]

Por otra parte, no es fácil para una persona o una institución establecer un conjunto de asignaturas ligadas en forma coherente con el objetivo de aprender software a nivel profesional ni crear las condiciones adecuadas para ello, aunque ya existen planteamientos preliminares interesantes.[6] En términos generales, se pueden señalar las siguientes recomendaciones:

1. Debe ser viable en las condiciones existentes, sobre todo en lo que respecta a la infraestructura existente y el respeto a los derechos de autor. La práctica es fundamental en el curso.
2. Los problemas a resolver deben preparar al estudiante para las demandas actuales del medio ambiente laboral y de investigación (¡el alumno aprende de lo que vive!).
3. Debe integrarse en forma ordenada, coherente y natural con los demás conceptos adquiridos, pues el conocimiento es acumulativo.

Si se desea aprender programación web, una posible liga de un curso de lenguaje C con posteriores asignaturas se muestra en la figura 1.9.

Figura 1.9 Una posible liga del curso de C con otras materias.

[5] Tomado de www.tiobe.com

[6] Véase López Goytia, José Luis; Oviedo Galdeano, Mario. *Hacia un replanteamiento de la enseñanza en programación*. 3er. Congreso Internacional de Educación Media Superior 2010. Gobierno del Distrito Federal, México, 2010.

REFLEXIÓN 1.9

¿Por qué se insiste en la aplicación de una metodología?

En diferentes casos, hemos pedido a los estudiantes que dibujen la pantalla de salida a partir del requerimiento. Entre 50 y 75% lo hacen diferente a como lo habíamos imaginado. A su vez, ese porcentaje puede dividirse en dos vertientes: dan una interpretación diferente al problema porque su redacción es ambigua o no revisan en forma adecuada la redacción.

Por citar un ejemplo: se les pide recibir el diámetro de un círculo y calcular su área (área = π*radio^2). Los estudiantes toman en forma directa el dato sin considerar que le piden el diámetro al usuario y la fórmula utiliza el radio.

Las evaluaciones no deben considerar solo la codificación; deben validar también si el software se realizó conforme a las especificaciones establecidas.

1.4 Los primeros elementos de calidad del software[7]

Antes de comenzar la descripción de los elementos de un lenguaje de programación conviene preguntarse: ¿para qué deseamos realizar un software? Existen al menos dos respuestas: para solucionar un problema que un usuario ha detectado o para brindar una posibilidad tecnológica que antes no existía, aunque tal vez la mayoría de usuarios no se la ha imaginado. Este libro va enfocado de manera implícita al primer tipo de casos, como todos los materiales de cursos introductorios. Empresas como Google o Apple trabajan con el segundo enfoque.

Ahora bien, si se va a solucionar el problema de un usuario, resulta razonable que el software esté hecho con la calidad y el tiempo esperados. Por desgracia, en la mayoría de los casos no es así. Desde hace décadas se habla de la "crisis del software", referencia obligada en casi todos los cursos de ingeniería de software y tema de múltiples chascarrillos con cierta dosis de verdad. Un estudio realizado en 2001 señala que solo 28% de proyectos de software son exitosos, contra 23% cancelados y 49% excedidos en tiempo o presupuesto o con características faltante.[8] Estos datos son consistentes con nuestra experiencia docente y de consultoría (tanto en la iniciativa privada como en el gobierno). Hay que trabajar mucho en las aulas para ser parte de la solución y no parte del problema. El estudiante y el docente deben estar conscientes que el propósito no es hacer un programa conforme el maestro lo dice para obtener la calificación máxima o al menos para no reprobar; la finalidad principal es aprender a crear un software de calidad, conforme a criterios técnicos específicos.

A continuación se brinda un panorama general de los requisitos a cubrir en los programas que se elaborarán a lo largo del libro y en los que el estudiante realice en el curso.

- **Funcionalidad.** Se refiere al conjunto de situaciones que debe considerar un programa. Por ejemplo, el factorial de un número es 1 * 2 * 3 * … * *n* y no existe el factorial de un número negativo. En este caso, el programa debe considerar ambas posibilidades.
- **Corrección.** Nuestros programas deben realizar con exactitud sus tareas y sin errores, tal como se definió en las especificaciones. Un programa que no cubra lo solicitado debe rechazarse, aunque trabaje correctamente según lo entendió el estudiante. Valga un ejemplo burdo: si usted hubiera pedido una computadora portátil porque necesita llevarla diario a su trabajo, ¿aceptaría que se la cambiaran por una computadora de escritorio aunque funcionara de manera impecable?

[7] Aunque existen otros listados referentes a características, incluso más actualizados, preferimos este por su claridad conceptual y práctica. Fue adaptado de Meyer, Bertrand. *Construcción de software orientado a objetos*. Prentice-Hall, segunda edición, Madrid, 1999, 1198 páginas.

[8] Estudio de Johnson *et al*. en 2001, citado en Schach, Stephen. *Ingeniería de software clásica y orientada a objetos*. McGraw-Hill, sexta edición, México, 2006, p. 6.

- **Facilidad de uso.** Se esperan mensajes claros al usuario que lo orienten sobre la forma en que trabajará nuestra aplicación.
- **Atractivo.** El software, además de ser correcto y fácil de usar, debe ser visualmente atractivo, dentro de las posibilidades que el entorno permita.
- **Robustez.** Por los límites del curso no se pedirá que se validen los tipos de datos. Sabemos de antemano que un programa "tronará" si se pide un entero y el usuario teclea una letra. Pero sí deben validarse situaciones previsibles en los cálculos. Por ejemplo, que no se intente sacar la raíz cuadrada de un número negativo.
- **Eficiencia.** A lo largo del curso se mencionará muy poco este tema. Sin embargo, cuando se hable de ordenamientos se notará la enorme diferencia en el consumo de tiempo entre dos o más algoritmos para resolver el mismo problema.
- **Portabilidad.** Nuestros programas deben ejecutarse sin modificaciones al menos entre Dev C++ y ZinjaI.
- **Oportunidad.** No se refleja en el texto, pero esperamos que el docente exija la entrega de programas en las fechas estipuladas.
- **Facilidad de mantenimiento.** Se manejarán nombres nemotécnicos para las variables, alineación del código y comentarios para que los programas puedan ser modificados con relativa facilidad por otros estudiantes. Además, se trabajará sobre ejercicios de "clínica de programación", donde los estudiantes deben modificar o corregir el código elaborado con anterioridad.

  Desde un primer curso se debe avanzar en el logro de esas características, y expresar de manera explícita los puntos que faltarían para llegar a la calidad deseada. Los siguientes pueden ser criterios generales de un curso introductorio de programación:
  - Es conveniente fomentar una gama amplia de ejercicios de diversa índole e incorporar todos aquellos propuestos por los estudiantes, aunque adaptados al avance del curso.
  - Los ejercicios deben ser acordes al avance temático y poder resolverse con los conceptos explicados hasta ese momento. Aunque puede darse el caso que los estudiantes tengan que investigar instrucciones por su cuenta.
  - Los requerimientos deben darse por escrito y resolver cualquier ambigüedad antes de pasar al diseño del algoritmo. Una persona externa o el propio docente representará el papel de usuario, al evitar que el estudiante defina los requerimientos. Desde luego, pueden aceptarse mejoras propuestas por el alumno.
  - Los programas deben realizarse conforme a las especificaciones, si se consideran los casos especiales que puedan darse, y entregarse a tiempo y debidamente documentados.
  - Las pantallas deben ser claras para el usuario y sin faltas de ortografía.
  - Ante varias posibilidades de sintaxis, se preferirá aquella aceptada por la mayoría de los compiladores, o bien, por reglas estándares.
  - La aplicación de subrutinas debe hacerse al atender a criterios técnicos de modularidad y facilidad de mantenimiento (que se comentarán en el capítulo correspondiente). Es muy interesante dividir el programa en subrutinas hechas por diferentes estudiantes y después tratar de conjuntarlas.
  - Es conveniente llevar a cabo ejercicios de mantenimiento de código, es decir, que el estudiante corrija o modifique software ya realizado o parcialmente hecho, en lugar de partir siempre "desde cero". Debemos recordar que gran parte del trabajo de desarrollo de sistemas son labores de mantenimiento.

EJERCICIOS SUGERIDOS

1. Elija dos softwares que haya utilizado. Uno que fue agradable de emplear y otro que le causó diversos problemas. Evalúe para cada uno las características de software. Utilice una escala de 0 a 10.

2. El programa para recibir las declaraciones informativas de sueldos y salarios de una dependencia gubernamental fue publicado una semana antes de la fecha límite en que los contribuyentes deben entregar la información. Al instalarlo, no corrió en todas las versiones de Windows ni con todas las resoluciones del monitor; además, si la máquina ya tenía la versión del año pasado, no actualizaba la ayuda. ¿Qué características de un software no cumple?
3. Se hizo una aplicación en un Sistema Manejador de Base de Datos que funciona en forma correcta. Sin embargo, corre con demasiada lentitud y los ingenieros de software de la empresa informaron que era muy difícil modificar el código que dejó el programador que lo realizó. ¿Qué características de software no cumplió?
4. Una institución de salud dio a conocer su sistema, el cual cubría lo necesario para dicha institución, pero descuidaba varios requerimientos útiles para la empresa. La organización aclaró que no era un error; así se había considerado desde el inicio. Con el paso del tiempo, varias empresas se dieron cuenta que podían modificar la información sin necesidad de conocer la clave de acceso. ¿Qué características de software no cumplió?
5. Indique en cada caso la característica de software que no se cumple:

Situación	Característica
Un software se publica una semana antes, lo que obliga a un millón de empresas a instalarlo y aprenderlo a "marchas forzadas".	
No corre en las diversas versiones del navegador.	
No corre en forma simultánea con el antivirus.	
Se generó un error al octavo mes y no lo soluciona el proveedor.	
Se puede tener acceso a los datos por fuera del sistema.	
Trabaja bien pero tiene opciones muy limitadas.	
Trabaja bien en general, pero cuando se introducen datos inconsistentes (por ejemplo: una fecha ilógica) se cierra el sistema.	
Trabaja bien, pero consume mucha memoria.	
Trabaja bien, pero es muy difícil de aprender.	

1.5 Un segundo ejemplo: programa para calcular descuentos

Como base para una explicación posterior, presentamos la documentación de un programa sencillo para calcular descuentos.

Requerimiento:

Hacer un programa que reciba el precio normal de un producto y su correspondiente descuento. Como salida, que entregue el precio final (con el descuento incluido).

Código:

```
// Programa 1.2: calcula el descuento de un producto.
#include <conio.h>
#include <stdio.h>
int main() {
   float precio, descuento, preciofinal;
   printf("Bienvenido.\n\n");
   printf("Este programa le ayuda a calcular el precio ");
   printf("con descuento de un producto.\n");
```

```
    printf("Teclee el precio normal y su porcentaje de descuento.\n");
    scanf("%f %f", &precio, &descuento);
    preciofinal = (precio - precio * descuento / 100);
    printf("El precio final es %4.1f", preciofinal);
    printf("\n\n Oprima cualquier tecla para terminar...");
    getch();
}
```

Pruebas aplicadas:

Datos del usuario	Resultado esperado
100 40	El precio final es 60.0
100 0	El precio final es 100.0
100 100	El precio final es 0.0
100 110	El precio final es –10.0

Este resultado, aunque correcto desde el punto de vista aritmético, no tiene sentido en el "mundo real". En una versión posterior del programa (cuando ya se haya visto el tema de condicionales) deberá validarse que el descuento capturado esté entre 0 y 100. Pantalla final:

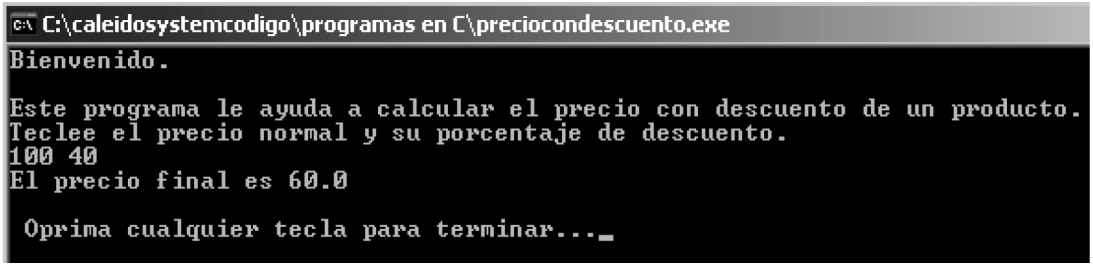


Figura 1.10

EJERCICIOS SUGERIDOS

Los ejercicios sugeridos para esta sección pueden resultar un poco extraños para estudiantes y docentes. Se trata de aplicar un par de estos con la metodología aplicada y otro par sin tomarla en cuenta (también puede dividirse el grupo en dos equipos). Después de ello, comparar el número de reprocesos que se tuvieron que hacer por no cumplir los requerimientos en forma adecuada o no considerar casos especiales.

También resulta interesante comparar el volumen de reprocesos si el ejercicio se dijo de manera verbal o por escrito. En algunos casos, las confusiones en la comunicación son asombrosas.

A manera de ejemplo, mencionamos dos posibles ejercicios:

1. Realizar un programa que calcule el área de un triángulo a partir de sus tres lados.
2. Realizar un programa que aplique descuento sobre descuento como lo hacen las tiendas departamentales en "fin de temporada". A partir del precio y los dos descuentos, indicar el precio final.

Los grupos que recibieron el requerimiento de manera verbal hacia el final de la clase tuvieron entre 28 y 45% de programas con reprocesamientos por error de requerimiento o de cálculo. Ese porcentaje disminuyó entre 8 y 15% en los equipos que siguieron la metodología propuesta.

1.6 Tipos de datos

Todas las variables y constantes de un programa deben ser de un tipo de dato específico, el cual indica en forma implícita el tipo de valores que se pueden almacenar, el espacio en memoria que ocuparán y su forma de almacenamiento, así como las operaciones que se pueden hacer con este.[9]

Los tipos de datos más usuales son: entero, decimal y carácter, que se identifican por `int`, `float` y `char`, respectivamente. En el lenguaje C no existen datos lógicos directos, aunque pueden simularse a través del tipo entero o carácter. Por otra parte, las cadenas se manejan como un arreglo de caracteres.

En lenguaje C y Java, una declaración especifica un tipo de dato y a continuación una lista de una o más variables de ese tipo. De manera opcional, a cada una de ellas puede asignársele un valor inicial. Por ejemplo:

```
float suma = 0.0, promedio;
float precio, descuento, preciofinal;
```

En algunos casos es mejor dejar una variable por renglón para colocar comentarios pertinentes.

```
float suma = 0.0; /* suma de las calificaciones */
float promedio = 0.0; /* promedio general del alumno */
```

Todas las variables deben tener un valor antes de ser empleadas. Cuando no se tiene cuidado en ello, el resultado es impredecible, como en la siguiente situación, en la cual cuentaDatos es utilizado sin que se le hubiera asignado un valor inicial.

```
intcuentaDatos;
cuentaDatos = cuentaDatos + 1;
```

En el lenguaje C existen los siguientes tipos de datos básicos:

`int`	entero normal
`float`	decimal normal
`char`	carácter de un solo byte
`double`	decimal de doble precisión

En el lenguaje C —y en todos los lenguajes de programación— cada constante y cada variable tienen un espacio físico asignado dentro de la memoria temporal con un determinado número de bytes, el cual queda distribuido en ciertos términos prefijados. Rebasa los límites del presente libro tratar la forma en que se almacenan los diferentes tipos de datos. No obstante, quien codifique debe tener en cuenta los límites de las variables manejadas a fin de utilizar el formato adecuado.

A manera de ejemplo, explicaremos lo que sucede con el tipo de datos entero (`int`). En la mayoría de los compiladores el tipo entero queda almacenado en 2 bytes o, lo que es lo mismo, 16 bits. Con 16 bits se pueden formar 2^{16} combinaciones = 65 536 combinaciones. Si esas combinaciones se dividen entre 2 quedan 32 768 negativos, el cero y 32 767 números positivos. No puede haber número entero que salga de esos límites si fue declarado de tipo `int`.[10] Aunque el lenguaje C conserva la posibilidad de utilizar modificadores para ampliar o reducir el número de bytes asignados.

[9] Existe una fuerte similitud conceptual entre los tipos de datos en programación estructurada y las clases en programación orientada a objetos (de hecho, una clase es una definición de un tipo de dato),las cuales tienen atributos (datos) y métodos (subrutinas); es decir, la información que se guarda y las operaciones que se pueden hacer con ellas.

[10] Valide si esta situación se cumple en su compilador. Puede darse el caso que el tipo int ocupe 4 bytes en lugar de 2, como en Dev C++.

REFLEXIÓN 1.10

Un nombre que no podía existir por un tipo de dato inadecuado

En los primeros años de este siglo los padres de una niña mexicana quisieron ponerle un nombre náhuatl (el dialecto más hablado en México, con antecedentes desde la época prehispánica). La respuesta de las autoridades fue que escogieran otro nombre, pues el sistema no manejaba esos caracteres (por cierto, una de tantas formas de discriminación que existen hacia los indígenas de México). Los padres, en una mezcla de valentía y perseverancia, apelaron la decisión hasta llegar a la Suprema Corte de Justicia de la Nación, quien por fin les dio la razón: debía modificarse el sistema.

Ese caso se dio en un sistema manejador de base de datos, pero suelen darse situaciones semejantes en todos los lenguajes de programación.

Hay modificadores que se aplican a todos los tipos básicos. Pueden combinarse los modificadores referentes al espacio con los referentes al signo.

`short`	disminuye el espacio asignado al tipo de dato.
`long`	aumenta el espacio asignado al tipo de dato.
`unsigned`	el tipo de dato trabajará solo con valores positivos.
`signed`	el tipo de dato trabajará con valores positivos y negativos.

Los modificadores `signed` y `unsigned` no cambian el espacio asignado. Los tipos de datos son `signed` por omisión.

Los modificadores se pueden anteponer al tipo de dato (por ejemplo: un entero corto se indicará como `shortint`). Si se utilizan para enteros, puede omitirse la palabra `int`. El tamaño en bytes para `short`, normal y `long` varía de compilador a compilador. Lo único seguro es que

```
tamaño de short <= tamaño normal <= tamaño long
```

En el cuadro 1.3 se muestra dónde se señala el número de bytes para cada tipo de dato, con base en el estándar ANSI C.[11]

Cuadro 1.3 Los tipos de datos en ANSI C

Tipo	Tamaño aproximado en bits	Rango mínimo
`char`	8	–128 a 127
`unsignedchar`	8	0 a 255
`int`	16	–32768 a 32767
`unsignedint`	16	0 a 65535
`short int`	16	Igual que int
`longint`	32	–2147483648 a 2147483647
`unsignedlongint`	32	0 a 4294967295
`float`	32	6 dígitos de precisión
`double`	64	10 dígitos de precisión
`longdouble`	128	10 dígitos de precisión

[11] Adaptado de: Schildt, Herbert. C. *Manual de bolsillo*. McGraw-Hill, segunda edición, España, 1992.

Los datos de tipo `float` y `double` se pueden manejar mediante notación científica. Es válido asignar un valor como **123.456e-7**, o bien, **0.12E3**.

"Las constantes `long` se escriben según la forma **123L**. Cualquier constante entera cuyo tamaño sea demasiado grande en una variable `int` se toma como `long`".[12]

Para especificar un valor hexadecimal se comienza con 0x; para un valor octal, con cero. Obsérvese el siguiente ejemplo:

```
intdec = 243;    /* 243 en decimal. La notación común y corriente */
inthex = 0x80;   /* 128 en decimal */
int oct = 012 ;  /* 10 en octal */
```

C no tiene algunos tipos de datos que sí existen en otros lenguajes (como datos lógicos y cadenas), pero ofrece alternativas que pueden suplir estas carencias, las cuales se explicarán en su momento.

A partir de los tipos de datos que el lenguaje proporciona se pueden hacer estructuras que los agrupan o combinan. Estas posibilidades se verán en los temas referentes a arreglos, registros (estructuras), uniones y apuntadores.

Vale la pena aclarar que el tipo `char` ocupa un único byte, que corresponde al código ASCII. Este código puede consultarse, entre otras formas, en el mapa de caracteres de Windows.

Uso de constantes

Las constantes son espacios de memoria con un nombre específico cuyo valor no cambia a lo largo de todo el programa; se definen con la directiva `#define`. Al utilizar la constante se facilita el mantenimiento y la claridad del código. Sirva este como ejemplo:

```
// Programa 1.3: ejemplificando uso de constantes (caso de PI)
#include <conio.h>
#include <stdio.h>
#define PI 3.1416
int main () {
double radio = 5;
   printf("El área de un círculo con radio 5 es: ");
   printf("%6.2f", PI * radio * radio);
   getch();
}
```

El programa de ejemplo no requirió el uso de constantes.

Reglas para los nombres de los identificadores

Toda subrutina, variable y constante debe llevar un nombre irrepetible a lo largo del programa. Dicho nombre —conocido como identificador— debe declararse antes de poder emplearse y es la forma en que el compilador ubica el elemento del cual se trata.

- Deben comenzar con una letra.
- No deben tener espacios, guiones ni caracteres especiales (ñ, acentos, etc.). Se admite el carácter de subrayado, aunque ya no es algo usual.
- Algunos lenguajes (C y Java, entre otros) hacen distinción de mayúsculas y minúsculas, mientras que otros (Basic y SQL, entre otros) no la toman en cuenta.

[12] Kernighan, Brian W.; Ritchie, Dennos M. *El lenguaje de programación* C. Prentice Hall, México, 1986.

- Deben ayudar a identificar el propósito de la variable o subrutina.
- En lenguaje C permanece la costumbre de manejar todo en minúsculas, con excepción de las constantes, que se escriben con mayúsculas. No obstante, en Java y tecnología .NET se acostumbra la "notación de camello": la primera letra de cada palabra comienza con mayúscula (excepto la primera). Por ejemplo: `promedioGeneral`.

En nuestro programa tenemos los siguientes identificadores: `precio`, `descuento` y `preciofinal`.

Conversiones de tipo

En sentido estricto, lo que indican los tipos de datos de C es el espacio que reservarán en la memoria y las operaciones aplicables a ellos, no un tipo de dato funcional. Dicho de otra forma, un `char` no es un carácter propiamente, sino un byte; puede desplegarse como un carácter o como un dato numérico. Esta posibilidad es uno de los mayores atractivos de C —y uno de los mayores dolores de cabeza. Es responsabilidad del programador indicar de manera adecuada los tipos de datos en las entradas y salidas.

El programa siguiente maneja una variable de tipo carácter y la despliega como un entero. El programa se ejecuta y muestra el valor que corresponde a ese carácter, conforme el código ASCII (para la A es 65).

```
// Programa 1.4: desplegando el valor de un caracter en ASCII
#include <stdio.h>
#include <conio.h>
int main() {
char letra;
letra = 'A';
   printf("Desplegaré la <A>: %6d", letra);
   printf("\n\nOprima cualquier tecla para continuar...");
   getch();
}
```

Los lenguajes de programación hacen conversiones automáticas de tipos de datos bajo reglas predeterminadas. Cuando se combinan diferentes tipos de datos en una expresión numérica, el de tipo "inferior" es ascendido al de tipo "superior" y el resultado se deposita en forma interna en un tipo "superior". Por ejemplo, la suma de un número entero y un número decimal se almacenará de manera automática en la memoria de la máquina como un número decimal; `char` y `short` se convierten en `int`, `int` se convierte en `long` y `float` se convierte en `double`.

A manera de ejemplo, el resultado del siguiente programa es 2.0, debido a que el resultado de 7/3 es almacenado en forma interna en una variable de tipo entero (un tipo entero entre un tipo entero se almacena en un tipo entero). Esa variable interna de tipo entero —que ya no tiene decimales— es asignada a `aux`.

```
// Programa 1.5: ejemplo de un manejo erróneo de tipos de datos
#include <stdio.h>
#include <conio.h>
int main() {
floataux = 7/3;
   printf("El resultado de 7/3 es %f", aux);
   printf("\n\n Oprima cualquier tecla para terminar...");
   getch();
}
```

La solución más sencilla es que alguno de los números se expresa como variable de tipo decimal, puesto que un número entero entre un decimal se almacenará en forma interna en una variable decimal. Es decir: `aux = 7 / 3.0`.

Claro está, esta solución no aplicaría si en lugar de expresiones indicadas de modo directo se tuvieran variables. En este caso se aplicaría una coerción de tipos, cuya sintaxis es:

```
(tipoDeDato) (expresión a convertir)
```

La línea en cuestión quedaría de la siguiente forma:

```
aux = (float) a / b; /* a y b son variables de tipo entero */
```

REFLEXIÓN 1.11

Hacia la portabilidad del código

Nunca asuma que el lenguaje hará las conversiones necesarias entre los tipos de datos manejados, incluso en los casos en que no marque ningún error. Asegúrese que así sea y —en caso necesario— hágalo de manera explícita.

Tipos de datos en otros lenguajes de programación

Los tipos de datos más comunes en los lenguajes de programación son entero, decimal, carácter, cadena de caracteres y lógico. Otros tipos de datos se incorporaron conforme surgían nuevas necesidades, sobre todo en relación con lenguajes de programación para ambientes visuales y sistemas manejadores de bases de datos: tiempo, fecha, gráfico, video y musical, entre otros.

Un lenguaje de programación o un sistema manejador de base de datos permite variantes entre los tipos de datos que afectan al rango de valores que se pueden guardar o a la precisión de estos. Por ejemplo, un entero normal en muchos compiladores abarca del –32768 al + 32767 y ocupa 2 bytes de memoria, mientras un entero corto va del –128 al + 127 y abarca un solo byte. En otros casos estas diferencias se deben a formatos manejados por diferentes proveedores o diseños más eficientes, como el caso de los distintos formatos para guardar imágenes (BMP, GIF, etcétera).

En algunos lenguajes (como C o Java) tienen que declararse antes de ser usadas; en otros, como BASIC, el compilador detecta una nueva variable y la declara por sí mismo. En apariencia, pareciera que la segunda opción es menos engorrosa, pero en realidad trae más trabajo. Imagine la siguiente instrucción:

```
numeroDeAlumnoos = numeroDeAlumnos + 1
```

Casi es seguro que el programador se refiere a la misma variable, pero la escribió mal. Si fuera BASIC, asumiría que es una nueva variable y las consecuencias serían impredecibles.

Para evitar errores como el anterior se prefiere que el compilador obligue siempre a declarar las variables e indique un error cuando se utilice una variable no declarada.

1.7 Sentencias y operadores

La rutina principal: `main`

Cabe aclarar que todo programa que se desee ejecutar de manera independiente debe tener al menos una rutina principal, que lleva el nombre de `main`.

Hay que aclarar que suele variar el requerimiento en cuanto al encabezado de esta rutina principal. Algunos compiladores piden que se indique un `int` antes de la palabra `main`; otros exigen `void`; incluso, algunos omiten cualquier palabra. `-.` En nuestro caso, elegimos `int main` porque es la alternativa que trabaja en todos los compiladores mencionados en este capítulo.

Por otra parte, si no se le pone un `getch` al final o una instrucción equivalente, en Dev C++ la pantalla "desaparece" tan pronto termina y no se ve el resultado. Sugerimos dejar la instrucción únicamente si se está trabajando en Dev C++: en los demás compiladores es innecesario.

Como se podrá observar, es una cierta molestia tratar de hacer un código portable por completo. En este libro se optó por hacerlo compatible con Dev C++ y ZinJaI (en una revisión general no se notaron variaciones con Code::Blocks). Dejamos al lector averiguar la situación a detalle de los demás compiladores.

```
int main() {
     /* Aquí va con getch y sin return
     /* Esta sintaxis se ejecuta tanto en Dev C++ como en ZinJaI */
  }
```

Todos los lenguajes de programación manejan operadores de tipo numérico, lógicos, para la conducción de cadena de caracteres y es muy posible que de otro tipo, cada uno de ellos con una función específica. Algunos son operadores nativos del lenguaje y otros forman parte de sus bibliotecas.

Entrada y salida de datos

Las entradas y salidas de datos se hacen sobre todo a través de `scanf` y `printf`, respectivamente. El formato es similar al siguiente ejemplo:

```
printf("El precio final es %4.1f", preciofinal);
```

El %f señala que ahí se mostrará un valor, el cual se indicará después del cierre de las comillas. La instrucción .1 se refiere al número de decimales y 4 a los lugares que se reservarán para el despliegue del dato (que incluyen el punto decimal y la parte fraccionaria). Se utiliza una `d` para enteros, una `c` para carácter, una `f` para tipo flotante y una `lf` para el tipo double.

Dentro del despliegue se pueden colocar opciones especiales. La más empleada es un salto de línea, que se especifica con diagonal invertida `n (\n)`, como en la línea del ejemplo.

```
printf("\n\n Oprima cualquier tecla para terminar...");
```

También se pueden especificar tabuladores `(\t)`, desplegar una diagonal `(\\)`, comillas `(\")` y apóstrofes `(\')`, entre otras posibilidades.

Por otra parte, la lectura se hace de manera similar al despliegue. Observe que en este caso el nombre de la variable va precedido por un ampersand, cuyo significado preciso se dará en la sección de subrutinas y apuntadores.

```
scanf("%f %f", &precio, &descuento);
```

Es muy común equivocarse en la sintaxis, sobre todo en el cierre de las comillas y el empleo de ampersand. Las consecuencias en estos casos son impredecibles.

Otra subrutina muy empleada es `getch()`, que recibe un carácter del teclado sin hacer eco. La forma típica de su empleo es:

```
char x; x = getch();
```

La diferencia con `scanf` es que no es necesario dar `<Intro>` para que la computadora tome esa información del teclado. La función `getch` es ideal a la hora de programar juegos.

Uso de comentarios

Los comentarios son fragmentos del código que orientan sobre aspectos que deben ser tomados en cuenta a futuro por el propio programador o por otros. También sirven para dar mayor claridad al código, orientar sobre la forma en que este enlaza con los requerimientos y brindar información del contexto de desarrollo (por ejemplo, versión del compilador empleado).

En el lenguaje C existe los comentarios que se indican con `/* y */`. La opción `//` fue incorporada en C++:

```
// a partir de aquí el resto de la línea es un comentario
/* esto es un comentario que puede abarcar varias líneas */
```

Un ejemplo de un comentario útil podría ser el siguiente:

```
/* el siguiente código responde a modificaciones en las prestaciones de la
empresa a partir del 1 de enero de 2011 */
```

Procure no emplear comentarios para situaciones que resultan obvias para quien maneja el lenguaje. El siguiente sería un comentario innecesario:

```
s = ( a + b ) / 2; // s es el promedio de a y b
```

Reglas de prioridad

Todos los operadores se rigen por **reglas de prioridad**. De hecho, por las reglas de prioridad matemáticas que conocemos desde siempre, aplicables también en álgebra y hojas de cálculo. Cabe aclarar que en programación la multiplicación se identifica con un asterisco (*).

La expresión **3 + 4 * 2** dará como resultado 11, porque por regla de prioridad se debe realizar primero la multiplicación. Cuando dos operadores tienen igual prioridad, las operaciones se ejecutan de izquierda a derecha en el caso de operadores numéricos.

En el ejemplo de este capítulo está una expresión basada en reglas de prioridad:

```
preciofinal = (precio - precio * descuento / 100);
```

El orden de evaluación es el siguiente:

- Primero se ejecuta precio * descuento.
- El resultado se divide entre 100.
- Se realiza precio menos lo obtenido de los dos incisos anteriores.
- El valor de los tres pasos se asigna a `preciofinal`.

REFLEXIÓN 1.12

Reglas de prioridad en la calculadora de Windows

La calculadora de Windows en versión científica respeta las reglas de prioridad, mientras que en modo estándar no las toma en cuenta. Considere este hecho a la hora de utilizarla para corroborar los cálculos de los programas.

La multiplicación se indica por el operador `*`, el cual es de uso obligado. La expresión 4ac o4(a)(c) no son válidas en programación. Lo correcto es: 4 * a * c.

Los paréntesis permiten agrupar operaciones que deben hacerse primero —igual que en matemáticas. En ocasiones se colocan por otros dos motivos: a) mejoran la claridad del código; b) el programador no se acordaba de las reglas de prioridad y los puso para asegurarse. Son motivos válidos. Sin embargo, no abuse del uso de paréntesis, porque su instrucción puede hacerse más difícil de entender.

El uso de espacios entre datos y operadores es optativo. El programa ejecutará en forma correcta **4*a*c** o **4 * a * c**. Recomendamos dejar un espacio entre cada operador y dato. En general, el programa se hace más sencillo de leer y la vista se cansa menos.

Al usar operadores debe tenerse cuidado con la combinación de tipos de datos. Sería ilógico aplicar una operación de multiplicación entre un carácter y un entero. La mayoría de los lenguajes marcará un error al compilar. No obstante, algunos —como el lenguaje C— sí realizarán la operación. El resultado por lo común será absurdo.

En cuanto a los operadores de condición, no existe una regla que señale el orden de evaluación y eso en ocasiones provoca errores de programación. Imagine la siguiente condición:

```
si (no se ha llegado al final del archivo) y (el empleado está activo)
```

En este caso basta con que una de las dos condiciones sea falsa para que toda la operación también lo sea. De hecho, si la primera es falsa no es posible evaluar la segunda. Pero no podemos dar por sentado que el compilador es "inteligente". En términos prácticos: no ponga dos condiciones cuando una de ellas es falsa, ya que no es posible evaluar la segunda. La solución al problema anterior consiste en separar las condiciones:

```
si (no se ha llegado al final del archivo)
   si (el empleado está activo)
```

Uno de los operadores más útiles es el de incremento y decremento (+ + y --, respectivamente). x+ + aumenta en uno la variable x y es equivalente a x = x + 1.

Los operadores de asignación ayudan a la concisión. Este tipo de operadores son más fáciles de entender si se "leen de manera coloquial". **x += 2** debe interpretarse como "súmale 2 a x". En estos casos debe tenerse presente que la asignación relaciona a la variable con toda la parte derecha de la expresión.

x *= y + 1 es equivalente a **x = x * (y + 1)** y no a **x = x * y + 1**.

C dispone de los operadores típicos de un lenguaje de programación: suma, resta, multiplicación, división, etc. Además posee dos operadores: incremento en uno (+ +) y decremento en uno (--). Estos operadores pueden emplearse como prefijos (antes de la variable) o como sufijos (después de la variable), según si su valor se incrementa antes o después de usarse. Cuando la instrucción está sola no hay diferencia, pero en combinación con otras instrucciones puede dar diferentes resultados.

Por ejemplo: suponga que *n* vale 5. ***x = n++***; pone un 5 en x, pero ***x = ++ n***; coloca un 6 en x. En ambos casos, *n* vale finalmente 6.

Lista de operadores en C

El lenguaje C tiene una gama de operadores bastante grande y proporciona una gran libertad al programador. Por lo mismo, es indispensable tener un cuidado especial con la prioridad y significado de los operadores.

La lista parcial de los operadores de C (sin contar con funciones de biblioteca) se encuentra en el cuadro 1.4.

Cuadro 1.4. Algunos operadores en ANSI C y su prioridad		
Tipo	Operador	Significado
Apuntadores	`&`	envía la dirección que apunta a una variable
Apuntadores	`*`	usa la dirección que apunta a una variable
Aritmético	`--`	decremento en 1 de una variable
Aritmético	`++`	incremento en 1 de una variable
Conversión de tipos	`(tipo)`	conversión de tipos
Espacio en bytes	`Sizeof`	cálculo del espacio necesario en bytes
lógico	`!`	negación (NOT)
Aritmético	`/`	División
Aritmético	`*`	Multiplicación
Aritmético	`%`	residuo de una división entera
Aritmético	`-`	Resta
Aritmético	`+`	Suma
De comparación	`>=`	mayor o igual que
De comparación	`>`	mayor que
De comparación	`<=`	menor o igual que
De comparación	`<`	menor que
De comparación	`!=`	¿es diferente a?
De comparación	`==`	¿es igual que?
Lógico	`&&`	y lógico (AND)
Lógico	`||`	o lógico (OR)
De asignación	`?:`	asignación condicional
De asignación	`%=`	asigna a la variable el residuo de su valor actual con la expresión indicada
De asignación	`/=`	asigna a la variable su valor actual dividido con la expresión indicada
De asignación	`*=`	asigna a la variable su valor actual multiplicado con la expresión indicada
De asignación	`-=`	asigna a la variable su valor actual restado con la expresión indicada
De asignación	`+=`	asigna a la variable su valor actual sumado con la expresión indicada
De asignación	`=`	Asignación

En su momento, se verán a detalle la mayoría de los operadores aquí expresados.

El siguiente programa ejemplifica el empleo de operadores aritméticos:

```
// Programa 1.6: ejemplos del uso de operadores
#include <conio.h>
#include <stdio.h>
int main() {
   int r = 8, s = 3;
   printf("Partimos de r = 8 y s = 5\n\n");
```

```
  r += s + 7 * 2;
  printf("Aplicamos r += s + 7 * 2. Ahora r = %d\n\n", r);
  s = r % 8;
  printf("Dividimos r entre 8. ");
  printf("El residuo lo ponemos en s. Ahora s = %d\n\n", s);
  s++;
  printf("Sumamos 1 a s. Ahora s = %d\n\n", s);
  printf("La división entera entre r y s es... %d\n\n", r/s);
  printf("Preguntamos si r >= s... %d\n", r >= s);
  printf(" (recuerde que si el resultado es verdadero");
  printf(" devuelve el valor de 1)");
  printf("\n\nOprima cualquier tecla para continuar...");
  getch();
}
```

Uso de subrutinas preprogramadas

Existen subrutinas que están en las bibliotecas del lenguaje C y que —en la práctica— actúan como si fueran operadores, entre ellas la raíz cuadrada. Para emplear estas subrutinas es necesario revisar su sintaxis y la biblioteca a la cual pertenecen. La raíz cuadrada (`sqrt`) devuelve la raíz cuadrada de un número mayor o igual que cero. Tiene la siguiente cabecera:

```
#include "math.h"
double sqrt (double num);
```

A continuación un programa que ejemplifica su uso:

```
// Programa 1.7: ejemplo de subrutinas (caso de sqrt)
#include <conio.h>
#include <stdio.h>
#include <math.h>
int main() {
  double x = 10.0;
  printf("La raíz cuadrada de 10 es %6.2f\n", sqrt(x));
  printf("\n\nOprima cualquier tecla para continuar...");
  getch();
}
```

En nuestro código de ejemplo se emplearon las siguientes subrutinas: `printf`, `scanf` y `getch`. Las dos primeras se encuentran en la librería `stdio.h`, mientras que la última se halla en `conio.h`.

Generación de números aleatorios

En muchos programas existe la necesidad de generar situaciones al azar: juegos de azar, simulación de filas en bancos, videojuegos, etc. En estos casos se suele usar la función `rand`, que devuelve un número entero de manera aleatoria. Si además se desea que en cada ejecución sea un número distinto, se combina con la subrutina `srand`. Ambas subrutinas están en la biblioteca `cstdlib`, aunque pudiera haber variaciones según el compilador.

A partir del número aleatorio proporcionado por `rand` es posible simular situaciones concretas. Por ejemplo: `srand()%2` devolvería el residuo de un número al azar entre 2, es decir, 0 o1, situación ideal para simular un volado. A manera de ejemplo, se muestra un código que simula 10 tiradas de un dado.

```
// Programa 1.8: un dado simulado con números aleatorios
#include <stdio.h>
#include <cstdlib>
#include <conio.h>
#include <time.h>
int main() {
   int dado, i;
   srand(time(NULL));
   for (i = 0; i< 10; i++){
   dado = rand()%6 + 1;
   printf("%d ", dado);
   };
   getch();
}
```

EJERCICIOS

Los primeros ejercicios de clínica de programación

Una persona transcribió los siguientes programas de una fuente fidedigna y pudo comprobar que corrían de manera adecuada: cumplen con el requerimiento y la lógica de programación es apropiada. No obstante, cometió varios errores: alguna línea omitida o la modificación inadvertida de algún signo. ¿Podría ayudarlo a que corrieran en forma adecuada? (de seguro muchos de ellos le resultarán conocidos).

Sugerencia: primero trate de ubicar las equivocaciones en papel. Luego, transcriba el código y verifique si logró ubicar todas las fallas.

REFLEXIÓN 1.13

La utilidad de la sala de código de primeros auxilios

Este tipo de ejercicios brinda la posibilidad de que los estudiantes se acerquen a los errores más frecuentes en el código, para cuando sean líderes de proyecto y tengan que ayudar a "destrabar" programas que no se ejecutan en forma correcta. Los errores pueden ser de tipo sintáctico o lógico (en este caso presentamos solo casos de fallas sintácticas).

¿Dónde obtenerlos? Directo de los casos típicos de los estudiantes.

```
// Programa que calcula el descuento de un producto.
#include <conio.h>
#include <studio.h>
main() {

   float precio, descuento, preciofinal;
   printf("Bienvenido.\n\n");
   printf("Este programa le ayuda a calcular el precio ");
   printf("con descuento de un producto.\n");
   printf("Teclee el precio normal y su porcentaje de descuento.\n");
   scanf("%f %f", precio, descuento);
   preciofinal = (precio - precio * descuento / 100);
   printf("El precio final es %4.1f", preciofinal);
   printf("\n\n Oprima cualquier tecla para terminar...");
   getch();
}
```

```
/* Programa que recibe 2 datos y devuelve su promedio.
#include <conio.h>
#include <stdio.h>
main() {
   float dato1, dato2, prommedio;
   printf("Bienvenido.\n\n");
   printf("Este programa obtiene el promedio de dos numeros.\n");
   printf("Teclee sus datos separados por un espacio.\n");
   scanf("%f %f", &dato1, &dato2);
   promedio = dato1 + dato2 / 2;
   printf("El promedio es%4.1f", promedio);
}
```

```
include <conio.h>
include <stdio.h>
Main()
{
   Printf("Hola mundo."); // y nos volvemos a ver
   Getch();
}
```

- Este programa pretende obtener el promedio de tres números. ¿Qué valor tendrá promedio al final del siguiente seudocódigo?, suponga que se ejecuta sin problemas.

```
1 define promedio como entero
2 define dato1 = 9, dato2 = 6, dato3 = 8
3 promedio = dato1 + dato2 + dato3 / 3
```

Reescriba la(s) línea(s) que están mal escritas.

# de línea	Corrección

- Indique el resultado del siguiente seudocódigo tal y como está escrito. Suponga que los valores de a, b y c son 8, 9 y 10, respectivamente

```
1 numdatos = 1
2 leer a
3 numdatos = numdatos + 1
4 leer b
5 numdatos = numdatos + 1
6 leer c
7 numdatos = numdatos + 1
8 promedio = a + b + c / numdatos
9 desplegar promedio
```

Reescriba la(s) línea(s) que están mal escritas.

# de línea	Corrección

A continuación, algunos ejercicios para practicar la codificación, **aún sin utilizar condicionales ni ciclos**.

- ¿Qué valor tendrá x después de las siguientes instrucciones (cada inciso es independiente)?

```
a)x = 8
   x = x + 2
b)  x = 8
   x = x + 2
   x = 5
   x = x * x
c)  w = 8
   t = 5
   x = w * t
d)  a = 8
   b = 10
   c = 9
   x = a + b + c / 3
e)  x = 8 * 6 + 2 * 6 / 2 + 1
```

- Crear un programa que lea una letra y despliegue aquella que se encuentra tres lugares adelante en el alfabeto (según el código ASCII).
- Elaborar un programa que lea un número entero y entregue ese número elevado a la cuarta potencia. El programa debe admitir como resultado cifras que pueden llegar hasta mil millones.
- Idear un programa que despliegue al azar un número del 1 al 6, que simule la tirada de un dado.
- Crear un programa que capture el monto a entregar por un "cajero automático", que tiene billetes de 500, 100 y 20 pesos, además de monedas de 5 y 1 (las denominaciones no cambian). El "cajero" debe determinar las cantidades a entregar de cada denominación, conforme la pantalla siguiente:

```
Bienvenido a su cajero "La amabilidad ante todo".
¿Qué cantidad desea recibir? 528
Le entregamos billetes:
1 de 500
0 de 100
1 de 20
y monedas:
1 de 5
3 de 1
¡Gracias! Oprima cualquier tecla para salir…
```

Capítulo 2

El paradigma de la programación estructurada

En el capítulo 1 se introdujo el primer ejemplo del código y la forma general de trabajo. En este capítulo abordaremos las reglas que debe seguir la codificación.

¿Por qué estamos trabajando en la forma descrita? En algunos sentidos, codificar se asemeja a jugar un deporte. Se deben conocer las reglas que rigen el juego. No obstante, eso sería insuficiente si no hubiera estrategias para ganar el partido y "jugadas prefabricadas". La sintaxis son las reglas; la lógica de programación conforma la estrategia. Por eso es primordial que el estudiante logre desarrollar y aplicar los conceptos de condicionales y ciclos, incluso la prueba del algoritmo.

Si se prosigue con esta analogía, casi nadie esperaría poder jugar bien un deporte sin practicar a diario. No obstante, un gran número de estudiantes aspira a aprender a programar estudiando días antes del examen.

Adentrémonos, pues, en la lógica de la programación.

REFLEXIÓN 2.1

La necesidad de desarrollar la abstracción en los estudiantes

La mitad del trabajo de programación se hace fuera del equipo de cómputo. Gran cantidad de algoritmos, componentes de software y arquitecturas de sistemas se expresan en diagramas, sin tener acceso al código fuente. Por ello es importante desarrollar conceptos de manera abstracta. De ahí la relevancia de la interpretación del seudocódigo y los diagramas de flujo, además de realizar pruebas de escritorio.

Pero también es necesario el saber práctico, manifestado en los programas que se ejecutan en forma directa en el equipo de cómputo.

No es clara la forma de lograr ambos propósitos, sobre todo en esta época en que los estudiantes están tan acostumbrados a sistemas interactivos, gráficos y de respuesta rápida. Estamos ante un problema generalizado que no tiene ninguna respuesta obvia ni sencilla.

El desarrollo de software tiene poco más de seis décadas, si se toma como referencia el nacimiento de la ENIAC, en 1946. Se trata de un área joven si la comparamos con ciencias como la física, la química o las matemáticas, que adquirieron madurez desde hace siglos.

En sus primeros años, la programación se basaba en instrucciones muy cercanas al código máquina y necesitaba un conocimiento del hardware sobre el cual se trabajaba. Realizar una multiplicación —o alguna instrucción de complejidad similar— requería al menos de una docena de instrucciones. Era la época del lenguaje ensamblador, que hoy se sigue empleando en microprocesadores y código embebido.

Después, se agruparon instrucciones y se inventaron programas que traducían estos comandos a lenguaje ensamblador. Nacieron de esta forma Fortran en 1957 (para aplicaciones científicas), COBOL en 1959 (para aplicaciones administrativas) y BASIC en 1964, como lenguaje de uso general.

Sin embargo, el estilo de programar en estos lenguajes presentaba un problema: el salto incondicional, representado sobre todo por la instrucción goto de BASIC, la cual permitía ir directo a cualquier línea de código. El empleo continuo del salto incondicional se convirtió en un gran dolor de cabeza para dar mantenimiento a los programas.

Para solucionar ese problema nació la programación estructurada, cuyos representantes más notorios fueron los lenguajes Pascal y C. La programación estructurada pugnaba porque todos los programas se basaran en tres estructuras básicas:

- **Secuencia.** El código se ejecutaba de arriba hacia abajo. La secuencia queda implícita a la hora de leer, compilar y ejecutar los programas.
- **Condicionales.** Elementos de decisión que permitían realizar un bloque de instrucción si se cumplía una condición y, de manera opcional, llevar a cabo otro conjunto de comandos si no se cumplía.
- **Ciclos.** Repetir una secuencia de instrucciones mientras se cumpliera una condición.

Aunque conservaban el salto incondicional hacían hincapié en que este no se empleara.

El avance natural en esta línea fue el nacimiento de la programación orientada a objetos (POO), cuyo representante más famoso es el lenguaje Java. La POO retoma los tipos de datos y las rutinas de la programación estructurada y crea con ellos una estructura de mayor potencia: las clases.

Este breve recorrido intenta contextualizar la programación estructurada, aunque resulta demasiado simplificado y deja fuera de manera injusta a lenguajes de propósito específico como el ya mencionado COBOL y a otros paradigmas, entre ellos los dedicados al campo de la inteligencia artificial. Tampoco describe lo sucedido con lenguajes para modo gráfico como Visual Basic (basado en la programación estructurada), que más tarde se transformaría en Visual .NET (basado en la programación orientada a objetos).

En cuanto a la programación web, esta se basa de manera parcial en la programación estructurada (ASP clásico, PHP, JavaScript, etc.) y en la programación orientada a objetos (J2EE, tecnología .NET, etcétera).

En síntesis, la programación estructurada sigue vigente en dos sentidos: varios lenguajes todavía la aplican y su conocimiento es base para el aprendizaje de la programación orientada a objetos.

2.1 ¿Qué es un algoritmo?

Como ya se mencionó, los ciclos y las condicionales son las estructuras básicas de la programación estructurada y sobre estas debe fincarse la creación de algoritmos básicos.

Un algoritmo es un conjunto de pasos debidamente ordenados y sin ambigüedades, que en un tiempo finito dan la solución a un problema planteado con claridad.

Un algoritmo puede plantearse de diversas formas, desde la palabra escrita hasta mecanismos formales. Entre los más conocidos están:

- **Seudocódigo.** Se escriben en el idioma nativo (el español, en nuestro caso) palabras cercanas a las instrucciones de un lenguaje de programación. Es muy recomendable como paso intermedio entre el problema y la codificación de la solución en un lenguaje de programación (Pascal, C, Java, etcétera).
- **Diagrama de flujo.** Establece de manera gráfica la solución del problema, utiliza íconos para cada estructura de programación (ciclos, decisiones, etc.). Es muy utilizado para aspectos didácticos.
- **UML (Unified Modeling Language).** Forma de diagramación utilizada para programación orientada a objetos. Su conjunto de símbolos se aplica en todas las etapas del proceso de desarrollo de software.

Elementos básicos para expresar el algoritmo en seudocódigo

La gran ventaja y, al mismo tiempo, la gran desventaja del seudocódigo es que no existen reglas formales para escribirlo. Cada persona puede establecer sus propios criterios. Por supuesto, esto puede dar lugar a ambigüedades. Para evitarlas, deberá orientarse al alumno hacia las estructuras básicas de control y establecer con claridad los pasos a seguir.

Al inicio sugerimos limitar el seudocódigo a las siguientes convenciones.

- **Variables.** Una variable es un lugar de memoria que puede guardar un valor, que puede cambiar a lo largo del programa, si considera siempre que cuando se asigna un nuevo valor, el valor anterior se pierde. Debe identificarse con un nombre determinado, el cual solo se utilizará para esa variable. Todas las variables a utilizar deben declararse al inicio del programa.
- **Nombres de identificadores.** Los identificadores son nombres que ubican a los diferentes elementos de los programas: el nombre del propio programa, las variables, etcétera.
- **Reglas de prioridad.** La ejecución de los operadores aritméticos se realiza con las mismas reglas del álgebra. Primero se harán las multiplicaciones y divisiones, y luego las sumas y restas. De esta

forma 8 + 3 * 4 será igual a 20. Se pueden utilizar paréntesis al igual que en matemáticas para forzar que en primer lugar se haga lo que esté dentro de ellos. Cabe aclarar que en programación solo se usan los paréntesis circulares.

- **Leer.** Se utilizará para recibir un dato del teclado. Por ejemplo: leer x significará que se recibirá el valor de la variable x del teclado.
- **Desplegar.** Se manejará para mostrar un mensaje textual o el valor de una variable en la pantalla.
- **Condicionales.** Indican una decisión.
- **Ciclos.** Indican que una acción se realizará mientras se cumpla la instrucción señalada.

EJEMPLO

A manera de ejemplo, suponga el ejercicio que fue presentado en el capítulo 1: Hacer un programa que reciba el precio normal de un producto y su correspondiente descuento, y como salida entregue el precio final con el descuento incluido (véase figura 2.1).

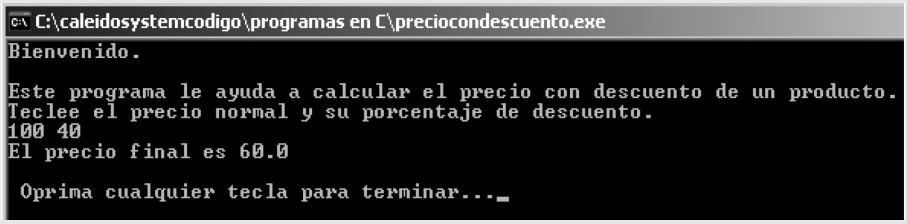


Figura 2.1 Pantalla final de un programa que aplica un descuento.

El algoritmo que lo soluciona, expresado en seudocódigo, queda como sigue:

```
declarar precio, descuento y preciofinal como decimales
desplegar "Bienvenido"
desplegar "Este programa le ayuda a calcular el precio "
desplegar " con descuento de un producto"
desplegar "Teclee el precio normal y su porcentaje de descuento"
leer precio y descuento
preciofinal = (precio - precio * descuento / 100)
desplegar "El precio final "
desplegar preciofinal
```

Elementos básicos para expresar el diagrama de flujo

Como ya se comentó, el seudocódigo no es la única forma de representar los algoritmos. Los diagramas de flujo son una forma más didáctica y muy recomendable al iniciar un curso de programación estructurada, aunque no se aplican mucho en la programación a nivel profesional por el tiempo que consumen.

En los libros de texto suele haber variaciones entre la forma de representación, aunque todos guardan el propósito general de modo coherente. Nosotros utilizaremos la versión que se muestra en la figura 2.2.

◯	Señala el inicio y el final del diagrama de flujo.	▭	Operación de asignación o cálculo.
▱	Entrada de datos del teclado.	▭	Despliegue de resultados.
→	Señala el flujo del algoritmo (el sentido de la secuenciación).	◇	Expresa una decisión a tomar según la condición expresada.

Figura 2.2 Elementos iniciales de los diagramas de flujo.

2.2 Elementos básicos de programación: condicionales y ciclos

A continuación se presentan las condicionales que son elementos básicos de la programación.

Condicionales

Las condicionales establecen una decisión con la estructura siguiente:

```
si (condición obligatoria)
   bloque 1 de instrucciones
en caso contrario (opcional)
   bloque 2 de instrucciones
termina si
```

La condición puede llevar inmersa más de una condición que al final se evalúa en su conjunto. El caso `calificación >= 6` y `calificación <= 8` dará al final un único resultado: verdadero si se cumplen ambas condiciones y falso si una o ambas no se cumplen. También existe el caso en el cual basta con que una condición sea verdadera para que la expresión en su conjunto se juzgue verdadera (por ejemplo: `calificacionparcial =10` o `calificacionfinal >=9`). En un diagrama de flujo se identificaría como lo muestra la figura 2.3.

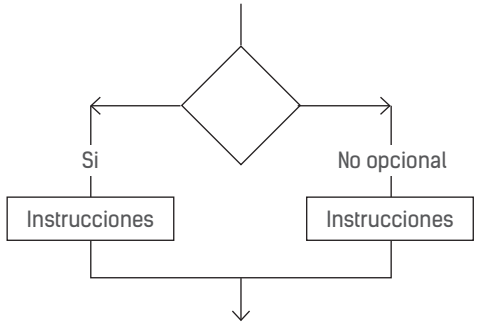


Figura 2.3 La representación de condicionales en un diagrama de flujo.

En casi todos los lenguajes de programación el "`si`" se identifica con un `if` y **"en caso contrario"** con un `else`. El `si` **(condición)** es obligatorio, mientras **en caso contrario (bloque 2 de instrucciones)** es opcional. Existen diversas formas para indicar sin ambigüedades hasta donde abarca cada bloque de instrucciones: llaves en lenguaje C, `begin-end` en Pascal, `endif` en SQL, etcétera.

A manera de ejemplo, suponga que desea recibir tres calificaciones, desplegar el promedio e indicar si la calificación es aprobatoria o no. La calificación mínima aprobatoria es 8. El algoritmo en diagrama de flujo se muestra en la figura 2.4 y a continuación se proporciona el seudocódigo.

```
declarar a, b,c y promedio como decimales
desplegar "Por favor, teclee sus tres calificaciones"
leer a, b y c
promedio = (a + b + c) / 3
desplegar "El promedio es "
desplegar promedio
si promedio >= 8
   desplegar "APROBADO"
en caso contrario
   desplegar "REPROBADO"
termina si
```

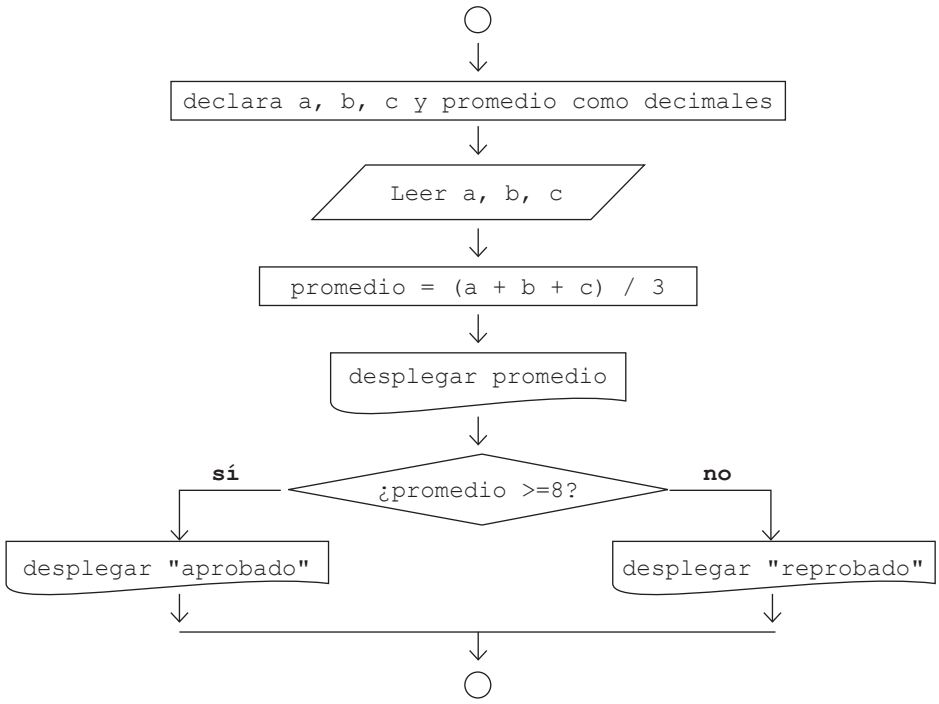


Figura 2.4 Algoritmo de prueba de 3 números empleando un diagrama de flujo.

Otros elementos básicos de la programación son los ciclos, los cuales se explican a continuación.

Ciclos

Los ciclos indican que una acción se realizará mientras se cumpla la instrucción señalada y siguen la estructura mostrada en la figura 2.5.

```
mientras (condición)
   instrucciones
termina mientras
```

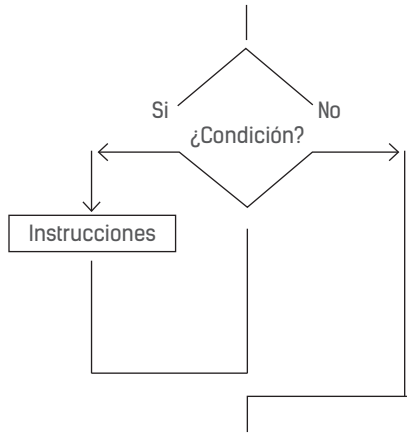


Figura 2.5 La representación de ciclos en un diagrama de flujo.

Hay que hacer hincapié en que la condición significa **mientras que**. Por tanto, una condición que siempre se cumpla generaría un ciclo infinito. Por el contrario, una condición que nunca se cumpla implicaría que no se ejecutarán las instrucciones del cuerpo del ciclo ni una sola vez.

Un error muy frecuente es no identificar con claridad aquello que es repetitivo (irá dentro del ciclo) con lo que debe hacerse una sola vez antes o después del ciclo (debe ir fuera del ciclo). Este error suele dar resultados equivocados o generar ciclos infinitos.

La condición mientras que se identifica con la palabra `while` en casi todos los lenguajes.

Al igual que en condicionales, existen diversas formas para indicar sin ambigüedades hasta dónde abarca cada bloque de instrucciones: llaves, `begin-end`, `endWhile`, etc., según el lenguaje de programación.

Aparte del ciclo mencionado existen otras variantes, cuya conceptualización se modifica un poco según el lenguaje de programación. En lenguaje C y Java por lo regular se manejan tres posibilidades: `while`, `for` y `do-while`.

El mismo ejemplo ya planteado pero con más calificaciones: se desea recibir ocho calificaciones y desplegar el promedio. Es obvio que podríamos declarar ocho variables, pero es más fácil hacerlo a través de ciclos.

El algoritmo expresado en diagrama de flujo se encuentra en la figura 2.6.

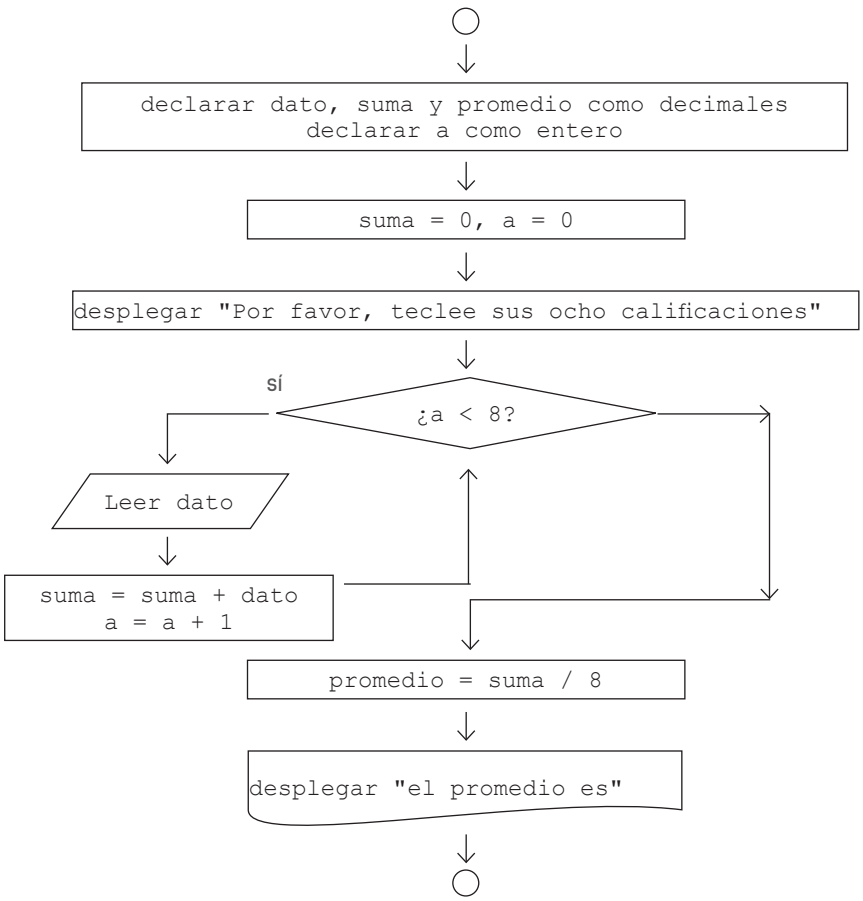


Figura 2.6 Algoritmo del promedio de ocho números en un diagrama de flujo.

```
  declarar dato, suma y promedio como decimales
  declarar i como entera
  suma = 0
  desplegar "Por favor, teclee sus ocho calificaciones"
  mientras (i < 8)
   leer dato
   suma = suma + dato
  termina mientras
  promedio = suma / 8
  desplegar "El promedio es "
  desplegar promedio
```

2.3 Prueba de escritorio

La "prueba de escritorio" consiste en tratar de seguir paso a paso lo que hace el algoritmo. En otras palabras, tratar de actuar como si uno fuera el equipo de cómputo. Hay que hacer hincapié en ello: no debe pensarse de manera "lógica" ni como "se supone que debe ser"; es menester "desprenderse de la condición humana" y "tratar de actuar como máquina", al seguir instrucción por instrucción lo que sucede en el código.

No hay una representación formal de la prueba de escritorio. Sugerimos poner una casilla por cada variable y anotar su valor. En el caso de ciclos, emplear un cuadrado y utilizar una columna por cada iteración.

Para ejemplificar, suponga que se tiene la siguiente definición del problema:

- Realizar un programa que lea un número y despliegue su factorial. El factorial de un número n está dada por la siguiente serie: 1 * 2 * 3 * ... * (n-1) * n. El factorial de 0 es 1 y no existen factoriales de números negativos.

```
Este programa obtendrá el factorial de un número.
Teclee el número, por favor
5
Su sumatoria es 120
```

El lote de pruebas es el siguiente:

Valores	Resultados	¿Pasó la prueba?
5	120	
0	1	
–3	No existen factoriales de números negativos	

El seudocódigo se describe a continuación.

```
variables a utilizar: i, suma,
despliegue "Bienvenido"
despliegue "Este programa obtendrá el factorial de un número."
despliegue "Teclee el número, por favor"
leer dato
si dato < 0
   despliega "No existen factoriales de números negativos"
en caso contrario
i = 1
   acumula = 1
```

```
   mientras i <= dato
      acumula = acumula * i
      i = i + 1
   despliegue "Su factorial es"
   despliegue acumula
despliegue "Oprima cualquier tecla para continuar…"
```

Y la prueba de escritorio se presenta a continuación:
Antes del ciclo

i	___1___
acumula	___1___
dato	___5___

Durante el ciclo

Número de repetición	1	2	3	4	5
Valor de acumula	1	2	6	24	120
Valor de i	2	3	4	5	6

Observe que se sale del ciclo cuando ya no se cumple la condición. El valor final de acumula fue 120, que era el resultado esperado.

2.4 Ejercicios de construcción de lógica a partir del problema

REFLEXIÓN 2.2

Sobre los ejercicios de construcción de lógica

Algunos docentes suelen comenzar con ejercicios de lógica del tipo: "¿cómo se hacen huevos con jamón?". Consideramos que una primera analogía de este tipo es útil para resaltar su parecido con algunos elementos de un algoritmo, sobre todo con la lógica secuencial en la resolución de un problema. Sin embargo, debemos tener cuidado en estos terrenos un tanto lejanos a la formalidad de un lenguaje de programación, en los cuales pueden darse polémicas sobre las palabras adecuadas o el nivel de detalle al cual debe llegarse.

Con las descripciones vistas hasta este punto tenemos los elementos suficientes para iniciar ejercicios de construcción de lógica.

Un primer bloque de ejercicios puede partir de una descripción general de un requerimiento que debe cumplirse a detalle. En este bloque pueden incorporarse todas las sugerencias de los estudiantes que sean viables con la aplicación de secuencias y ciclos.

La estrategia sugerida es formar equipos de trabajo conformados por un "usuario final", un "programador" y un "probador". El "usuario final" deberá recibir del profesor la especificación de los requerimientos y realizar un ejemplo de cálculo con datos reales, así como el lote de pruebas que debe pasar su sistema (puede darse la pantalla desde un inicio o que el mismo usuario la defina). El "programador" realizará el algoritmo expresado en seudocódigo o diagrama de flujo, partirá solo de los elementos vistos (se omitirá por el momento el uso de arreglos, archivos y subrutinas). Por último, el "probador" hará la prueba de escritorio. La prueba definitiva se hará en el capítulo relativo a la codificación de condicionales y ciclos.

EJERCICIOS SUGERIDOS

La siguiente es una lista de ejercicios propuestos, con los dos primeros elementos: especificación de requerimientos del usuario y diseño preliminar de la pantalla. Por razones de espacio, se omitieron en la pantalla las palabras de bienvenida y despedida, que sí deberán estar en el seudocódigo y diagrama de flujo.

- Una librería ofrece descuentos por volumen: 3% a partir de tres libros; 5% si son más de tres libros y 10% si son más de 10. Los descuentos no se acumulan. Realice un programa que reciba el costo del libro, el número de libros y despliegue el total a pagar.

```
Señale el costo unitario del libro: 120
Indique el número de libros: 3
El total a pagar es 349.20
```

- Realice un programa que calcule el promedio de varias calificaciones. Al inicio se le preguntará al usuario cuántos números capturará (suponga que el usuario lo hará de manera adecuada).

```
¿Cuántas calificaciones se van a capturar? 3
Captúrelas: 7 6 4
El promedio es 5.7
```

- Realice un programa que calcule el promedio de varios números. El usuario capturará datos y al final pondrá un –1 para señalar que ya no existen más datos (suponga que el usuario lo hará en forma adecuada).

```
Capture las calificaciones (indique un -1 para finalizar):
7 6 4 -1
El promedio es 5.7
```

- Un trabajador tiene un salario por hora (el salario diario supone 8 horas laboradas). Si trabaja más de 8 horas recibe pago por horas extras. De la novena a la undécima percibe el doble; de la décimo segunda en adelante es el triple. Las horas pueden ser fraccionadas. Realice un programa que reciba el salario diario y las horas trabajadas; a partir de esos datos señale el monto a recibir por el salario normal, las horas extras y el total.

```
Indique el sueldo diario: 200
Indique las horas trabajadas: 9.5
Sus percepciones en el día son: 275.00
```

A continuación se presentan varios ejercicios sencillos en donde los estudiantes tienen que iniciar el trabajo desde la elaboración de las pantallas.

- Capturar los tres lados de un triángulo y desplegar si es isósceles, escaleno o equilátero.
- Capturar tres números y desplegar el mayor de ellos.
- Realizar un programa que lea la altura del agua, el diámetro y la longitud de un tanque cilíndrico completamente horizontal. A partir de esos datos se desea saber el volumen de agua en litros que tiene.[1]

Enseguida se señalan varios ejercicios para realizar la prueba de escritorio. La intención es que el estudiante visualice que un problema puede solucionarse de diferentes formas y que existen errores clásicos "escondidos", sin importar el procedimiento utilizado.

- Dado el siguiente código:

[1] La solución de este ejercicio tiene que ver más con elementos básicos de trigonometría y álgebra. Y ese es, justo, el motivo de ponerlo aquí: que el estudiante viva situaciones en las que el cálculo sea lo más difícil. Dejamos al docente que valore si lo hace opcional u obligatorio.

```
mayor = b
si (b > a) y (b > c)
mayor = b
si (c > a) y (c > b)
mayor = c
despliega "El resultado es:"
despliega mayor
```

Indique el resultado de las tríadas señaladas a continuación, tal como está escrito.

Datos de prueba	Resultado
7 8 9	
8 9 8	
8 9 9	
9 8 7	

- ¿Qué mensaje desplegará el siguiente seudocódigo tal como está escrito, si supone que el usuario responde 5.5 cuando se le pide el dato?

```
x = calificación proporcionada por el usuario
si (x > 5)
   Desplegar "aprobado"
termina si
si (x < 6)
   Desplegar "reprobado"
termina si
```

- El siguiente seudocódigo está en orden alfabético. Acomódelo para obtener el promedio de tres números de manera adecuada.

```
declarar dato, suma y promedio como decimales
declarar i como entera
desplegar "El promedio es "
desplegar "Por favor, teclee sus tres calificaciones"
desplegar promedio
i = 0
i = i + 1
leer dato
mientras (i < 3)
promedio = suma / 3
suma = 0
suma = suma + dato
termina mientras
```

- ¿Qué desplegará en pantalla el siguiente seudocódigo tal como está escrito? Suponga que el usuario teclea: 5 6 7 8.

```
   suma = 0
   n = 4
   i = 1
   mientras i <= n
    despliega "Teclee los 4 datos: "
    leer dato
    suma = suma + dato
    i = i + 1
```

```
fin de mientras
despliega "El promedio es:
despliega suma / i
```

- ¿Qué valor tendrán x y w después de las siguientes instrucciones?

```
x = 1
w = x
mientras (x <=5)
 w = w * x
 x = x + 1
termina mientras
```

2.5 Condicionales: un proceso de decisión sencillo

En la figura 2.7 se presenta el diagrama básico de la decisión, empleado por primera vez en páginas anteriores.

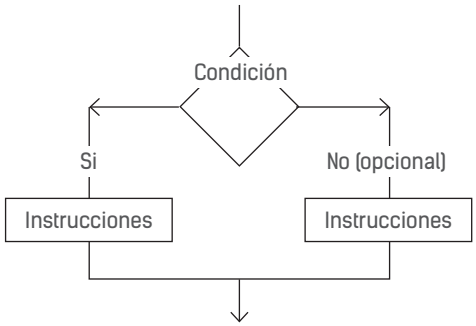


Figura 2.7 Representación de condicionales en un diagrama de flujo.

Todas las condiciones preguntan acerca de un requisito (`if`). Una respuesta afirmativa provocará que se lleve a cabo el bloque de instrucciones indicado; en caso contrario se ejecuta un segundo bloque de instrucciones (`else`). El `if` es obligatorio, mientras el `else` es opcional. Para indicar el inicio y final de cada bloque se emplean llaves, aunque pueden omitirse si el bloque está compuesto por una sola instrucción.

La condicional en seudocódigo se expresa de la siguiente forma:

```
si (condición)
   bloque 1 de instrucciones
en caso contrario
   bloque 2 de instrucciones
termina si
```

A nivel código la instrucción quedaría como sigue:

```
if (calificacion >= 6)
   printf("aprobado);
else
   printf("reprobado");
```

Desde el punto de vista técnico, una condición verdadera "devuelve" el valor de 1; una condición falsa retorna el valor de 0. Por eso, la instrucción **z = (a > b)** es perfectamente válida: **z** valdrá 1 si a es mayor que b, o valdrá 0 si la condición no se cumple.

El operador `?` es una forma abreviada de `if-else`, que se emplea cuando se trata de una asignación con base en un `if`. La instrucción:

```
z = (a > b) ? a : b;
```

es equivalente a

```
if (a > b)
z = a;
else
 z = b;
```

REFLEXIÓN 2.3

Cuidados básicos al utilizar `else`

Aunque la estructura básica de una condicional es sencilla, debe tenerse cuidado en algunos aspectos:

- La condición debe ser precisa. Por ejemplo: `if (calificacion >= 6)` es equivalente a `if (calificacion > 5)` solo si la variable calificacion es de tipo entero, pero no son equiparables si `calificacion` es de tipo decimal.
- Cuidado con la sintaxis: la condición se pone entre paréntesis y no lleva coma después de la misma.
- Debe evitarse "duplicar" la condición en lugar de emplear `else`. El siguiente código es válido:

```
if (calificacion >= 6)
 printf("aprobado);
if (calificacion < 6)
 printf("reprobado");
```

pero dificultaría el mantenimiento del código. Si la calificación mínima aprobatoria cambiara a 8, entonces habría que modificar el dato en dos lugares y podría dar pie a errores de mantenimiento en el código.

REFLEXIÓN 2.4

Utilizar condicionales para validar datos

Es muy buena idea utilizar las condicionales para validar la consistencia de la información. Por ejemplo: si se pide una fecha, verificar que no se proporcione un dato equivocado (por ejemplo, 30 de febrero).

EJEMPLO

A continuación se presenta un ejemplo para determinar el área de un triángulo (programa 2.1). Observe que la condicional se utilizó para validar que el triángulo sí pueda construirse.

Requerimiento:

Hacer un programa que reciba los tres lados de un triángulo y devuelva su área. El área queda determinada bajo la siguiente fórmula:

$$s = (a + b + c) / 2$$

$$\text{área} = \sqrt{s * (s - a) * (s - b) * (s - c)}$$, donde a, b y c son las longitudes de los lados.

Ejemplo: si los lados valieran 6, 8 y 10, s valdría 16 y el área valdría 24.

Código:

```
// Programa 2.1 recibe los 3 lados de un triángulo y devuelve su área.
#include <conio.h>
#include <stdio.h>
#include <math.h>
int main() {
   float a, b, c, s, area;
   printf("Bienvenido.\n\n");
   printf("Este programa obtiene el área de un triángulo.\n");
   printf("Teclee el valor de los lados separados por un espacio.\n\n");
   scanf("%f %f %f", &a, &b, &c);
   // previamente se validará que sea posible construir el triángulo
      if ( (a > b + c) || (b > a + c) || (c > a + b))
   printf("No puede formarse un triangulo de esas dimensiones.\n");
   else {
      s = (a + b + c ) / 2;
      area = sqrt( s * (s - a) * (s - b) * (s - c));
      printf("\n");
      printf("El area del triángulo es: %4.1f \n", area);
      }
      printf("\n\n Oprima cualquier tecla para terminar...");
      getch();
}
```

Pruebas aplicadas:

Datos del usuario	Resultado esperado
6 8 10	El área del triángulo es 24.0
3 5 9	No puede formarse un triángulo de estas dimensiones

Pantalla final:

```
Bienvenido.

Este programa obtiene el area de un triangulo.
Teclee el valor de los lados separados por un espacio.

6 8 10

El area del triangulo es: 24.0


 Oprima cualquier tecla para terminar...
```

Figura 2.8 Pantalla final de un programa que calcula el área de un triángulo a partir de sus tres lados.

Aspectos a cuidar en el uso de condicionales

- **Las condicionales pueden anidarse.** Las condicionales se pueden escribir de manera anidada, como en el programa 2.2 (hay que tener especial cuidado con la anidación):

```
// Programa 2.2:
// promedio de 3 calificaciones con condicionales anidadas
#include <conio.h>
#include <stdio.h>
```

```
int main(void)
{
   int a, b, c;
   float promedio;
   printf("Por favor, teclee sus tres calificaciones: ");
   scanf("%d %d %d", &a, &b, &c);
   promedio = (a + b + c) / 3;
   printf("El promedio es: %6.1f", promedio);
   if (promedio == 10)
    printf(" EXCELENTE");
   else
      if (promedio >= 8) {
         printf(" MUY BIEN. ");
         printf("FELICITACIONES.");
         }
         else
         printf(" REPROBADO");

   printf("\nOprime cualquier tecla..");
   getch();
}
```

- **Cuidar el alcance de los bloques.** En todo momento se debe tener presente que los bloques quedan delimitados por las llaves. Si estas no se colocan, se asume que el bloque abarca una sola instrucción. Como ejemplo, observe el código del programa 2.3.

```
// Programa 2.3: condicionales con mal manejo de llaves
#include <conio.h>
#include <stdio.h>
int main () {
   int x;
    x = 8;
   if (x >= 6)
      printf("Aprobado.\n");
   else
         printf("Reprobado.\n");
         printf("A estudiar más.\n");

   printf("\nOprime cualquier tecla para continuar...");
   getch();
}
```

La frase ***A estudiar más*** se desplegará siempre. El programa asume —a falta de llaves— que el `else` abarca solo una instrucción. La corrección sería como sigue:

```
 else {
 printf("Reprobado.\n");
 printf("A estudiar más.\n");
 }
```

- **No utilizar el operador de asignación (=) en lugar del de comparación (==).** Es muy común que en las condiciones se requiera comparar una variable o una operación con un valor determinado, por ejemplo: si x es igual a 8. En este caso hay que recordar que el operador de comparación es **==**. Observe el código del programa 2.4.

```
// Programa 2.4: condicionales con comparación mal aplicada
#include <stdio.h>
#include <conio.h>
int main () {
   int opcion;
   printf("Bienvenido al cine.\n\n");
   printf("1) Amores Perros\n");
   printf("2) Babel\n");
   printf("\nElija una opción: ");
   scanf("%d", &opcion);
   if (opcion = 1)
      printf("\nUsted desea ver Amores Perros");
   else
      printf("\nUsted desea ver Babel");

   printf("\n\nOprima cualquier tecla para continuar...");
   getch();
}
```

En este programa siempre aparecerá el letrero **Usted desea ver Amores Perros**. La razón es la siguiente: la instrucción `if (opcion = 1)` indica que se asigne un 1 a la variable opción, si esto es posible (lo cual es obvio en este caso), se desplegará **Usted desea ver Amores Perros**. La instrucción en realidad se debió manejar como `if (opcion == 1)`.[2]

Este error es tan común y tan difícil de rastrear que el lenguaje Java (basado en la sintaxis de C) ya no permite la instrucción `if (opcion = 1)`.

EJERCICIOS SUGERIDOS

Problemas propuestos para decisiones sencillas

Los siguientes problemas —extraídos de situaciones reales— pueden resolverse con decisiones sencillas. Son ejercicios que a esta altura el estudiante debe poder resolver.

- Realice un programa que lea tres calificaciones, despliegue el promedio final y diga si el alumno aprobó o no. La calificación final será el promedio ponderado de las tres calificaciones, las cuales valen 20, 30 y 50%, respectivamente. La calificación mínima aprobatoria es 8.
- Realice un programa que obtenga el promedio entero de tres calificaciones. Se redondea al entero más cercano si el promedio es cuando menos de 6. En caso contrario, la calificación bajará al entero inferior.
- Realice un programa que reciba tres calificaciones de tipo entero e indique cuál fue la calificación más alta. Por ejemplo, si el usuario captura los datos 8, 9, 10, el programa desplegará que el dato más grande es 10.
- Simule un volado en la computadora. La computadora solicita la apuesta y genera al azar la suya. Si coincide, el usuario gana; en caso contrario, pierde.

```
¿Qué crees que salga (S=sol; A=aguila)? S
La moneda cayó águila. ¡Usted perdió!
¡Gracias por participar!
```

- Realice un programa que reciba los coeficientes de una ecuación cuadrática y despliegue sus raíces.
- Realice un programa que reciba dos datos para realizar una división y despliegue el resultado a dos decimales. Por ejemplo, si el usuario captura 9 y 4, el programa indicará como resultado 4.25. Nota importante: debe validarse que el programa no intente realizar una división entre cero.
- Realice un programa que reciba los tres lados de un triángulo e indique si es equilátero (tres lados iguales), isósceles (dos lados iguales) o escaleno (tres lados diferentes).

[2] Cabe aclarar que ZinjaI hace una advertencia (*warning*) cuando se compila el código.

- Dado el siguiente código:

```
1  mayor = b;
2  if ((b > a) && (b > c))
3  mayor = b;
4  if ((c > a) && (c > b))
5  mayor = c;
6  printf("El resultado es: %d\n", mayor);
```

a) Asuma que el programa va a correr tal como está escrito. ¿Cuál sería el resultado bajo las condiciones dadas enseguida?

Datos de prueba a b c	Resultado
7 8 9	
8 9 8	
8 9 9	
9 8 7	

b) Coloque los paréntesis que hagan falta para que el programa pueda ejecutarse en forma correcta (solo hay errores en algunas líneas):

Número de línea	Corrección

2.6 Casos de decisión múltiple

Es muy común que en diversas ocasiones se tenga que seleccionar alguna opción con base en un valor determinado. Por ejemplo, desplegar el nombre del mes. Esto podría realizarse a través de condicionales múltiples, como en el programa 2.5, que despliega el número de días transcurridos del 1 de enero de 2000 a la fecha capturada.

En este caso, dejaremos que el estudiante "descifre" el programa con base en los siguientes puntos clave:

- Sumar los años completos multiplicándolos por 365.
- Sumar los días que corresponden a los meses completos del año en curso. Por ejemplo, si estamos en febrero, ya transcurrió enero en su totalidad, es decir, hay 31 días por meses completos transcurridos.
- Sumar los días que corresponden por años bisiestos: 29 de febrero de 2004, 29 de febrero de 2008, etc. Se suma un día por cada cuatro años. Cuando es un año bisiesto, se tiene cuidado en contar el día bisiesto solo en marzo o en un mes posterior.
- Sumar, por último, los días del mes en curso.

REFLEXIÓN 2.5

Empleo de Excel para corroborar un programa de fechas

Un programa de manejo de fechas puede corroborarse por medio de una hoja de cálculo. Excel almacena las fechas como los días transcurridos entre el 1 de enero de 1900 y la fecha indicada. Esta característica puede aprovecharse para verificar el programa. Si quiere comprobar su código, puede poner en una celda la fecha elegida y en otra el 1 de enero de 2000. Si se hace una resta de ambas celdas, se obtendrán —justo— los días transcurridos.

Sirva este ejemplo para una recomendación general: siempre verifique que su algoritmo dé los resultados correctos y cuando sea conveniente recurra a otras herramientas que puedan ayudarle.

```
// Programa 2.5: días transcurridos del 01/01/2000 a la fecha indicada
#include <conio.h>
#include <stdio.h>
int main(void)
{
   int anio, mes, dia, suma;
   printf("Por favor, indique una fecha posterior al año 2000");
   printf(" (dd mm aaaa).\n");
   scanf("%d", &dia);
   scanf("%d", &mes);
   scanf("%d", &anio);
   suma = (anio - 2000 ) * 365; // días por años completos

   // días por meses completos del año en curso
   switch (mes) {
      case 1: suma += 0; break;
      case 2: suma += 31; break;
      case 3: suma += 59; break;
      case 4: suma += 90; break;
      case 5: suma += 120; break;
      case 6: suma += 151; break;
      case 7: suma += 181; break;
      case 8: suma += 212; break;
      case 9: suma += 243; break;
      case 10: suma += 273; break;
      case 11: suma += 304; break;
      case 12: suma += 334; break;
   }
   suma += (anio - 2000) / 4 + 1; // días por años bisiestos
   if ((anio % 4 == 0) && (mes < 3))
      suma--;
   suma += dia; // días del mes en curso
   printf("\nEl resultado es: %d", suma);

   printf("\n\nOprime cualquier tecla..");
   getch();
}
```

Al final se obtendrá una pantalla como la que se muestra en la figura 2.9.

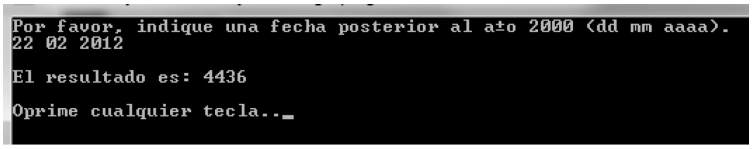


Figura 2.9 Pantalla final de un programa que calcula los días transcurridos entre el 1 de enero de 2000 y una fecha señalada por el usuario.

Como se puede observar, los días del mes dependen del valor de la variable `mes`. Este es el caso en que se aplica una instrucción `switch`: diferentes acciones en razón del valor de una variable. Su estructura general es la siguiente:

```
switch (variable)
case valor: {bloque de instrucciones}
   break; // el break es opcional
/* pueden repetirse tantos case como se deseen */
default: {bloque de instrucciones}
```

El `break` indica que ya no se sigan explorando los siguientes casos. En la mayoría de las situaciones se desea un `break` para cada opción, pero en otras es deseable que —además de lo realizado en el caso anterior— se lleven a cabo otras instrucciones, en cuyo caso no se coloca ningún `break` en la opción que finaliza. Por último, si ninguna de las opciones se cumplió, se ejecutará el bloque de comandos señalado por `default`.[3]

EJERCICIOS SUGERIDOS

Problemas propuestos para decisiones múltiples

Una tienda comercializadora maneja un nivel de descuentos con base en el monto total de la compra, según la siguiente tabla:

Monto de la compra	% de descuento
$ 2 990.00	3
7 490.00	5
13 990.00	7
54 990.00	10

- Elabore un programa que reciba el monto de la compra original y, en razón de ese dato, localice el porcentaje de descuento y lo aplique. Por último, desplegará el monto final a pagar.
- Realice un programa que "juegue" piedra, papel o tijera. El usuario elige una opción de las tres y la computadora genera al azar una de ellas. Luego, el resultado se dará bajo las siguientes reglas; la piedra le gana a la tijera; la tijera le gana al papel; el papel le gana a la piedra; si ambos casos son iguales, resulta un empate.[3]
- Realice un programa que reciba dos datos enteros: el día y el mes y con base en ellos despliegue el día y el mes. Por ejemplo: si recibe como datos 19 12, desplegará 19 de diciembre.

[3] Existe una versión menos conocida, pero un poco más divertida, con base en conejo, flecha y muro.

2.7 Uso de ciclos

En lenguaje C todos los ciclos indican que un determinado bloque de instrucciones se repita mientras se cumpla determinada condición. Existen tres tipos, como se sintetiza en el cuadro 2.1.

Cuadro 2.1 Los ciclos en lenguaje C y Java

`while`	Mientras que, se valida al inicio por lo cual el bloque de instrucciones puede no ejecutarse ni una sola vez.
`do-while`	Mientras que, se valida al final por lo cual el bloque de instrucciones se ejecuta al menos una vez.
`for`	Por lo general se aplica cuando se conoce de antemano las veces que se repetirá un conjunto de instrucciones. Aunque en lenguaje C tiene la misma potencialidad que el `while`: la condición se coloca al principio y se ejecuta el bloque de instrucciones mientras se cumpla la condición.

El ciclo `while`

En el `while` se repite un bloque de instrucciones mientras se cumple una determinada condición. La condición se verifica al inicio, como lo muestra la figura 2.10.

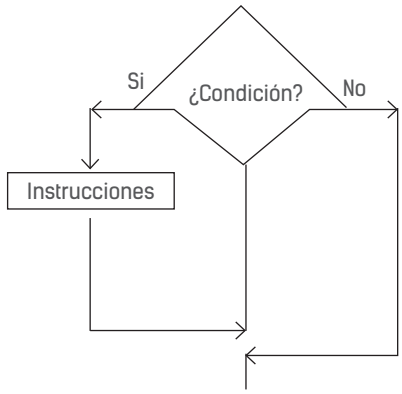


Figura 2.10 Representación del ciclo `while` en un diagrama de flujo.

En resumen:

```
mientras (condición)
   bloque de instrucciones
termina mientras
```

Hay que hacer hincapié en que la condición significa mientras que. Una condición que siempre se cumpla generaría un ciclo infinito; por el contrario, una condición que nunca se cumpla implicaría que no se ejecutaran las instrucciones del cuerpo del ciclo ni una sola vez.

Un error muy frecuente es no identificar con claridad aquello que es repetitivo (va a ir dentro del ciclo) con lo que debe hacerse una sola vez antes o después del ciclo (debe ir fuera de este). Este error suele dar resultados equivocados o generar ciclos infinitos.

EJEMPLO

A manera de ejemplo, suponga que desea obtener el promedio de varios números proporcionados por el usuario. La pantalla que desea es similar a la siguiente.

```
Este programa obtiene el promedio de calificaciones.

¿Cuántas calificaciones se van a introducir?
3
Capture las calificaciones, por favor.
8
9
10
El promedio es 9.00

Oprima cualquier tecla para terminar...
```

¿Qué debe ir dentro del ciclo? Resulta obvio que la lectura de las calificaciones. Como el valor de una variable se pierde al recibir un segundo valor, entonces se requiere ir sumando las calificaciones en otra variable. Las demás partes del programa irán fuera del ciclo. El código al final quedaría como lo indica el programa 2.6. En la figura 2.11 se muestra el diagrama de flujo correspondiente producido en forma automática por ZinjaI a través del menú Herramientas.

```
// Programa 2.6: promedio de calificaciones con while
#include <conio.h>
#include <stdio.h>
int main() {
   int i, n;
   float suma = 0.0, promedio, dato;
   printf ("Este programa obtiene el promedio de calificaciones.\n");
   printf ("¿Cuántas calificaciones se van a introducir?\n");
   scanf("%d",&n);
   if (n > 0) {
      printf("Capture las calificaciones, por favor.\n");
      i = 1;
      while (i <= n) {
         scanf("%f", &dato);
         suma += dato;
         i++;
      }
      promedio = suma / n;
      printf("El promedio es: %5.2f\n", promedio);
   }
   else
      printf("No puedo sacar el promedio de 0 calificaciones.\n");
      printf("\nOprima cualquier tecla para terminar...\n");
   getch();
}
```

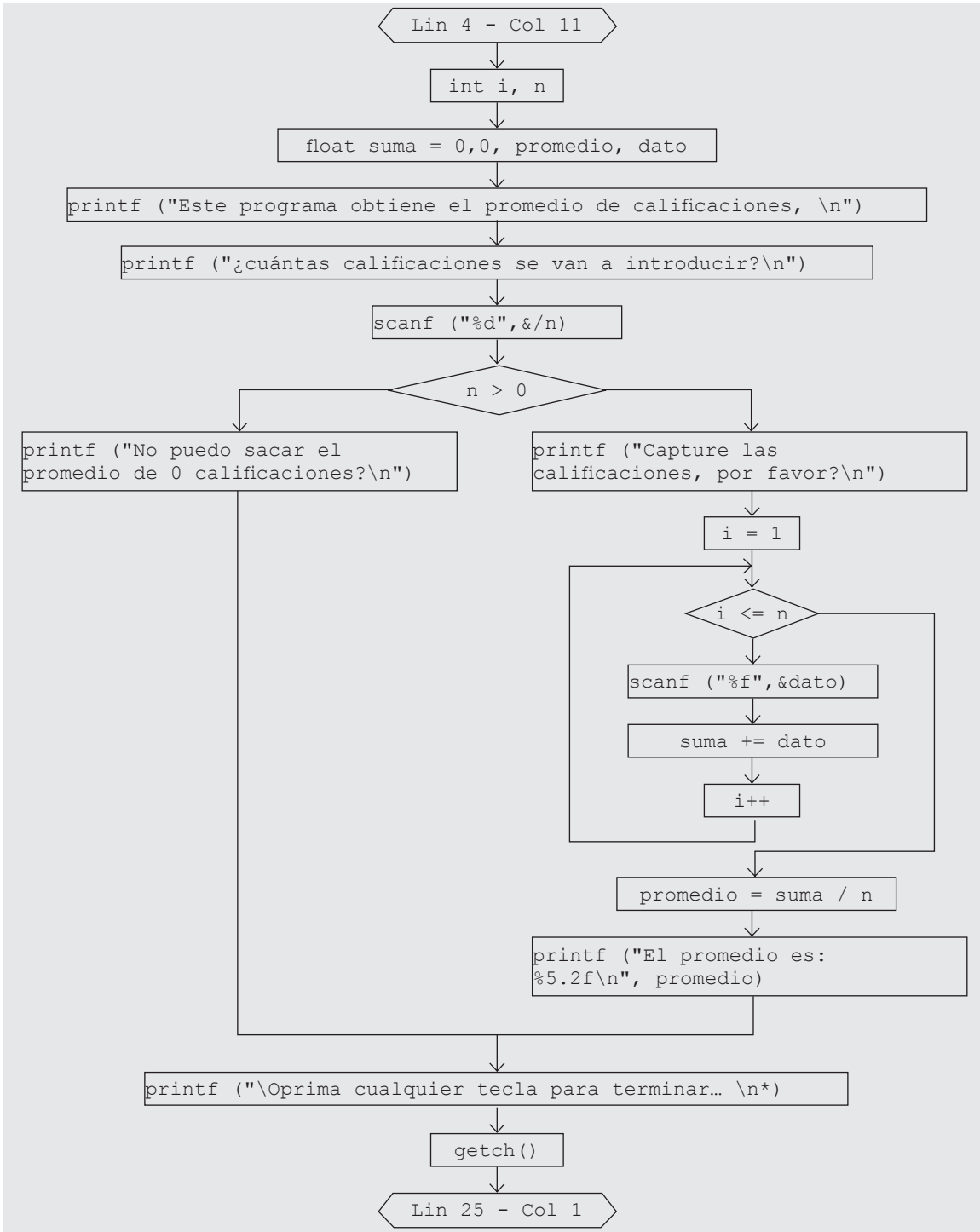


Figura 2.11 Diagrama de flujo de un programa que obtiene el promedio de varios números, obtenido en forma automática en el compilador Zinjal.

El ciclo `for`

En lenguaje C, en realidad el ciclo `for` es otra forma de expresar la misma funcionalidad que el ciclo `while`, por lo cual su diagrama es el mismo. Sin embargo, en otros lenguajes su sintaxis les permite únicamente hacer un ciclo cuando ya se conoce con exactitud el número de veces que se repetirá algo. Su sintaxis es la siguiente:

```
for (sentencias antes del ciclo; condición; últimas sentencias a repetir)
```

Cabe aclarar que cualquiera de las tres partes que constituyen el `for` puede omitirse. En un caso exagerado, un ciclo podría expresarse como `for (;;)`. Como recomendación general, hay que evitar este tipo de instrucciones poco entendibles.

Como ejemplo de un `for` bien aplicado, el programa 2.7 despliega cuatro veces hola.

```
// Programa 2.7: despliegue de hola con ciclo for
#include <stdio.h>
#include <conio.h>
int main() {
   int i;
   printf("Este programa ejemplifica un for desplegando 4 veces hola");
   for (i=0; i<4; i++)
      printf("hola");
      printf("\nOprima cualquier tecla para terminar...\n");
      getch();
}
```

El ejemplo desarrollado en páginas anteriores quedaría como lo muestra el programa 2.8.

```
// Programa 2.8: promedio de calificaciones con for
#include <stdio.h>
#include <conio.h>
int main() {
   int i, n;
   float suma = 0.0, promedio, dato;
   printf ("Este programa obtiene");
   prinff(" el promedio de calificaciones.\n");
   printf ("¿Cuántas calificaciones se van a introducir?\n");
   scanf("%d",&n);
   if (n > 0) {
      printf("Capture las calificaciones, por favor.\n");
      for (i=1; i <= n; i++) {
         scanf("%f", &dato);
         suma += dato;
      }
      promedio = suma / n;
      printf("El promedio es: %5.2f\n", promedio);
   }
   else
      printf("No puedo sacar el promedio de 0 calificaciones.\n");
      printf("\nOprima cualquier tecla para terminar...\n");
      getch();
}
```

El cuadro 2.2 resume las diferencias entre ambos programas; es decir, entre el promedio obtenido a través de un `while` y utilizando un `for`.

Cuadro 2.2 Diferencias entre ambos programas

Caso `while`

```
i = 1;
while (i <= n) {
   scanf("%f", &dato);
   suma += dato;
   i++;
}
```

Caso `for`

```
for (i=1; i <= n; i++) {
   scanf("%f", &dato);
   suma += dato;
}
```

El ciclo `do-while`

Tanto el `while` como el `for` validan la condición antes de ejecutar el bloque de instrucciones la primera vez, con lo cual abren la posibilidad de que este no se ejecute ni una sola vez. El `do-while` valida la condición al final, por lo cual existe la garantía de que dicho bloque se ejecute al menos una vez. En los tres casos se repetirá **mientras que** la condición se cumpla.

La estructura `do-while` queda como sigue. Por su parte, la figura 2.12 muestra el ciclo `do-while` en diagrama de flujo y el programa 2.9 muestra el promedio de varios números, ahora al emplear `do-while`.

```
realiza
   bloque de instrucciones
mientras (condición)
```

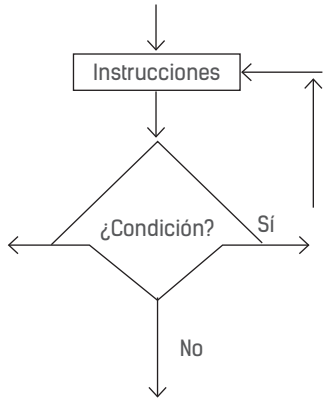


Figura 2.12 El ciclo `do-while` en un diagrama de flujo.

```
// Programa 2.9: promedio de calificaciones con do-while
#include <conio.h>
#include <stdio.h>
int main() {
   int i, n;
   float suma = 0.0, promedio, dato;
   printf ("Este programa obtiene el promedio de calificaciones.\n");
   printf (";Cuántas calificaciones se van a introducir?\n");
   scanf("%d",&n);
   if (n > 0) {
      printf("Capture las calificaciones, por favor.\n");
      i=1;
      do {
```

```
        scanf("%f", &dato);
        suma += dato;
        i++
     } while (i<=n);
     promedio = suma / n;
        printf("El promedio es: %5.2f\n", promedio);
  }
  else
     print("No puedo sacar el promedio de 0 calificaciones.\n"_;
     print("nOprima cualquier tecla para terminar...\n");
  getch();
}
```

¿Cuándo usar cada tipo de ciclo?

En muchas ocasiones un problema puede resolverse con cualquiera de los tres ciclos existentes. Sin embargo, conviene recalcar algunos puntos para facilitar su selección:

- En todos los lenguajes, el `while` puede hacer lo que realicen el `for` y el `do-while`.
- De manera específica en lenguaje C y Java, el `for` tiene la funcionalidad de un `while`.
- Aunque el `for` es tan potente como el `while`, se recomienda usarlo solo cuando se conoce con exactitud el número de veces que se repetirá un bloque de instrucciones. Aplicarlo a otras situaciones traerá como consecuencia códigos confusos.
- En el `do-while`, el bloque de instrucciones se ejecuta al menos una vez.

El ejemplo anterior se prestaba, de una u otra manera, para todos los tipos de ciclos. Ahora suponga que el requerimiento cambia al siguiente:

```
Este programa obtiene el promedio de calificaciones.
Capture las calificaciones. Digite -1 para terminar.
8
9
10
-1
El promedio es 9.00

Oprima cualquier tecla para terminar...
```

Si se utiliza un `while`, el código queda como lo muestra el programa 2.10.

```
// Programa 2.10: promedio de datos con while
#include <conio.h>
#include <stdio.h>
int main() {
   int i;
   float suma = 0.0, promedio, dato;
   printf ("Este programa obtiene el promedio de calificaciones.\n");
   printf ("Capture las calificaciones. Digite -1 para terminar.\n");
   i = 0;
   scanf ("%f", &dato);
   while (dato >= 0) {
      suma+=dato;
      i++;
      scanf("%f",&dato);
   }
```

```
   if (i > 0) {
      promedio = suma / i;
      printf("El promedio es: %5.2f\n", promedio);
   }
   else
      printf("No es posible obtener el promedio si no teclea ningún número.");
      printf("\n\nOprima cualquier tecla para terminar...\n");
   getch();
}
```

Como ya se mencionó, se puede realizar ese mismo programa con `for`, pero el código pierde claridad, sin importar cómo quede. El programa 2.11 muestra una de las distintas posibilidades:

```
// Programa 2.11: promedio de datos con un for nada claro
#include <conio.h>
#include <stdio.h>
int main() {
   float suma = 0.0, promedio, dato;
   int i;
   printf("Este programa obtiene el promedio de calificaciones. ");
   printf( "Capture -1 para terminar.\n");
   i = 0;
   for ( ; dato >= 0 ; ) {
      scanf("%f", &dato);
      if (dato >= 0) {
      suma += dato;
      i++;
      }
   }
   if (i > 0) {
      promedio = suma / i;
   printf("El promedio es: %5.2f\n", promedio);
   }
   else
      printf ("No se proporcionó ninguna calificación.\n");
      printf ("\nOprima cualquier tecla para terminar...\n");
   getch();
}
```

EJERCICIOS SUGERIDOS

Ejercicios sobre ciclos a nivel código

A continuación se enlistan diversos ejercicios. Algunos de ellos están en el contexto de un programa con una finalidad orientada hacia el usuario final; otros solo pretenden que el estudiante practique la codificación a través de la correcta aplicación de ciclos.

- Realice un programa que despliegue cuatro veces hola, utilice: a) `while`; b) `for`; c) `do-while`.
- ¿Cómo obtendría el siguiente despliegue?

```
¿Cuántas veces deseas desplegar hola? 3
hola hola hola
```

- ¿Cómo obtendría el siguiente despliegue?

```
hola hola hola
a todos a todos a todos
```

- ¿Cómo obtendría el siguiente despliegue?

```
hola
   a todos
   a todos
   a todos
hola
   a todos
   a todos
   a todos
```

- ¿Cómo obtendría el siguiente despliegue?

```
¿Cuántas veces deseas desplegar hola? 3
¿Cuántas veces deseas desplegar a todos? 2
hola
   a todos
   a todos
hola
   a todos
   a todos
hola
   a todos
   a todos
```

- Transforme el siguiente código a un `for`

```
i = 0;
while (i<5) {
   printf("hola");
   i++;
}
```

- Corrija el siguiente programa sin tener a la vista el programa correcto.

```
#include <stdio.h>
#include <conio.h>
int main () {
   float suma, dato, promedio;
   int numDatos, i;
   printf("Este programa despliega");
   printf(" el promedio de varios números.\n\n");
   printf("Indique la cantidad de números: ");
   scanf("%d", &numDatos);
   i = 1;
   suma = 0.0;
   printf("Teclee los datos:\n ");
   while (i < numDatos) {
      scanf("%f", &dato);
      suma += dato;
      i++;
      promedio = suma / numDatos;
      printf("El promedio es: %6.1f\n", promedio);
   }
   printf("\n\nOprima cualquier tecla para continuar...");
   getch();
}
```

- El siguiente programa pretende obtener el promedio de 4 números. Analícelo e indique lo que desplegaría en pantalla al ejecutarlo, suponga que cuando el programa pide "Teclee los datos" el usuario responde 2, 5, 9, 14.

```
#include <stdio.h>
#include <conio.h>
int main () {
  float suma, dato, promedio;
  int numDatos, i;
  printf("Este programa despliega el promedio de 4 números.\n\n");
  numdatos = 4;
  i = 1;
  suma = 0.0;
  printf("Teclee los datos:\n ");
  while (i < numDatos)
  {
     scanf("%f", &dato);
     suma += dato;
     i++;
     promedio = suma / numDatos;
     printf("El promedio es: %6.1f\n", promedio);
  }
  printf("\n\nOprima cualquier tecla para continuar...");
  getch();
}
```

- Un usuario desea obtener la siguiente pantalla:

```
Este programa calcula el promedio de varias calificaciones:

¿Cuántas calificaciones va a capturar?
3
Teclee las calificaciones
6.6
7.2
9.9
El promedio es 7.9

Oprima cualquier tecla para continuar...
```

- Elija las siguientes instrucciones en el orden correcto para llegar al resultado deseado. Las instrucciones pueden utilizarse más de una vez, sobran algunas y usted debe darles la sangría adecuada. Hay más de una solución correcta.

```
i++;
while (i < n)
{
float i, n;
i = 0;
getch();
printf("Oprima cualquier tecla para continuar...");
i = 1;
}
while (i <= n)
scanf("%d", &n);
suma = 0;
promedio = suma / n;
float suma, promedio;
```

```
int i, n;
float suma=0, promedio;
#include <conio.h>
int main()
#include <stdio.h>
printf("Este programa calcula...");
float dato;
printf("Teclea las calificaciones");
suma = suma + dato;
suma += dato;
scanf("%f%, &dato);
printf("¿Cuántas calificaciones va a capturar");
printf("El promedio es %5.1f", promedio);
```

- Realice el programa que responda al siguiente requerimiento.

```
Voy a contar las calificaciones aprobadas.
Teclea las calificaciones (termina con -1)
7
4
8
10
-1
Obtuviste 3 calificación(es) aprobatoria(s) y 1 calificación(es)
reprobatoria(s).
Tu promedio fue: 7.25
```

- El siguiente programa debe obtener la multiplicación de x * n. Codifique la parte que haga falta para obtener el resultado sin emplear la multiplicación, es decir, con base en sumas sucesivas.

```
int main() {
  float x, resultado;
  int n, i;
  scanf("%f", x);
  scanf("%d", n);

  /* poner código */

  printf("Resultado=%f", resultado);
}
```

- Realice un programa que calcule el factorial de un número indicado por el usuario. No existen factoriales de números negativos; el factorial de 0 es uno. Para guardar el factorial se requiere de un tipo de dato que guarde rango de valores mayores que el int. La pantalla será similar a la siguiente:

```
  Bienvenido.
  Este programa calcula el factorial de un número.
  ¿Qué número es?
  5
  Resultado: 1 * 2 * 3 * 4 * 5 = 120
```

- Realice un programa que obtenga la división de dos números enteros sin utilizar el operador de división (a través de restas sucesivas).

- Un apostador asiduo tiene la costumbre de apostar si atina al número que resulta al tirar un dado. Si lo logra gana $5.00; en caso contrario, pierde $1.00. Comienza con $7.00 y el juego termina cuando gana $6.00 o pierde todo su capital.

```
¿Qué número saldrá en el dado? 5
El dado arrojó un 4. Acaba de perder $1.00
Su saldo es $6.00

¿Qué número saldrá en el dado? 5
```

- Juguemos a menor-mayor. Usted tiene que apostar un peso a alguna de las siguientes opciones:

Menor	7	Mayor
2-6	7	8-12

Después tira dos dados. Si le apostó a menor y la suma de los dados está en el rango de 2 a 6 gana un peso. Algo similar sucede con el caso de mayor. Si le apostó a 7 y la suma de los dados coincide con 7, entonces gana 5 pesos. Arranca con 5 pesos y termina su participación cuando tiene 10 pesos o pierde todo su capital.

Si parte del siguiente código, indique cuál sería el resultado si el usuario teclea 3 para el valor de n:

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
int main() {
   /* las iteraciones se declararan flotantes
      porque pueden exceder el rango de enteros */
   float i, numiteraciones;
   double sumatoria=0.0, resultado;
   printf("Este programa obtendrá un valor muy cercano a PI ");
   printf("aplicando la fórmula\n PI = 1 / 2 * raíz ");
   printf("cuadrada(sumatoria(1..n de 24/(i*i))).");
   printf("\n\nTeclee el valor de n, por favor.\n");
   scanf("%f", &numiteraciones);
   for (i = 1.0; i <= numiteraciones; i += 1.0)
      sumatoria += 24 / (1.0 * i * i);
   resultado = sqrt(sumatoria) / 2;
   printf("\nEl valor obtenido es: %8.4f", resultado);
   printf("\nEl error absoluto es: %8.4f",
   fabs(resultado-3.1416));
   printf("\nEl error relativo es: %8.4f",
   fabs(resultado-3.1416) / 3.1416 * 100.0);
   printf ("\nOprima cualquier tecla para terminar...\n");
   getch();
}
```

Capítulo 3

Subrutinas y estructuras básicas para el manejo de datos

3.1 Arreglos

Hemos llegado a un punto en que la programación se vuelve más interesante: comenzaremos a trabajar con arreglos, listas o tablas. Para este fin es importante hacer referencia a las estructuras de datos, que en un entorno de programación representan una forma de organizar un grupo o conjunto de datos con el objetivo de hacer más fácil su manipulación.

Si consideramos que un dato es como un objeto que será procesado por un programa de computadora, una estructura de datos es una colección de elementos cuyo comportamiento se caracteriza por las operaciones de acceso usadas para almacenar y recuperar los elementos individuales.

En programación se llama estructuras estáticas a datos compuestos de otros más simples que se utilizan como si fueran un único dato y ocupan un espacio concreto en la memoria de la computadora.

Las estructuras estáticas son:

- **Arreglos** (`array`). También se les llama, según el caso, listas estáticas o matrices. Son una colección de datos del mismo tipo.
- **Cadenas** (`strings`). Son un conjunto de caracteres tratados como un texto. De hecho, una cadena es un arreglo de caracteres. En lenguaje C no existe un tipo de datos `string` como tal; pero en otros lenguajes (como Java) se maneja.
- **Apuntadores** (`punteros`). Definen variables que contienen posiciones de memoria. Señalan a otras variables.
- **Estructuras**. También llamadas **registros**. Son datos compuestos de datos de distinto tipo, puede considerar enteros, carácter o arreglos.

Las estructuras de datos son una clase caracterizada por su organización y tipo de operaciones que se pueden definir sobre ellas. Entonces tenemos:

- **Estructuras lógicas de datos.** Cada estructura de datos puede tener varias representaciones físicas diferentes para sus posibles almacenamientos.
- **Estructuras primitivas.** Aquellas que no están compuestas por otras estructuras de datos; ejemplo: enteros y caracteres. Las estructuras de datos simples se construyen a partir de estructuras primitivas y son: cadenas, arreglos y registros.

Con la combinación de las estructuras de datos simples se pueden formar otras más complejas. Por ejemplo, las estructuras de datos lineales (pilas, colas y listas ligadas lineales) y las estructuras de datos no lineales (grafos y árboles).

De acuerdo con el número de dimensiones que tienen los arreglos, se clasifican en:

- Unidimensionales, también conocidos como vectores.[1]
- Bidimensionales, también conocidos como matrices o tablas.
- Multidimensionales, que tiene tres o más dimensiones.

Al referirnos a los elementos que contiene un arreglo, se usan índices, mismos que deben iniciar desde una posición cero, por ejemplo: los días de la semana:

[1] A los arreglos unidimensionales también se les conoce como vector, pero es necesario aclarar que es un término coloquial y no debe confundirse con los vectores empleados de manera formal en el campo de la física y las matemáticas.

```
char Dia[6], por tanto: Dia[0], Dia[1],…,Dia[6]

char Dia[0] = 'l';
char Dia[1] = 'm'
…
char Dia[6] = 'd'
```

Arreglos unidimensionales

Un arreglo unidimensional, también conocido de manera coloquial como vector, es una colección de datos de un mismo tipo que ocupan posiciones de almacenamiento en forma consecutiva en la memoria de la computadora y reciben un nombre común. Se hace referencia a los elementos del arreglo mediante un índice, el cual señala su ubicación relativa dentro del arreglo.

Un arreglo es una colección finita, homogénea y ordenada de elementos. Finita porque tiene un límite, es decir, se determina el número máximo de elementos que lo componen; homogénea porque los elementos del arreglo son de un mismo tipo, y ordenada en razón de que estos tienen una disposición establecida desde el primero hasta el último elemento.

Existen algunas restricciones al tratar con arreglos:

- Todos los elementos del arreglo deben tener el mismo tipo.
- En general, el tamaño del arreglo es fijo.
- Se ocupan comúnmente para almacenar datos numéricos.

La función que nos da la posición o dirección de un elemento en un arreglo unidimensional asociado a la expresión índice o subíndice es: dirección [índice o subíndice] (véase cuadro 3.1).

Cuadro 3.1 La posición de memoria A[n-1] contiene el *n*-ésimo dato

Dato 1	Dato 2	Dato 3	Dato 4	...	*n*-ésimo dato
A(0)	A(1)	A(2)	A(3)	–	A(*n*–1)

La dirección de memoria A[0] contiene el Dato 1
La dirección de memoria A[1] contiene el Dato 2
La dirección de memoria A[2] contiene el Dato 3
La dirección de memoria A[3] contiene el Dato 4
….

Entonces, podemos entender los arreglos o vectores como variables que contienen diferentes tipos de datos homogéneos, a los que se puede acceder de manera individual mediante un índice o subíndice, por ejemplo: A[0] = Dato 1.

El programa 3.1 lee cinco valores desde el teclado a través de un ciclo y los almacena en el arreglo A (líneas de código 9 a 14); después los despliega en pantalla (líneas 17-20).

```
1  // Programa 3.1: leer, almacenar e imprimir
2  // los valores de un arreglo
3  #include <stdio.h>
4  #include <stdlib.h>
5  int main()
6  {
```

```
7     // lee valores
8     int A[5],i,valor;
9     for(i=0; i<5;i++){
10       printf("Teclee un número: ");
11       scanf("%d",&valor);
12       // almacena valores
13       A[i]=valor;
14    }
15    // imprime valores
16    printf("\nContenido del arreglo con los datos leídos:\n\n");
17    for(i=0; i<5;i++) {
18       printf("La direccion de memoria A[%d]= ");
19       printf("contiene el dato %d\n",i,A[i]);
20    }
21    system("pause");
22    return 0;
23 }
```

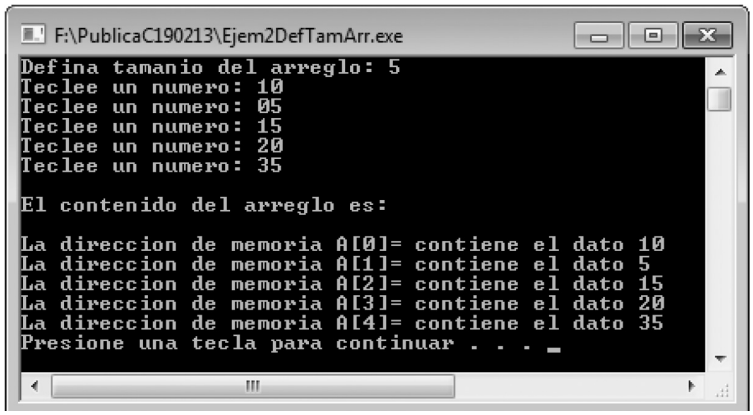


Figura 3.1

Hay que hacer hincapié en las cuatro propiedades básicas de los arreglos:

- Los términos individuales de datos de un arreglo se denominan elementos.
- Los elementos deben ser del mismo tipo de dato.
- Los elementos se almacenan en posiciones contiguas de memoria de la computadora y el índice o subíndice del primer elemento es cero.
- El nombre de un arreglo es un valor constante que representa la dirección de memoria del primer elemento del arreglo.

Definición de arreglos en términos de una constante

En ocasiones es necesario y conveniente especificar el tamaño de los arreglos en función de una constante dentro de un programa, con lo cual se garantiza que los arreglos no excedan el tamaño definido. El lenguaje C no valida el rango definido por los índices o subíndices. El programa 3.2 con su respectiva salida ilustra la situación mencionada.[2]

[2] Esta posibilidad no la tenían los compiladores originales de lenguaje C. De hecho, el código ilustrado se ejecuta sin problemas en Dev C++ 4.9.9.2, pero la versión de Zinjal a inicios de 2013 no lo compila.

```
/* Programa 3.2: programa para definir el tamaño de un arreglo
en función de una constante */
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int constante,contenido,i,j,;
   printf("Defina tamanio del arreglo: ");
   scanf("%d",&constante);
   int A[constante];
   for(i=0; i<constante;i++)
   {
      printf("Teclee un numero: ");
      scanf("%d",&contenido);
      A[i]=contenido;
   }
   printf("\nEl contenido del arreglo es:\n\n");
   for(j=0; j<constante;j++)
   {
   printf("La direccion de memoria ");
   printf(" A[%d]= contiene el dato %d\n",j,A[j]);
   }
   system("pause");
   return 0;
}
```

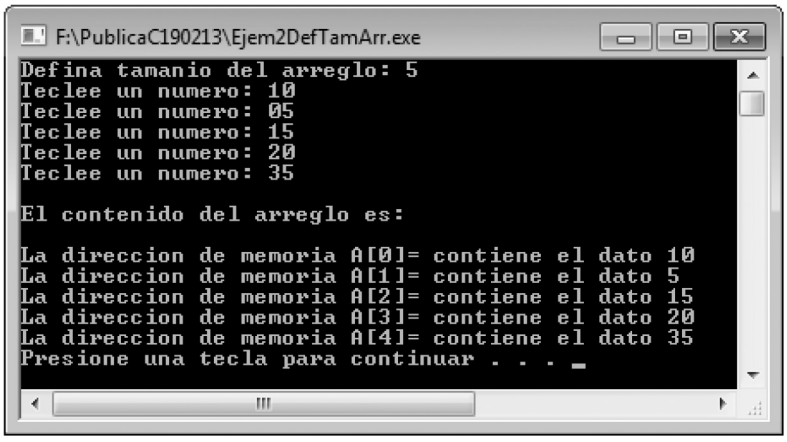


Figura 3.2

De modo similar, se pueden definir arreglos de cualquier tipo de datos, por ejemplo: enteros, con decimales, números con punto flotante y caracteres.

```
int calificaciones[3] = {9,10,9}; // arreglo de tres elementos enteros.
int calificaciones [ ] = {9,10,9}; // arreglo de tres elementos enteros.
```

El programa 3.3 muestra cómo se puede inicializar un arreglo.

```
//Programa 3.3: programa de ejemplo que inicializa arreglos
#include <stdio.h>
#include <conio.h>
```

```
#include <stdlib.h>
int main()
{
   int calificaciones[3] = {9,10,9}; // Linea 6 del código.
   int i=0;
   for(i=0;i<3;i++)
      printf("%2d ",calificaciones[i]);
   system("pause");
}
```

La línea 6 es equivalente al siguiente código. Dejamos la comprobación al estudiante,

```
int calificaciones[3];
calificaciones[0] = 9;
calificaciones[1] = 10;
calificaciones[2] = 9;
```

Otros ejemplos de inicialización de arreglos:

```
// arreglo de 5 elementos con punto flotante y double
float promedio[5]={8.5,9.5,8.0,9.0,8.5};
double grafica[5]={0.431, 2.283, -0.5, 3.819, -34.124};

//arreglo de caracteres indicado en forma breve
char nombrelibro[] = "Programación";

// arreglo de trece elementos elemento por elemento
// la última posición es un carácter nulo (NULL).
char nombrelibro[13] =
 {'P','r','o','g','r','a','m','a','c','i','o','n','\0'};
```

Conviene aclarar que las cadenas (`string`) son un arreglo de caracteres más un carácter de control interno que no forma parte de la cadena, este carácter equivale a un valor de cero en bits y por lo normal es expresado con '\0'. El número de bytes ocupados es el número de los caracteres de la palabra más uno, que corresponde al nulo (NULL).

Una vez que tenemos datos almacenados en un arreglo, se puede proceder a realizar diferentes operaciones. Por ejemplo, sumar el contenido de los elementos de un arreglo. El programa 3.4 ejemplifica cómo hacerlo:

```
1  // Programa 3.4: imprime la sumatoria de los elementos del arreglo
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define constante 5
5  int main()
6  {
7     int arreglo[] = {56,75,28,49,95};
8     int i, sumatoria = 0;
9     for ( i=0; i<constante; i++ )
10    {
11       printf("arreglo[%d]= %d\n",i,arreglo[i]);
12    sumatoria += arreglo[i];
13    }
14    printf("Sumatoria = %d\n\n", sumatoria);
15    system("pause");
16    return(0);
17 }
```

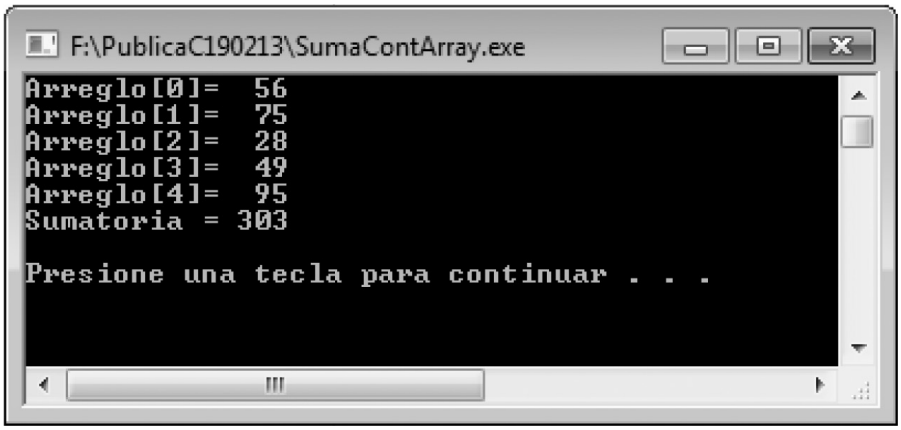


Figura 3.3

Arreglos multidimensionales

Los arreglos multidimensionales son estructuras de datos que contienen o agrupan múltiples datos de un mismo tipo, que pueden ser tratados en forma individual y se hace referencia a ellos con un mismo nombre. Los arreglos bidimensionales son un caso concreto de arreglo multidimensional, que contiene dos índices.

A diferencia de los arreglos o vectores, en los arreglos bidimensionales o matrices, sus elementos no se organizan de manera lineal, sino bidimensional (renglones y columnas).

Su representación gráfica puede verse como una tabla con renglones y columnas, en donde, para localizar o almacenar valores, se especificarán dos índices o subíndices, uno para renglones y otro para columnas (véase cuadro 3.2).

Cuadro 3.2 Representación gráfica de un arreglo

		Columnas				
		0	1	2	3	4
Renglones	0	A[0, 0]	A[0, 1]	A[0, 2]	A[0, 3]	A[0, 4]
	1	A[1, 0]	A[1, 1]	A[1, 2]	A[1, 3]	A[1, 4]
	2	A[2, 0]	A[2, 1]	A[2, 2]	A[2, 3]	A[2, 4]
	3	A[3, 0]	A[3, 1]	A[3, 2]	A[3, 3]	A[3, 4]
	4	A[4, 0]	A[4, 1]	A[4, 2]	A[4, 3]	A[4, 4]

Podemos imaginar una matriz (arreglo) como localidades de memoria o casillas, en las que se pueden almacenar los elementos de la colección de datos, de manera que si tenemos un arreglo de cinco renglones y cinco columnas (arreglo[5][5]), y deseamos almacenar los elementos de la colección de datos como la que se muestra en el cuadro 3.3, el contenido del arreglo A(1, 1) será igual a 3, el contenido de A(1, 3) será igual a 7 y el de A(4, 4) será igual a 70.

Cuadro 3.3 Ejemplo de un arreglo bidimensional

		Columnas				
		0	1	2	3	4
Renglones	0	2	4	6	8	10
	1	1	3	5	7	9
	2	12	14	16	18	20
	3	11	13	15	17	19
	4	30	40	50	60	70

El programa 3.5 ejemplifica cómo se almacenan datos en el arreglo (Arreglo[5][5]).[3]

```
1  // Programa 3.5: ejemplo de una matriz bidimensional
2  //               de cinco renglones y cinco columnas
3  #include <stdio.h>
4  #include <stdlib.h>
5  int main()
6  {
7     int renglon, columna;
8     int Arreglo[5][5]=
9           {2,4,6,8,10,1,3,5,7,9, 12,14,16,18,20,
10          11,13,15,17,19,30,40,50,60,70};
11    for(renglon=0; renglon<5; renglon++)
12       for(columna=0; columna<5; columna++) {
13          printf("La casilla o celda de memoria ");
14          printf("[%d][%d] = contiene un   %d\n",
15             renglon, columna, Arreglo[renglon][columna]);
16       }
17    system("pause");
18    return 0;
19 }
```

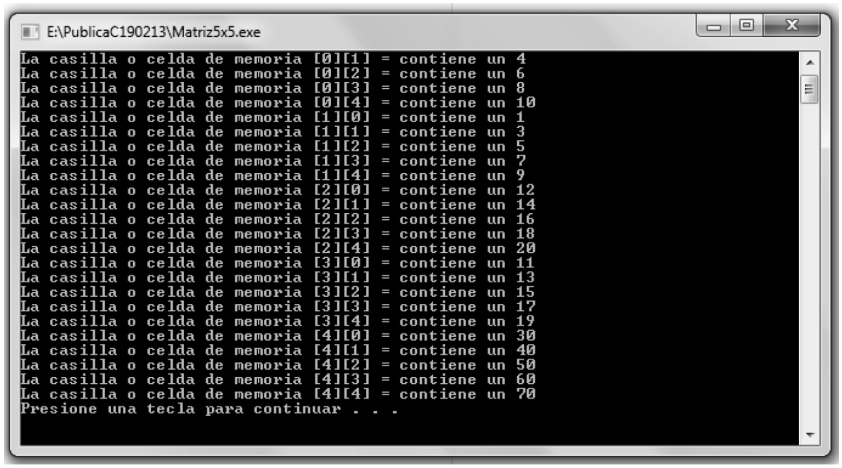


Figura 3.4

[3] La sintaxis expresada en este ejemplo es aceptada sin problemas en Dev C++; no así en Zinjal.

EJEMPLO

Programa ejemplo sobre arreglos

El cálculo de la desviación estándar es un ejemplo en el cual de manera forzosa se aplican arreglos. El cálculo se realiza de la siguiente forma:[4]

1. Suponga que se quiere obtener la desviación estándar de los siguientes números: 8, 9, 10, 5 , 7.
2. Se obtiene el promedio. En nuestro caso: (8 + 9 + 10 + 5 + 7) / 5 = 7.8.
3. Se obtiene la sumatoria del cuadrado de las diferencias entre cada dato y el promedio.
4. – 7.8) 2 + (9 – 7.8) 2 + (10 – 7.8) 2 + (5 – 7.8) 2 + (7 – 7.8) 2 = 14.8
5. Se divide entre el número de datos y se extrae la raíz cuadrada.

$$\sqrt{(14.80 / 5)} = 1.72$$

Como se observará, es necesario leer los números y obtener su promedio (pasos 1 y 2). Esos datos volverán a requerirse. Como es ilógico pedirlos de nuevo al usuario, deberán almacenarse en algún lugar de memoria. Al ser datos del mismo tipo, la forma más práctica es un arreglo. Ya en el arreglo, se hará un recorrido a través del mismo para calcular la sumatoria de la diferencia entre cada dato y el promedio (paso 3). Por último, se hará la división y la raíz cuadrada (paso 4).

El código se muestra en el programa 3.6.

```
/* Programa 3.6: programa que calcula la desviación estándar
de un grupo de números.
#include <conio.h>
#include <stdio.h>
#include <math.h>
int main() {
   float datos[50]; /*arreglo para guardar los datos*/
   float dato; /*variable para leer el dato del usuario*/
   float promedio, suma=0; /*promediar y sumar datos del usuario*/
   float s=0; /*desviación estándar*/
   int numdatos; /*número de datos del usuario*/
   int i; /*variable auxiliar para controlar los ciclos*/

   printf("Programa para calcular la desviacionestándar.\n");
   printf("¿Cuántos números va a ingresar (máximo 50)?\n");
   scanf("%d", &numdatos);
   printf("\nIngrese los datos a calcular\n");
   for (i=0; i<numdatos; i++) {
      scanf("%f", &dato);
      suma += dato;
      datos[i] = dato;
   }
   promedio = suma / numdatos;
   for (i=0; i<numdatos; i++) {
      /*suma el cuadrado de las diferencias */
      s += (datos[i] - promedio) * (datos[i] - promedio);
   }
   s = sqrt(s/numdatos);
   printf("\n\nResultado = %6.2f \n", s);

   printf("\n\nOprima cualquier tecla para continuar\n");
   getch();
}
```

[4] La fórmula es distinta para una muestra y para la población total. En nuestro caso, asumiremos que esos valores son toda la población existente.

El resultado sería como sigue:

```
Programa para calcular la desviación estándar.
¿Cuántos números va a ingresar (máximo 50)?
5
Ingrese los datos a calcular:
8
9
10
5
7
Resultado = 1.72

Oprima cualquier tecla para continuar...
```

EJERCICIOS SUGERIDOS

1. Llene con el valor de 6 los elementos del siguiente arreglo:

```
#define LIMITE 10
calif int [LIMITE]
```

2. Suponga un arreglo de nombre valores. a) Inicialice el arreglo con los valores señalados; b) disminuya el valor del elemento 5 en dos unidades; c) intercambie el valor de los elementos 3 y 6 (puede auxiliarse con otra variable); despliegue los valores finales del arreglo.

```
Indice ->         0    1    2    3    4    5    6
Contenido ->      4    6    9    2    8    3    9
```

3. Revise si el programa siguiente corre en forma correcta. Si es necesario, corríjalo. Si considera conveniente, mejore los mensajes hacia el usuario y lleve el código hacia buenas prácticas de programación.

```
/*
REQUERIMIENTO ORIGINAL
 Realice un programa que calcule el promedio de varios números
proporcionados por el usuario (la captura terminará con un -1).

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{

int i=1;
int p[7];
float promedio=0;
printf("marca -1 cuando allas ingresado todos los datos\n");
do{
 p[i]++;
printf("ingresa dato");
scanf("%d",&p[i]);
promedio=promedio+p[i];
```

```
}while (p[i]>-1 );

promedio= promedio/i;
printf("el promedio es %f", promedio);
system("PAUSE");
return EXIT_SUCCESS;
}
```

4. Realice un programa que reciba n números enteros y despliegue cuántos de esos números son mayores al último número proporcionado. La pantalla sería similar a la siguiente.

```
Bienvenido.
Este programa le dirá cuántos números son mayores
al último número que capture.
¿Cuántos números va a capturar?5
Indique los números:
6
7
9
10
8
Hay 2 números mayores al último número capturado.
```

Sugerencia: cree un arreglo que guarde los números capturados y después recorra ese arreglo con un ciclo después de contar los números mayores al último dato que se proporcionó.

5. Suponga que hay un camión con 10 lugares. Todos están disponibles al principio. Realice un programa que permita al usuario reservar "n" lugares, indique cuáles. Al final, despliegue los lugares aún disponibles. Nota: si el usuario da un lugar mayor de 10 se hará caso omiso de este dato.

```
Este programa reserva lugares en camión.
¿Cuántos lugares desea reservar?
3
¿Cuáles
4  7  9
Los lugares aún disponibles son:
1  2  3  5  6  8  10
```

Variante. Cambie la pantalla por la siguiente:

```
Este programa reserva lugares en camión.
¿Cuántos lugares desea reservar?
3
¿Cuáles
4 7 9
Situación al final:
1  2  3  4  5  6  7  8  9 10
D  D  D  -  D  D  -  D  -  D
```

6. Declare, llene y despliegue el arreglo, utilice el menor número de líneas de código posibles. La primera línea es el contenido y la segunda es el índice.

```
INDICE  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
VALOR   0 1 4 9 16 25 36 49 64 59 60 61 62 63 64
```

7. Suponga que existe el siguiente arreglo:

INDICE	0	1	2	3	4	5	6	7	8	9	10
VALOR	8	9	1	6	7	12	10	3	4	13	0

Pida un número al usuario. Trate de ubicar ese número en el arreglo. Si existe, despliegue el primer lugar donde lo encontró. En caso contrario, señale que el número no existe. Por ejemplo: si el usuario da 9, su programa debe decir: ubicado en el lugar 1.

8. Realice un programa que permita llevar a cabo la multiplicación de dos matrices, conforme a las reglas matemáticas normales.

9. Realice un programa que juegue tres en línea (también conocido como "gato"). El usuario podrá elegir quién tira primero, si él o la computadora. El programa detectará cuando es obvio que va a ganar o perder; las demás tiradas son al azar. Si lo desea, puede agregar otras reglas que lo hagan más eficaz.

3.2 Manejo de registros y archivos

Concepto de registros

Antes de la explicación del manejo de archivos, es necesario hacer un repaso a algunos conceptos, de manera que empezaremos por definir lo que es un registro. Se puede entender como un tipo de dato estructurado conformado por una colección de datos de igual o diferente tipo almacenados en distintas variables (campos), que pueden ser tratados como una unidad de información.

En otras palabras, un registro es un tipo de dato estructurado conformado por la unión de varios elementos bajo una misma estructura, que pueden ser datos elementales como entero, real, carácter u otras estructuras de datos. A cada uno de esos elementos se le llama campo.

Por ejemplo, para tener un registro de autos con la información: marca, modelo, año, identificador.

```
struct Autos{
   char marca[MAXCAD];
   char modelo[MAXCAD];
   char año[MAXCAD];
   int Identificador;
}
```

Por ejemplo si queremos tener los registro de diferentes marcas de autos y cuál es el color más vendido por año (véase cuadro 3.4).

- El dato 1, corresponde a la marca
- El dato 2, corresponde al año 2009
- El dato 3, corresponde al año 2010
- El dato 4, corresponde al año 2011
- El dato 5, corresponde al año 2012

Cada uno de esos datos, también conocidos como campos, conforman un registro, de manera que una colección de registros nos lleva a tener un archivo.

Cuadro 3.4 Tabla del archivo AutosColor.txt					
Autos con el color más vendido por año					
Datos	Dato 1	Dato 2	Dato 3	Dato 4	Dato 5
Registros	Marca	2009	2010	2011	2012
Registro 1	Tsuru	Blanco	Gris	Negro	Blanco
Registro 2	Chevy	Azul	Negro	Blanco	Rojo
Registro 3	Jetta	Negro	Rojo	Gris	Blanco
Registro 4	Focus	Plata	Blanco	Azul	Negro
Registro 5	Golf	Negro	Plata	Rojo	Blanco

Bajo este contexto, un archivo está sujeto a cambios constantes; es decir, con el paso del tiempo hay necesidad de adicionar, borrar, corregir o en su defecto consultar los registros contenidos en un archivo. Debe tenerse en cuenta que hay un proceso de actualización constante del archivo o archivos.

Manejo de registros y archivos

Como se mencionó con anterioridad un archivo es un conjunto de datos estructurados en una colección de entidades elementales llamadas registros que son de igual tipo y constan de diferentes entidades llamadas campos. En este punto podemos decir que existen y se manejan dos tipos de archivos: de texto y binarios.

Los archivos de **texto** se representan por secuencia o cadena de caracteres estructuradas en líneas y son delimitados con un carácter especial que marca e identifica una nueva línea.

Los archivos **binarios** son representados por una secuencia de bytes que tienen correspondencia uno a uno con un dispositivo externo.

La entrada y la salida de datos a un archivo se realiza mediante el uso de la biblioteca de funciones.

Las funciones comúnmente utilizadas son:

`fopen()`	abre un archivo.
`fclose()`	cierra un archivo.
`fgets()`	lee una cadena de un archivo.
`fputs()`	escribe una cadena en un archivo
`fseek()`	busca un byte específico de un archivo.
`fprintf()`	escribe una salida con formato en el archivo.
`fscanf()`	lee una entrada con formato desde el archivo.
`feof()`	valor cierto si es fin de archivo.
`ferror()`	valor cierto si existe error.
`rewind()`	se posiciona al principio de un archivo.
`remove()`	elimina un archivo.
`fflush()`	vacía un archivo.

Como podemos observar, las funciones comienzan con la letra "f", utilizadas en estándar C.

Existen diferentes modos para crear o abrir archivos:

`r`	abre archivo de texto para lectura.
`w`	crea archivo de texto para escritura.

a	abre archivo de texto para añadir.
rb	abre archivo binario para lectura.
wb	crea archivo binario para escritura.
ab	abre archivo binario para añadir.
r+	abre archivo de texto para lectura / escritura.
w+	crea archivo de texto para lectura / escritura.
a+	añade o crea un archivo de texto para lectura / escritura.
r+b	abre archivo binario para lectura / escritura.
w+b	crea archivo binario para lectura / escritura.
a+b	añade o crea un archivo binario para lectura / escritura.

Al utilizar archivos, debemos tomar en cuenta ciertas consideraciones. Entre ellas, la forma en que se almacenan sus registros; en otras palabras, si forzosamente su almacenamiento requiere una secuencia (un registro seguido de otro de manera secuencial) o si no importa la secuencia al momento de su almacenamiento (se hace de manera aleatoria). Esto depende sobre todo del tipo de proceso que queremos ejecutar.

Tanto en C como en C + +, se integra una librería de entrada salida: Standard input-output Library que realiza operaciones de entrada / salida. Las declaraciones correspondientes para esta librería se consideran en el archivo de encabezado "stdio.h".

```
#include <stdio.h>
```

Aquí se definen los "streams" estándar: stdin, stdout y stderr, utilizados en la definición de entradas y salidas estructuradas en caracteres y abiertas de manera automática. Usualmente stdin y stderr son direccionadas a pantalla, y stdin al teclado.

Si utilizamos las funciones getchar() y putchar():

```
int getchar()          – lee un carácter desde stdin
int putchar(char c)    – escribe el carácter "c" en stdout
```

Las funciones printf() y scanf(), facilitan la aplicación de algún tipo de formato a los datos de entrada / salida.

Por otra parte, al hablar de un manejo de archivos consideraremos de manera implícita que estos se encuentran en algún medio o soporte de almacenamiento, ya sea de la misma unidad de almacenamiento de la computadora o de otro de los muchos medios que ya existen y que se pueden conectar en los equipos de cómputo. Esto permite su posterior utilización, ya sea para su actualización o solo para su resguardo.

En ejemplos anteriores hemos recurrido a datos que se almacenan en variables utilizadas a lo largo de la ejecución de un proceso. Pero al finalizar este se pierden o no quedan almacenadas en lugar alguno: solo tienen vigencia durante el lapso en que se ejecuta un proceso o una operación. En muchas ocasiones, se hace necesario su almacenamiento; ahí es donde entran los archivos.

Según el proceso y la dirección en que fluyen los datos, podemos identificar o clasificar los archivos de tres formas:

1. Los archivos de entrada son aquellos que se requieren para que un proceso se pueda ejecutar.
2. Los archivos de salida son los que se requieren para almacenar datos o información producto de un proceso.
3. Los archivos de entrada/salida son aquellos que se requieren para leer o almacenar datos o información producto de un proceso.

Cuando se define un archivo de entrada, de salida o de entrada / salida, hay que tomar en cuenta sus características o propiedades al momento de su utilización. Los archivos pueden ser referenciados de acuerdo con las siguientes características y según el flujo que sigan los datos dentro del proceso.

Es importante resaltar que una de las librerías más utilizadas en los programas es la que permite leer y escribir en modo de consola del sistema operativo. En C, la librería está disponible en la cabecera `stdio.h`, que contiene varias funciones para la edición de archivos:

- Acceso a un archivo definido con la característica de solo lectura es decir, que no permite modificaciones, solo su lectura (`r`).
- Acceso a un archivo definido con la característica de modo escritura es decir, se abre para escribir si el archivo no existe, lo crea, en caso de existir, lo sobrescribe (`w`).
- Acceso a un archivo definido con la característica de añadir es decir, el apuntador se posiciona al final del archivo y a partir de ese punto inicia la escritura, en caso de no existir, lo crea (`a`).
- Acceso a un archivo definido con las características de lectura y escritura, en este caso el archivo debe existir (`r+`).
- Acceso a un archivo definido con la característica de lectura y escritura, en este caso se crea un archivo nuevo, o si ya existe se sobrescribe (`w+`).
- Acceso a un archivo con la característica de añadir, lectura y escritura, en este caso el apuntador se posiciona al final del archivo y a partir de ese punto inicia la escritura, en este caso si el archivo no existe, se crea (`a+`).

Es común en programación llamar estructuras estáticas a datos compuestos de datos simples (enteros, reales, caracteres,...) que se utilizan como si fueran un único dato y que ocupan un espacio de memoria de la computadora.

Recordemos entonces las siguientes estructuras estáticas:

- **Arrays.** Una colección de datos del mismo tipo: listas estáticas, matrices y arreglos.
- **Cadenas.** Conjunto de caracteres tratados como un texto, también llamados `strings` (recordar que en C estándar no existe este tipo de dato).
- **Apuntadores o punteros.** Definen variables que contienen posiciones de memoria; se utilizan para apuntar a otras variables.
- **Estructuras.** Datos compuestos de otros de distinto tipo, también llamados registros. Una estructura puede estar compuesta, por ejemplo, de un entero, un carácter y un `array`.

Para la manipulación de archivos en C, se utilizan algunas funciones, como son **FILE**, `fopen`, `fputs` y `fclose`, que nos ayudarán a entender el manejo de archivos.

Comencemos por ejemplificar el uso de estas funciones: escriba el código para abrir y cerrar un archivo de texto en modo lectura(r) llamado "LeeTexto.txt" (véase programa 3.7).

```
// Programa 3.7: programa para abrir y cerrar un archivo de texto
#include <stdio.h>
#include <conio.h>
int main()
{
   FILE *fp;
   fp = fopen("LeeTexto.txt","r");
   fclose (fp);
   return 0;
}
```

El archivo se puede ver como una cadena de caracteres (*string*) almacenada en el disco duro de la computadora o en algún otro medio. Se pueden utilizar diferentes formas y funciones para leer un archivo; aquí tenemos algunas con su respectiva sintaxis:

- `int fgetc(FILE *archivo)`
- `char *fgets(char *buffer, int tamano, FILE *archivo)`
- `size_t fread(void *puntero, size_t tamano, size_t cantidad, FILE *archivo);`

Las funciones propias para el manejo de archivos en general consisten en crear punteros o apuntadores del tipo FILE; para abrir y/o cerrar archivos utilice las funciones `fopen` y `fclose`, respectivamente, y por supuesto las acciones de escribir, leer, borrar o modificar archivos en sus diferentes formas de representación. Para tener un ejemplo, puede verse el programa 3.8.[5]

```
1  // Programa 3.8: programa que lee datos de pantalla
2  // y los almacena en un archivo AutosColor.txt[6]
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <iostream.h>
6  #include <conio.h>
7  int main ()
8  {
9     char Marca[10];
10    char Coloranio_1[10];
11    char Coloranio_2[10];
12    char Coloranio_3[10];
13    char Coloranio_4[10];
14    FILE * fp;
15    fp = fopen ("AutosColor.txt ","w");
16    cout << "Marca de auto: ";
17    cin >> Marca;
18    cout << "Color del anio 2009: ";
19    cin >> Coloranio_1;
20    cout << "Color del anio 2010: ";
21    cin >> Coloranio_2;
22    cout << "Color del anio 2011: ";
23    cin >> Coloranio_3;
24    cout << "Color del anio 2012: ";
25    cin >> Coloranio_4;
26    fputs(Marca,fp);
27    fputs(Coloranio_1,fp);
28    fputs(Coloranio_2,fp);
29    fputs(Coloranio_3,fp);
30    fputs(Coloranio_4,fp);
31    fclose (fp);
32    system ("PAUSE");
33 }
```

[5] Las funciones `cin` y `cout` que se emplearán en este subcapítulo fueron incorporados en C++ y cubren una función equivalente a `scanf` y `printf`, respectivamente. No existe problema en emplearlas si se trabaja en un compilador que cubra a C y C++, aunque conviene aclarar que a pesar de ello se sigue trabajando en el paradigma de la programación estructurada.

[6] En este ejemplo nos permitimos utilizar otras formas de lectura y despliegue de pantalla. El `cin` y el `cout` realizan la función de `scanf` y `printf`, respectivamente.

Como se puede apreciar en el ejemplo anterior, se utilizan las funciones `cin` y `cout`, que si bien es cierto son funciones propias de C++, su utilización en lenguaje C es similar, en este programa se incluyen con fines de referencia a C++, que veremos más adelante

En el lenguaje C

```
printf("Marca de auto: ";);    // en C
scanf("%s",&marca);    // en C
```

`cin` y `cout` son funciones utilizadas para controlar la entrada y salida estándar de la computadora, están incluidas en la biblioteca `<iostream.h>` y se utilizan para enviar y recibir mensajes o variables a la salida estándar (monitor) y entrada estándar(teclado), respectivamente

Para lenguaje C++

```
cout<< "Marca de auto: ";    //en C++
cin>> marca;    // en C++
```

Una vez ejecutado el programa que lee datos de pantalla, se muestra el contenido del archivo "AutosColor.txt" como resultado del proceso.

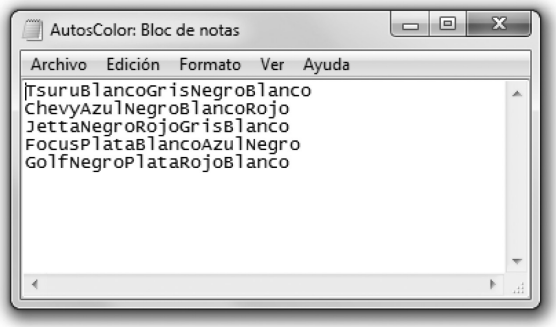


Figura 3.5

La declaración `#include` hace posible la utilización de la librería `stdio.h`, que facilita leer y escribir datos desde la pantalla (consola), aquí se especifica la referencia a un apuntador de tipo `FILE`, que define la cabecera de los archivos. La sintaxis que aplica para este caso es:

```
FILE *fopen(char *nombre, char *modo);
FILE * fopen (const char *nombre, const char *modo);
```

Donde **nombre** representa una cadena de caracteres que define el nombre del archivo, y **modo** define el tipo de acceso al archivo. Para el archivo definido, el modo de acceso es "w", que como se explicó con anterioridad, es un archivo de escritura. En el ejemplo se puede identificar su uso con las instrucciones:

```
FILE * fp;
fp = fopen ("AutosColor.txt ","w");
```

Como se puede ver en el ejemplo, con esta función creamos y a la vez abrimos el archivo "AutosColor.txt", en modo escritura (w). Las variables `cin` y `cout`, permiten leer texto introducido desde la pantalla y almacena los valores correspondientes en cada variable. La función `fputs` escribe el contenido de las variables en un archivo. La sintaxis de la función `fputs` es la siguiente:

```
int fputs(const char texto, FILE *archivo)
```

Esta función permite escribir texto en el archivo definido. Al final de este incluye un carácter que indica un salto de línea, si por alguna causa se genera un error, se envía un EOF. La función `fclose` se utiliza para cerrar los archivos que fueron abiertos durante el proceso. Su sintaxis es la siguiente:

```
int fclose(FILE *archivo);
```

Una buena costumbre al finalizar la ejecución de un proceso es cerrar los archivos que fueron abiertos. Al hacerlo, los datos que se encuentran en memoria son almacenados; esto facilita su utilización posterior para otros programas o procesos. El valor de retorno **cero** indica que el archivo fue cerrado en forma correcta; en caso de error, el valor de retorno es **EOF**. El ejemplo en el programa 3.9 muestra el código que lee el contenido del archivo "AutosColor.txt" generado con el ejemplo anterior.

```
1  // Programa 3.9: programa que lee y muestra en pantalla
2  // el contenido del archivo AutosColor.txt
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <iostream.h>
6  #include <conio.h>
7  int main()
8  {
9     FILE *AutosColor;
10    char caracter;
11    AutosColor = fopen ("AutosColor.txt","r");
12    if (AutosColor==NULL)
13    {
14       cout << "Archivo no existe o no se puede abrir" <<endl;
15       system ("PAUSE");
16       exit(1);
17    }
18    printf("El contenido del archivo AutosColor.txt:\n");
19    caracter=getc (AutosColor);
20    while (feof (AutosColor)==0)
21    {
22       cout << caracter ;
23       cout <<"";
24       caracter=getc(AutosColor);
25    }
26    if (fclose(AutosColor)!=0)
27       cout <<"Error al cerrar archivo" <<endl;
28    cout <<endl;
29    system ("PAUSE");
30 }
```

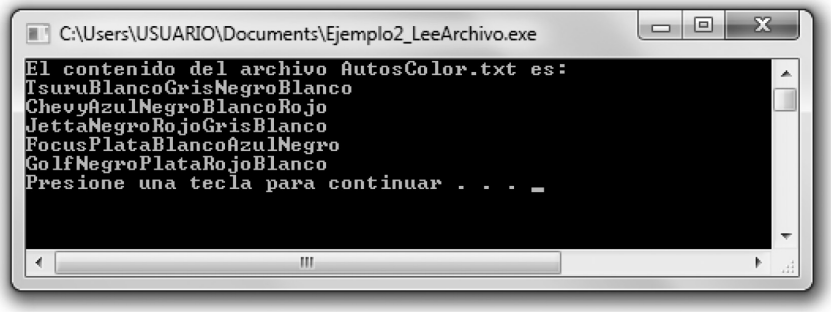


Figura 3.6

Como se puede apreciar, se utiliza la función `getc` para leer caracteres contenidos en el archivo.

La función `fgetc` lee un carácter desde un archivo y el valor de retorno es el carácter leído como un `unsigned char` convertido en `int`. Si no hay ningún carácter disponible, el valor de retorno es EOF. El parámetro es un puntero a una estructura `FILE` del archivo del que se hará la lectura, su sintaxis es la siguiente:

```
int fgetc(FILE *archivo);
```

La función `fgets` lee cadenas de caracteres hasta que lea un retorno de línea. En este caso, el carácter de retorno de línea también es leído. El parámetro `n` nos permite limitar la lectura para evitar desbordar el espacio disponible en la cadena.

```
char *fgets(char *cadena, int n, FILE *archivo);
```

El valor de retorno es un puntero a la cadena leída si se leyó con éxito y es `NULL` si se detecta el final del archivo o si hay un error. Los parámetros son: la cadena a leer, el número de caracteres máximo a leer y un puntero a una estructura `FILE` del archivo del que se leerá.

La función `fputc`, escribe un carácter a un archivo. El valor de retorno es el carácter escrito si la operación fue completada con éxito; en caso contrario será EOF. Los parámetros de entrada son el carácter a escribir, convertido a `int`, y un puntero a una estructura `FILE` del archivo en el que se hará la escritura, su sintaxis es la siguiente:

```
int fputc(int caracter, FILE *archivo);
```

En el programa, la función `feof` regresa un valor verdadero, si es fin de archivo.

La función `feof` permite comprobar si es el final del archivo, si el valor de retorno es diferente de cero, entonces no se ha alcanzado el fin del archivo. El parámetro es un puntero a la estructura `FILE` del archivo que queremos verificar, su sintaxis es la siguiente:

```
int feof(FILE *archivo);
```

Los archivos que se han utilizado son de tipo carácter o texto. Sin embargo, muy a menudo los archivos deben tener cierta estructura. Esto lo podemos lograr con el uso de la función `fprintf` que facilita la escritura de archivos de texto. Su sintaxis es la siguiente:

```
int fprintf(FILE *archivo, const char *formato, ...);
```

La función `fprintf` es similar en parámetros y funcionamiento a `printf`, a diferencia de que la salida es hacia un archivo y no a la pantalla. Esta función permite una gran versatilidad al escribir archivos. Modifique un poco el ejemplo anterior referente al color de los autos, supongamos que ahora deseamos tener almacenados los datos de un inventario de autos, como lo muestra el cuadro 3.5.

Cuadro 3.5

Inventario	Marca	Modelo	Color	Precio
20101	Tsuru	2010	Gris	120 500.80
20131	Chevy	2013	Negro	115 000.50
20111	Jetta	2011	Rojo	160 750.00
20121	Focus	2012	Blanco	180 270.50
20112	Golf	2011	Plata	175 800.50

En un primer análisis de los tipos de datos tendremos que:

Inventario	(entero)
Marca	(cadena de texto)
Modelo	(cadena de texto)
Color	(cadena de texto)
Precio	(valor decimal)

Para este ejemplo los datos se leen uno a uno y al escribirlos se tendrá una separación entre ellos mediante una tabulación. Una posible codificación quedaría en el programa 3.10:

```
1  // Programa 3.10: programa que lee datos de pantalla y los escribe
2  // en el archivo InvAutos.txt
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <iostream.h>
6  #include <conio.h>
7  int main()
8  {
9     int inventario=1;
10    char marca[10];
11    char modelo[10];
12    char color[10];
13    double precio;
14    FILE *InvAutos;
15    InvAutos=fopen("InvAutos.txt","w");
16    if (InvAutos!=NULL) {
17       do {
18          printf("No. de Inventario ? ");
19          scanf("%d",&inventario);
20          if (inventario>0) {
21             printf("Marca ");
22             scanf("%s",marca);
23             printf("Modelo ");
24             scanf("%s",modelo);
25             printf("Color ");
26             scanf("%s",color);
27             printf("precio ");
28             scanf("%lf",&precio);
29             fprintf(InvAutos, "%d\t%s\t%s\t%s\t%lf\n", inventario,
30             marca, modelo, color, precio);
31          }
32       } while(inventario>0);
33       fclose(InvAutos);
34    }
35 }
```

El contenido del archivo "InvAutos.txt" quedaría de la siguiente forma:

Figura 3.7

Las estructuras representan una colección de variables agrupadas bajo un mismo nombre y, como se mencionó al principio, representa un registro dentro de un archivo.

Entonces, si definimos una estructura que se llame "Automovil", tendremos:

```
struct Automovil {
   int inventario;
   char marca[10];
   char modelo[10];
   char color[10];
   double precio;
};
typedef struct automóvil Automovil
```

Esta estructura representa diferentes variables de distintos tipos.La instrucción `struct` define el tipo de estructura y las variables de estructura, inventario, marca, modelo, color, precio, son variables del tipo "Automovil". Por lo que se puede definir la estructura y crear el tipo de datos al mismo tiempo. Por ejemplo:

```
typedef struct {
   int inventario;
   char marca[10];
   char modelo[10];
   char color[10];
   double precio;
} Automovil;
```

En el programa 3.11 se escribe un posible código que crea la estructura de datos para el archivo "Automovil" de nuestro ejemplo:

```
1  // Programa 3.11: programa para crear la estructura
2  // del archivo Automovil
3  #include <stdio.h>
4  #include <iostream>
5  #include <conio.h>
6  void Crea(FILE *Automovil);
7  typedef struct {
8     int inventario;
9     char marca[10];
10    char modelo[10];
11    char color[10];
12    double precio;
13 } Automovil;
14 int main()
15 {
16    FILE * fp;
17    Crea(fp);
18 }
19 void Crea(FILE * fp)
20 {
21    fp = fopen("Automovil.txt","w");
22    {
23    printf("estructura de archivo creada\n");
24    }
25    fclose (fp);
26    system ("PAUSE");
27    return;
28 }
```

El siguiente código permite grabar datos en el archivo "Automovil" de acuerdo con la estructura definida en el ejemplo anterior (programa 3.12).

```
1  // Programa 3.12: grabar datos en el archivo Automovil
2  #include <stdio.h>
3  #include <iostream>
4  #include <conio.h>
5  void Gdatos(FILE *Automovil);
6  struct {
7     int inventario;
8     char marca[10];
9     char modelo[10];
10    char color[10];
11    int precio;
12 } Automovil;
13 int main()
14 {
15    FILE * fp;
16    Gdatos(fp);
17 }
18 void Gdatos(FILE * fp)
19 {
20    fp = fopen("Automovil.txt", "w+");
21    printf("\nNo. de Inventario.......? ");
22    scanf("%d", &Automovil.inventario);
23    printf("\nIndique la marca........? ");
24    scanf("%s", &Automovil.marca);
25    printf("\nIndique el Modelo.......? ");
26    scanf("%s", &Automovil.modelo);
27    printf("\nIndique el Color........? ");
28    scanf("%s", &Automovil.color);
29    printf("\nIndique el Precio.......? ");
30    scanf("%d", &Automovil.precio);
31    fwrite(&Automovil, sizeof(Automovil), 1, fp);
32    fclose(fp);
33    system ("PAUSE");
34    return;
35 }
```

La función `fwrite` permite escribir uno o varios registros de la misma longitud en un archivo. El valor de retorno es el **número de registros escritos**, no el número de bytes. Su sintaxis es la siguiente:

```
size_t fwrite(void *puntero, size_t tamaño, size_t nregistros,
FILE *archivo);
```

En el ejemplo, tenemos:

```
fwrite(&Automovil, sizeof(Automovil), 1, fp);
fwrite(&Registro, sizeof(Registro), 1, alias);
```

Donde:

- `&Registro`. Hace referencia a la estructura creada.
- `sizeof(Registro)`. Determina el tamaño del registro de la estructura.
- `1`. Indica que se grabe un solo registro.
- `alias`. Es el apuntador al cual asignamos la apertura del archivo.

Ahora veamos el código para leer datos del archivo "Automovil" de acuerdo con la estructura con la que fueron grabados (programa 3.13).

```
1  // Programa 3.13: leer datos del archivo Automovil
2  #include <stdio.h>
3  #include <iostream>
4  void Lee(FILE *Automovil);
5  struct {
6     int inventario;
7     char marca[10];
8     char modelo[10];
9     char color[10];
10    int precio;
11 } Automovil;
12 int main() {
13    FILE * fp;
14    Lee(fp);
15 }
16 void Lee(FILE * fp) {
17    fp = fopen("Automovil.txt", "rb");
18    fread(&Automovil, sizeof(Automovil), 1, fp);
19    printf("\No.Invent.\tMarca \tModelo \tColor \tPrecio \n \n");
20    printf("\t%d \t%s \t%s \t%s \t%d \n \n",
   Automovil.inventario, Automovil.marca, Automovil.modelo,
   Automovil.color, Automovil.precio);
21    fclose(fp);
22    system ("\nPAUSE");
23    return;
24 }
```

Figura 3.8

La función `fread` permite leer un archivo con registros de una misma longitud. El valor de retorno es el **número de registros leídos**, no el número de bytes. Los parámetros son: un puntero a la zona de memoria donde se almacenarán los datos leídos, el tamaño de cada registro, el número de registros a leer y un puntero a la estructura `FILE` del archivo del que se hará la lectura. Su sintaxis es la siguiente:

```
size_t fread(void *puntero, size_t tamaño, size_t nregistros,
FILE *archivo);
```

En el ejemplo tenemos:

```
fread(&Automovil, sizeof(Automovil), 1, fp);
fread(&Registro, sizeof(Registro), 1, alias);
```

Que es muy similar a `fwrite` y donde:

- `&Registro`. Hace referencia a la estructura creada.
- `sizeof(Registro)`. Determina el tamaño del registro de la estructura.
- `1`. Indica que se grabe un solo registro.
- `alias`. Es el apuntador al cual asignamos la apertura del archivo.

La función `fputs` escribe una cadena en un archivo. No se añade el carácter de retorno de línea ni el carácter nulo final. El valor de retorno es un número no negativo o EOF en caso de error. Los parámetros de entrada son la cadena a escribir y un puntero a la estructura `FILE` del archivo donde se realizará la escritura.

```
int fputs(const char *cadena, FILE *stream);
```

La función `fprintf` dirige la salida de datos a un archivo, su sintaxis es la siguiente:

```
int fprintf(FILE *Archivo, const char *formato, ...);
```

La función `fscanf` permite la entrada de datos desde un archivo en lugar del teclado.

```
int fscanf(FILE *archivo, const char *formato, ...);
```

EJERCICIOS SUGERIDOS

Casos por resolver sobre registros y archivos

1. En un archivo de texto "entrada.txt" existe una lista de datos de tipo decimal. Se requiere que se ordenen esos números de mayor a menor bajo cualquier algoritmo y el resultado sea colocado en un archivo "salida.txt".
2. Existe un archivo de estudiantes llamado "estudiantes.txt" que contiene en cada línea el nombre y el promedio global de cada uno de los estudiantes de esa escuela. Ambos datos están separados por un punto y coma. Se requiere que se lea ese archivo y se despliegue en pantalla la información siguiente:
   - Número total de estudiantes;
   - Número de estudiantes con promedio menor a 6;
   - Número de estudiantes con promedio igual o mayor a 6 y menor a 8;
   - Número de estudiantes con promedio igual o mayor a 8 y menor a 9;
   - Número de estudiantes con promedio igual o mayor a 9.
3. El siguiente ejercicio pretende realizar de la manera muy básica el famoso juego de "serpientes y escaleras" (aunque para un solo jugador). En cada turno se simula la tirada de un dado. El jugador avanza el número de lugares indicado por el dado, pero al llegar a esa casilla puede "caer" en una serpiente o en una escalera: las escaleras aumentan el avance y las serpientes lo disminuyen. Gana quien caiga con exactitud en el lugar número 50. Para hacerlo más interesante, en el archivo "serpientes.txt" se encuentran las serpientes del juego y en "escaleras.txt" se hallan las escaleras. El tablero más sencillo puede quedar como sigue:

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Mientras que el archivo "serpientes.txt" quedaría como sigue:

```
22    5
33    2
47    32
```

Lo que debe interpretarse de la siguiente forma. Si el jugador cae en la casilla 22 retrocede a la 5. Sucede algo similar con la casilla 33, pues se baja a la 2; y en la 47, en cuyo caso se traslada a la 32.

3.3 Subrutinas

El concepto de subrutina

La idea básica de modularidad es la creación de módulos lo más independientes posible, con entradas y salidas definidas con claridad. "Un método de construcción de software es modular, por tanto, ayuda a los diseñadores a producir sistemas de software a partir de elementos autónomos interconectados mediante una estructura simple y coherente."[7]

La idea de modularidad se puede encontrar en áreas muy diversas. Por ejemplo, los aparatos electrónicos se basan en componentes, cada uno de ellos con una entrada y una salida definidas en forma clara (bocinas, audífonos, etc.). Una pieza que cumpla las especificaciones marcadas podría reemplazar a otra y el sistema en su conjunto seguiría funcionando.

REFLEXIÓN 3.1

El concepto abstracto de modularidad, más allá del código

Conviene hacer hincapié con los estudiantes en el concepto de modularidad "más allá del software", pues esto crea una idea abstracta de componente que los prepara hacia la programación orientada a objetos y hacia otras áreas de conocimiento y laborales. En particular, se sugiere hacer énfasis en el aspecto de reutilización, en general descuidado en los cursos de programación.

La modularidad facilita cubrir los criterios de calidad de software de extensibilidad y reutilización. En términos más prácticos:

- Permite el trabajo en paralelo de varios desarrolladores.
- Simplifica el desarrollo de sistemas, pues es más fácil realizar una subrutina que el sistema completo, tanto en lo referente al volumen como en la complejidad.
- Simplifica el mantenimiento de sistemas, al hacer más fácil las modificaciones.
- Permite el desarrollo por capas, en el cual solo unas rutinas dependen del hardware, con lo cual es más fácil caminar hacia la portabilidad.
- Prevé la redundancia, pues evita repetir código que tiene el mismo propósito.

Una subrutina recibe uno o más valores (llamados parámetros), realiza una labor específica y devuelve, opcional, un valor de un tipo específico (véase figura 3.9).

Cada uno de los parámetros que recibe una rutina debe ser de un tipo específico y su nombre solo tiene efectos para esta. En otras palabras, es independiente de la subrutina que la llama. El orden con que se declaran y envían los parámetros debe ser el mismo, pues la rutina los identificará a través de su posición. En forma similar, el valor que devuelve es de un determinado tipo. Si la subrutina no retornara nada

[7] Meyer, Bertrand. *Construcción de software orientado a objetos*. Prentice-Hall, segunda edición, España, 1999, p. 40.

Figura 3.9

se debe indicar de manera explícita a través de la palabra reservada `void`. La palabra `return` sirve para retornar el valor desde la rutina hacia el módulo que la llamó.

Cada rutina puede tener variables locales, las cuales solo tienen efecto para la propia rutina. Se crean al comenzar esta y se destruyen al terminar. Ninguna otra rutina las puede ver aparte de la propia subrutina.

En lenguaje C, las rutinas se suelen agrupar en librerías. Cada librería cubre un área específica (por ejemplo, manejo de cadenas o funciones matemáticas).

Conviene hacer algunas puntualizaciones antes de continuar:

- Los términos rutina y subrutina deben considerarse como sinónimos.
- Por cuestión didáctica se suele decir que el programa principal llama a la subrutina, aunque en realidad una subrutina puede ser llamada por otra subrutina. De hecho, el programa principal (*main*) es una subrutina.
- El sobrenombre que una rutina le asigna a una variable puede ser diferente o igual al que le asignó a su vez el programa principal.

Un primer ejemplo de código

A manera de ejemplo, revise el programa 3.14 que contiene dos rutinas: la rutina principal (*main*) y la subrutina de sumatoria.

```
// Programa 3.14: sumatoria de un número aplicando subrutina por ciclos
#include <conio.h>
#include <stdio.h>
int sumatoria (int a);
int main() {
   int dato, resultado;
   printf("Este programa obtiene la sumatoria de un número.\n");
   printf("Por favor, teclee el número: ");
   scanf("%d", &dato);
   resultado = sumatoria(dato);
   printf("La sumatoria de %d es %d\n\n", dato, resultado);
   printf("Oprima cualquier tecla para continuar...");
   getch();
}
int sumatoria (int a) {
   int i, suma = 0;
   if (a == 0)
      return 0;
   else {
      for (i = a; i > 0; i--)
         suma += i;
      return(suma);
   }
}
```

Cabe hacer hincapié en que las variables locales solo pueden ser accedidas por la propia rutina. En nuestro caso:

Rutina	Variables locales que usa
main	**dato, resultado**
sumatoria	**i, suma**

El parámetro se recibe a partir de la variable a. Observe que, en este caso, la rutina principal es la que lee los datos del usuario, mientras que la rutina recibe su información a través de los parámetros y no realiza ningún proceso de entrada-salida. En términos generales, esta es la estructura más recomendable para el manejo de rutinas.

Ocultamiento de la información

El diseñador de cada módulo debe seleccionar un subconjunto de propiedades del módulo como información oficial sobre este para ponerla a disposición de los autores de módulos clientes.

Para entender los beneficios de la ocultación de la información partimos del hecho de que un módulo es conocido a través de una descripción oficial que señala sus propiedades públicas. Todo lo demás se desconoce. Cualquier uso de este módulo y cualquier prueba de su funcionamiento deben basarse solo en estas propiedades públicas. Por lo regular, las propiedades se refieren a la función del módulo y los datos de entrada y salida que requiere. El proceso interno de su tarea (el "cómo" lo hace) debe quedar como parte privada y nadie debe hacer uso de ella en forma directa.

Si se cumplen estas reglas se puede modificar el proceso interno (algoritmos, variables locales, etc.) sin que se afecte en modo alguno al sistema en su conjunto.

EJEMPLO

La rutina de raíz cuadrada en lenguaje C maneja la siguiente descripción oficial:[8]

```
 #include "math.h"
double sqrt (double num);
La función sqrt() devuelve la raíz cuadrada de num.
Si se llama a esta función con un argumento negativo,
se producirá un error de dominio
```

Indica que se encuentra en la librería math.h, que requiere un número de tipo double como entrada y devolverá un número también de tipo double. El parámetro de entrada tiene que ser mayor o igual a cero. Los demás detalles quedan ocultos: si usa variables locales, el algoritmo que emplea, etc. Esta parte privada no se dice ni tiene porqué conocerla quien hace uso de este módulo.

Observe otra variante de la rutina sumatoria. Como no se modifica ninguna de las partes que ligan con otras rutinas, no se tiene que cambiar ninguna línea del programa principal.

```
int sumatoria (int a) {
   return a * (a + 1) / 2;
}
```

En este punto debemos hacer una aclaración. Nos hemos saltado una validación elemental. ¿Qué sucedería si la variable que envía el programa principal es negativa? No podría hacerse el cálculo. Sin embargo, nuestra subrutina lo realiza. El aspecto de validación en subrutinas se abordará más adelante, cuando se plantee cómo delimitar las subrutinas.

[8] Schildt, Herbert C. *Manual de bolsillo*. McGraw-Hill, España, 1992, p. 180.

Recursividad

En este capítulo se han visto dos versiones de una subrutina sumatoria, que reproducimos a continuación:

```
// versión 1, utilizando ciclos
int sumatoria (int a) {
   int i, suma = 0;
   if (a == 0)
      return 0;
   else {
      for (i = a; i > 0; i--)
         suma += i;
      return(suma);
   }
}

// versión 2, empleando una fórmula directa
int sumatoria (int a) {
   return a * (a + 1) / 2;
}
```

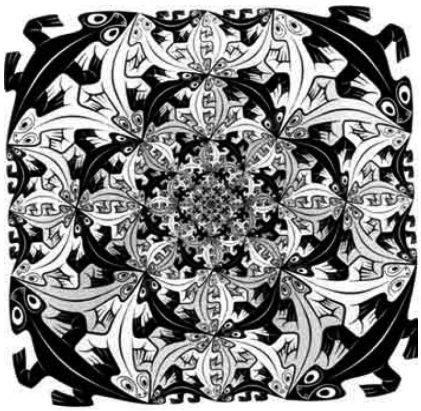


Figura 3.10 La recursividad en una pintura de Escher.

Presentaremos una tercera versión a través de una llamada recursiva; es decir, de una rutina que se llama a sí misma, aunque con un valor distinto en los parámetros. Esta llamada recursiva puede invocarse otra vez a sí misma y así de manera sucesiva, pero para que no se dé un círculo infinito en algún punto se dará un resultado preciso y se retornará el valor a la última llamada recursiva que se hizo, que a su vez hace lo propio con la penúltima invocación… hasta llegar a la primera invocación, que ya puede dar el resultado definitivo.

A nivel interno y de resolución de algoritmos, las llamadas recursivas son un campo fascinante y laborioso, pues se encuentran en muy diferentes problemas y formas (véase figura 3.10). Por fortuna, en los lenguajes actuales los detalles internos de las llamadas recursivas son "transparentes" para el programador, quien se concentra en la forma adecuada de la llamada recursiva y en la condición que da un resultado y evita una llamada recursiva más.

La sumatoria con la llamada recursiva queda programada de la siguiente forma:

```
// versión 3, empleando recursividad
int sumatoria (int a) {
   if (a == 1)
      return 1;
   else
      return a + sumatoria (a-1);
}
```

Insistimos: ¡no se deje intimidar por la recursividad! Piense bien en lo que hará la llamada recursiva y cómo se romperá la recursividad. Eso es todo.

Este punto de vista tiene su razón de ser por el fundamento conceptual de la recursividad, que se haya en el tema de la inducción matemática. Suponga que usted logra demostrar dos situaciones:

a) Una afirmación es cierta para un caso inicial.

b) Suponga que una determina fórmula es cierta para el caso n, se puede demostrar que es cierta para el caso n + 1.

Si ambas situaciones se cumplen, entonces esa fórmula trabajará en forma correcta para todos los números, hasta el infinito. De hecho, este es el planteamiento que se aborda en el tema de inducción matemática en los libros dedicados al tema.

Por ejemplo, algún usuario da las dos afirmaciones siguientes: a) la sumatoria del número 0 es cero; b) suponga que se tiene la sumatoria de un número expresada por sumatoria(n), entonces la sumatoria del número n + 1 será sumatoria(n) + (n + 1). Bastará que usted lleve estas dos afirmaciones a una subrutina de algún de lenguaje de programación, cuyo seudocódigo quedará como sigue:

```
devuelve sumatoria (n) // validar previamente que n >= 0
  si n es igual a cero
     devuelve 0
  sino
     devuelve n + sumatoria(n-1)
```

Como se podrá notar, quien codifica debe fijarse en tres aspectos: 1) validar que el dato sea consistente (en nuestro caso, que n sea mayor o igual a cero); 2) la llamada recursiva, y 3) el caso que da por terminada la recursividad.

Apuntadores, parámetros por valor y por referencia

Hasta este momento, el manejo de las subrutinas parece muy sencillo, pero suponga la siguiente situación: se desea saber el número menor y mayor de tres datos. Podría realizarse una rutina que devolviera el número menor y otra que hiciera algo similar con el número mayor (de hecho, así trabajan las hojas de cálculo). ¿Sería posible hacerlo con una única rutina?

Para resolver este tipo de escenarios y otros más, se encuentran los parámetros por referencia.

Antes de dar un ejemplo es indispensable ubicar con claridad la diferencia entre parámetros por valor y parámetros por referencia.

- **Parámetro por valor.** El programa principal llama a la subrutina y le pasa un dato. La subrutina hace una copia y le pone un sobrenombre. En todo momento trabaja sobre la copia.
- **Parámetro por referencia.** El programa principal llama a la subrutina y le pasa la dirección del dato. La subrutina recibe la dirección y le pone su propio sobrenombre. En todo momento trabaja con la dirección del dato, por lo cual cualquier modificación que realice puede afectar en forma directa a la variable del programa principal.

Si se permite una analogía burda: el parámetro por valor pasa un folder con fotocopias de un escrito y le cambia el nombre en la pestaña, mientras que el parámetro por referencia pasa un folder con los documentos originales y solo le cambia el nombre de la pestaña, por lo cual cualquier modificación se realizará de manera directa sobre el folder original (véase figura 3.11).

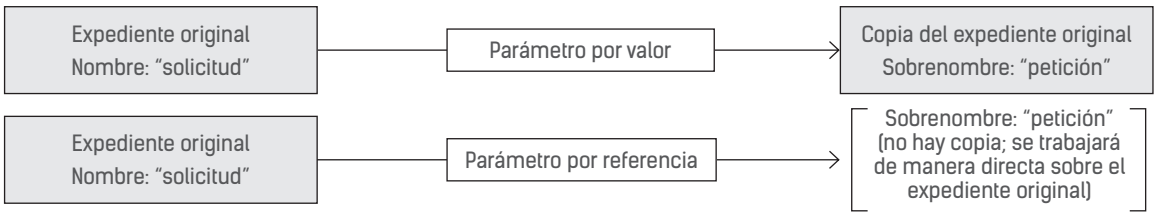


Figura 3.11 Diferencia conceptual entre parámetros por valor y parámetros por referencia.

¿De qué manera se logra conocer si un parámetro es por valor o por referencia? Esta diferenciación ya se ha empleado sin que nos hayamos dado cuenta:

```
scanf ("%d", &dato); // parámetro por referencia
printf("La sumatoria es %d", resultado); // parámetro por valor
```

El *ampersand* (`&`) significa que se envía un parámetro por referencia. La subrutina, por su parte, debe indicar que se recibe un parámetro por referencia con un asterisco (*variable).

Desde un punto de vista técnico, `&x` significa la dirección de memoria en la cual se encuentra la variable `x`. Cuando se recibe el dato en la subrutina se emplea `*sobrenombre`. De esta forma, `sobrenombre` es la dirección de la memoria y `*sobrenombre` debe interpretarse como la variable que se encuentra en la dirección de memoria señalada por `sobrenombre`.

A esta altura, el lector debe sentirse algo confundido. Esperamos aclarar varios aspectos a través del programa 3.15.

```
// Programa 3.15: devolver el mayor y menor de 3 números.
#include <conio.h>
#include <stdio.h>
#include <math.h>
void minmax (float x, float y, float z, float *menor, float *mayor);
int main() {
   float a, b, c, min, max;
   printf("Bienvenido.\n\n");
   printf("Este programa obtiene el mayor y menor de 3 números.\n");
   printf("Teclee los 3 datos separados por un espacio: ");
   scanf("%f %f %f", &a, &b, &c);
   minmax (a, b, c, &min, &max);
   printf("El menor y el mayor son: %4.1f %4.1f", min, max);
   printf("\n\nOprima cualquier tecla para terminar...");
   getch();
}
void minmax (float x, float y, float z, float *menor, float *mayor) {
   float minimo = x, maximo = x;
   if (y < minimo) minimo = y;
   if (y > minimo) maximo = y;
   if (z < minimo) minimo = z;
   if (z > maximo) maximo = z;
   *menor = minimo;
   *mayor = maximo;
}
```

Como podrá observar, las variables a, b y c son declaradas en el programa principal y enviadas por valor a la subrutina `minmax`, la cual hace una copia de cada una de ellas y les pone el nombre de x, y, z, respectivamente. En todo momento trabaja sobre copias. Cualquier alteración a las variables x, y, z no modifica el contenido de a, b y c.

Intentaremos ilustrar la situación anterior con la figura 3.12. Para facilitar las cosas, suponga que el usuario tecleó los siguientes valores: a = 6, b = 7 y c = 5. Considere que `min` quedó alojada en la dirección **E3D1:0703** y `max` en **E1D9:0370**.

```
// llamada del programa principal
float a, b, c, min, max;
minmax (a, b, c, &min, &max);
```

a=6	b=7	c=5	min= aún sin valor alojado en E3D1:0703	max= aún sin valor alojado en E1D9:0370
Cuando se invoca a la subrutina, el programa principal cede el control y transfiere los valores que corresponden según los parámetros declarados. Sus variables locales se vuelven inaccesibles desde la subrutina.				

```
// código de la subrutina
void minmax (float x, float y, float z, float *menor, float *mayor)
float minimo = x, maximo = x;
*menor = minimo;
*mayor = maximo
```

x=6	y=7	z=5	minimo y maximo	menor=E3D1:0703	mayor=E1D9:0370

Valores recibidos desde el programa principal y copiados a las variables x, y, z.			Variables locales inaccesibles para el programa principal, pues son creadas cuando se ejecuta la subrutina y desaparecen antes de que se devuelva el control al programa principal.	Las variables menor y mayor guardan las direcciones de memoria en donde se encuentran las variables min y max del programa principal. Por ello cuando se indican las instrucciones
				
				*menor = minimo;
				*mayor = maximo;
				
				se está alterando a `min` y a `max`.	

Figura 3.12 Parámetros por valor y parámetros por referencia a nivel código.

Las variables `min` y `max` del programa principal son transferidas por referencia a la subrutina `minmax`. La subrutina recibe las direcciones de memoria y hace una copia de dichas direcciones en `menor` y `mayor`. Por eso cuando afecta al dato señalado por `menor`, en realidad cambia el valor de la variable `min` del programa principal.

En este punto vale la pena señalar que las direcciones de memoria que señalan a una variable específica son conocidas como apuntadores.

¿Cómo delimitar las subrutinas?

Uno de los puntos más interesantes es delimitar qué pertenece a la subrutina y cómo se comunica esta con el programa principal. Cuando se hace esta delimitación, es indispensable pensar en la facilidad de construir el código. Pero, a la par, en reutilizar la subrutina en otros posibles programas.

REFLEXIÓN 3.2

Separar a los estudiantes que hacen la subrutina y el programa principal

En este nivel es recomendable que los programas basados en subrutinas no sean realizados por un único estudiante. Por el contrario, un alumno debería hacer el programa principal y otro la subrutina. Solo así se visualiza como primordial la delimitación de la subrutina: su entrada, su salida y las validaciones que necesita realizar.

EJEMPLO

Para explicar la coordinación entre las subrutinas y el programa principal utilizaremos el ejemplo de obtener el área de un triángulo a partir de sus lados, presentado en otra sección de este mismo libro.

Requerimiento:

Hacer un programa que reciba los tres lados de un triángulo y devuelva su área. El área queda determinada bajo la siguiente fórmula:

$$s = (a + b + c) / 2$$

$$\text{área} = \sqrt{s * (s - a) * (s - b) * (s-c)}$$, donde a, b y c son las longitudes de los lados.

Ejemplo: si los lados valieran 6, 8 y 10, s valdría 16 y el área valdría 24.

Código:

declaración de librerías	`// Programa área de un triángulo a partir de sus 3 lados.`
	`#include <conio.h>`
	`#include <stdio.h>`
	`#include <math.h>`
	`main() {`

declaración de variables	`float a, b, c, s, area;`

lectura de datos	`printf("Bienvenido.\n\n");`
	`printf("Este programa obtiene el área de un triángulo.\n");`
	`printf("Teclee sus datos separados por un espacio: ");`
	`scanf("%f %f %f", &a, &b, &c);`

validación	`// previamente se validará que sea posible construir el triángulo`
	`if ( (a > b + c) || (b > a + c) || (c > a + b))`
	`printf("No puede formarse un triangulo de esas dimensiones.\n");`
	`else {`

cálculo	`s = (a + b + c ) / 2;`
	`area = sqrt( s * (s - a) * (s - b) * (s - c));`
	`printf("\n");`
	`printf("El area del triangulo es: %4.1f \n", area);`
	`}`

despliegue del resultado	`printf("\n\n Oprima cualquier tecla para terminar...");`
	`getch();`
	`}`

Pruebas aplicadas:

Datos del usuario	Resultado esperado
6 8 10	El área del triángulo es de 24.0.
3 5 9	No puede formarse un triángulo de esas dimensiones.

¿Qué elementos deben ir en el programa principal y cuáles en la subrutina?

declaración de librerías	En el programa principal, pues de otra forma no se ejecutaría.
declaración de variables	Cada parte debe tener las variables locales que requiera.
lectura de datos	Por lo regular, lo debe tener el programa principal y enviar los datos que necesita a la subrutina a través de los parámetros.
validación	Al menos en alguno de los dos. En nuestro caso, lo hará la subrutina y si detecta algún error devolverá un -1.
cálculo	Lo hará la subrutina y lo transmitirá al programa principal a través de un return.
despliegue del resultado	Por lo regular, lo hará el programa principal.

REFLEXIÓN 3.3

Nunca olvide las validaciones al hacer subrutinas

Las validaciones deben ir al menos en el programa principal o en la subrutina. Si las va a omitir en una de ellas, asegúrese de que la contraparte la va a tener. A menos, claro está, que desee que el programa "truene" con consecuencias impredecibles.

Con esta delimitación de cada parte, nos queda una última pregunta. Suponga que cada quien trabaja por separado, ¿cómo podría estar seguro de que su trabajo se ejecuta en forma correcta? Quien haga la subrutina, debe simular el programa principal de la manera más sencilla; quien haga el programa principal, simulará a la subrutina. Si se nos permite una analogía burda, quien fabrique los focos tendrá un probador para ver si prenden; quien haga la instalación eléctrica, tendrá un foco probador. Al juntar ambas partes se reducirá a niveles muy bajos la probabilidad de error en las pruebas de integración.

La codificación del programa 3.16 es la que tendría el encargado de hacer la subrutina. En cursivas está el código que se tuvo que poner para probarla.

```
// Programa 3.16: área de un triángulo a partir de sus 3 lados.
// Versión con la rutina completa, con su prueba.
#include <conio.h>
#include <stdio.h>
#include <math.h>
float area (float lado1, float lado2, float lado3);
int main() {
   printf("Resultados de prueba de la rutina.\n");
   printf("Prueba con 3 4 5 => %6.1f \n", area (3,4,5));
   printf("Prueba con 1 2 7 => %6.1f \n", area (1,2,7));
   printf("\n\nOprima cualquier tecla para terminar...");
   getch();
}
float area (float lado1, float lado2, float lado3) {
   float s, area;
   if ( (lado1 > lado2 + lado3) || (lado2 > lado1 + lado3)
         || (lado3 > lado1 + lado2))
      return -1;
   else {
      s = (lado1 + lado2 + lado3) / 2;
      area = sqrt( s * (s - lado1) * (s - lado2) * (s - lado3));
      return area;
   }
}
```

El código que tiene las instrucciones de la rutina principal se coloca en el programa 3.17. Observe que se puso una subrutina con un único `return` para poder verificar que el programa principal se logra ejecutar sin errores.

```
// Programa 3.17: área de un triángulo a partir de sus 3 lados.
// Versión que tiene el programa principal terminado y simula la rutina.
#include <conio.h>
#include <stdio.h>
#include <math.h>
```

```
float area (float x, float y, float z);
int main() {
   float a, b, c, resultado;
   printf("Bienvenido.\n\n");
   printf("Este programa obtiene el área de un triángulo ");
   printf("a partir de sus lados.\n");
   printf("Teclee sus datos separados por un espacio: ");
   scanf("%f %f %f", &a, &b, &c);
   // previamente se validará que sea posible construir el triángulo
   resultado = area(a, b, c);
   if (resultado < 0)
      printf("No puede formarse un triángulo de esas dimensiones.\n");
   else {
      printf("El área del triángulo es: %4.1f \n", resultado);
   }
   printf("\n\nOprima cualquier tecla para terminar...");
   getch();
}
float area (float x, float y, float z) {
   return 6;
}
```

Ahora, ¡a conjuntar los dos códigos! La subrutina real debe sustituir a la subrutina simulada que puso en el programa principal y debe revisarse el encabezado de la subrutina que se puso después de los `include`. De manera ideal, deben poder correr sin tocar una sola línea de código (véase programa 3.18). En la práctica, se llegan a dar inconsistencias mínimas que casi siempre se solucionan con rapidez.

```
// Programa 3.18: área de un triángulo a partir de sus 3 lados.
// Versión final: programa utilizando una subrutina
#include <conio.h>
#include <stdio.h>
#include <math.h>
float area (float lado1, float lado2, float lado3);
int main() {
   float a, b, c, resultado;
   printf("Bienvenido.\n\n");
   printf("Este programa obtiene el área de un triángulo ");
   printf("a partir de sus lados.\n");
   printf("Teclee sus datos separados por un espacio: ");
   scanf("%f %f %f", &a, &b, &c);
   // previamente se validará que sea posible construir el triángulo
   resultado = area(a, b, c);
   if (resultado < 0)
      printf("No puede formarse un triángulo de esas dimensiones.\n");
   else {
      printf("El área del triángulo es: %4.1f \n", resultado);
   }
   printf("\n\nOprima cualquier tecla para terminar...");
   getch();
}
float area (float lado1, float lado2, float lado3) {
   float s, area;
   if ( (lado1 > lado2 + lado3) || (lado2 > lado1 + lado3)
         || (lado3 > lado1 + lado2))
      return -1;
   else {
```

```
      s = (lado1 + lado2 + lado3) / 2;
      area = sqrt( s * (s - lado1) * (s - lado2) * (s - lado3));
      return area;
   }
}
```

REFLEXIÓN 3.4

¿Hasta dónde llevar la concisión del código?

A continuación dos versiones equivalentes de la subrutina que hace la sumatoria, ambas basadas en la fórmula de sumatoria y con la validación correspondiente:

```
int sumatoria (int a) {
   int res;
   if (a < 0)
      res = -1;
   else
      res = a * (a + 1) / 2;
   return res;
}

int sumatoria (int a) {
   int res = (a<0) ? -1 : a * (a + 1) / 2;
   return res;
}
```

Una es más sencilla; la otra es más concisa. ¿Cuál debe fomentarse entre los estudiantes? ¿Es preferible un código más claro o uno más corto, aunque tal vez críptico? Sin duda, un aspecto que merece reflexión.

Hacer nuestras propias librerías

¿Qué sucedería si vamos a utilizar una subrutina en diferentes archivos? Convendría tenerla en un lugar separado para llamarla a nuestros programas a través de un `include`. Para hacerlo, basta guardar el archivo como `.h` y compilarlo.

Se tienen dos opciones para guardarlo:

a) En el directorio definido en el mismo compilador para las demás librerías, en cuyo caso se llamará con #`include <archivo.h>` (donde `archivo.h` es el nombre del archivo).
b) En el directorio del programa que lo usará. Si es así se invocará con `#include "archivo.h"`.

Los archivos quedan de la siguiente manera:

```
// archivo con nombre sumatoria.h
int sumatoria (int a) {
   if (a < 0)
      return -1;
else
   return a * (a + 1) / 2;
}
```

```
// archivo con extensión .cpp
#include <conio.h>
#include <stdio.h>
#include "sumatoria.h"
main() {
   int dato, resultado;
   printf("Este programa obtiene la sumatoria de un número.\n");
   printf("Por favor, teclee el número: ");
   scanf("%d", &dato);
   resultado = sumatoria(dato);
   printf("La sumatoria de %d es %d\n\n", dato, resultado);
   printf("Oprima cualquier tecla para continuar...");
   getch();
}
```

Criterios y reglas para establecer la modularidad

Hasta aquí hemos trabajado con una forma intuitiva para decidir qué debe quedar codificado en la subrutina. Conviene a esta altura tener un acercamiento formal a reglas que nos permitan llegar a una estructura modular. El autor Bertrand Meyer enuncia cinco criterios que debe cumplir la modularidad.[9]

REFLEXIÓN 3.5

La modularidad: un concepto descuidado en las aulas

La modularidad es, sin duda, uno de los temas más descuidados en las clases de programación. Es cierto que los estudiantes dividen sus programas y crean subrutinas, pero casi siempre lo hace un mismo alumno sin pensar en estándares acordados por un equipo de desarrollo o una posible reutilización del código.

Una de las estrategias didácticas que deberían abordarse es que un estudiante diera mantenimiento a un programa realizado por otro compañero (tal vez de semestres anteriores). También puede funcionar la utilización de librerías colectivas.

El planteamiento de estas estrategias queda fuera del alcance del presente libro, pero al menos deseamos manifestar que la modularidad solo se aprende si se interactúa con ella y eso nunca sucederá con programadores solitarios que hacen programas "desde cero".

Descomposición modular. Un método de construcción de software satisface la descomposición modular si ayuda en la tarea de descomponer el problema de software en un pequeño número de subproblemas menos complejos, interconectados mediante una estructura sencilla, y lo bastante independientes para permitir que el trabajo futuro pueda proseguir por separado en cada uno de ellos.

Una de las formas más utilizadas para abordar esta descomposición es mediante el diseño descendente (*top-down*), en el cual se parte de una abstracción del funcionamiento del sistema y se divide a manera de un árbol que crece hacia abajo. Este sistema tiene una representación muy similar a la de un organigrama.

Un software rompe este principio si las partes son demasiado dependientes entre ellas. Por ejemplo, si existiera un módulo que iniciara todas las variables del sistema (todos dependerían de él en gran medida). Para no crear una dependencia fuerte entre los módulos es necesario reducir al mínimo el número de variables globales. De hecho, en el método orientado a objetos cada módulo es responsable de la inicialización de sus propias estructuras de datos.

[9] Los criterios y reglas mencionados en esta sección se sintetizaron del libro: Meyer, Bertrand. *Construcción de software orientado a objetos*. Prentice-Hall, segunda edición, España, 1999, pp. 40-51.

Composición modular. Un método satisface la composición modular si favorece la producción de elementos de software que se puedan combinar con libertad unos con otros para producir nuevos sistemas, quizá en un entorno bastante diferente de aquel en que fueron desarrollados al principio.

La composición está relacionada en forma directa con el objetivo de la reutilización: la meta es encontrar formas de diseñar elementos de software que lleven a cabo tareas bien definidas y que sean utilizables en diferentes contextos. Este criterio refleja un viejo sueño: transformar el proceso de diseño de software en una actividad de construcción 'de cajas', de modo que se puedan construir programas mediante la combinación de elementos prefabricados.

Las librerías reflejan esta filosofía. Son subprogramas con propósitos, entradas y salidas definidos con claridad que se combinan con relativa facilidad para producir programas más complejos.

Hay un punto fino que por lo común pasa desapercibido. Cuando se divide un problema mediante la técnica de *top-down* o cualquier otro método, casi nunca se piensa que los submódulos puedan combinarse con libertad para producir nuevos sistemas. Cubrir ambas ideas no es un problema trivial.

Comprensibilidad modular. Un método favorece la comprensibilidad modular si ayuda a producir software en el cual un lector humano puede entender cada módulo sin tener que conocer los otros o, en el peor de los casos, al tener que examinar solo unos pocos de los módulos restantes.

El impacto principal de este criterio se da en el proceso de mantenimiento, aunque también puede influir de manera muy significativa en las tareas de soporte y en el manejo cotidiano. Es muy difícil hacer cualquier cambio o entender un módulo si para su manejo se requiere rastrear elementos de otras partes del sistema. Una situación similar se da cuando el capítulo de un libro está entrelazado con otros capítulos de tal forma que es imposible comprenderlo por separado.

Continuidad modular. Un método satisface la continuidad modular si en la arquitectura de software que produce, un pequeño cambio en la especificación de un problema solo provoca cambios en un módulo o en un pequeño número de módulos.

Un software rompe este principio si un pequeño cambio en las especificaciones implica grandes esfuerzos o grandes tareas de adecuación.

Se avanza mucho en este criterio si el usuario puede modificar cualquier especificación susceptible de cambiar a lo largo de la vida del sistema. Por citar un ejemplo: un sistema de control de salones debería permitir que el usuario dé de alta nuevas aulas o ponga alguna en estatus de mantenimiento.

A nivel de programación también deben preverse posibles cambios. De ahí que se usen constantes cuando el caso lo amerite y muchos de los datos internos del sistema se manejen como parámetros relativamente fáciles de cambiar. Por ejemplo, los estilos de letra para un sitio web se colocan en un archivo de clases. Un contraejemplo: dejar direcciones físicas dentro del código.

Protección modular. Un método satisface la protección modular si produce arquitecturas en las cuales el efecto de una situación anormal que se produzca dentro de un módulo durante la ejecución queda confinado a dicho módulo o en el peor de los casos se propaga solo a unos pocos módulos vecinos.

Errores como fallos de hardware, entradas erróneas o el agotamiento de recursos necesarios (por ejemplo: el espacio de memoria) deben afectar lo menos posible.

En ocasiones la protección modular se consigue si se piensa como "abogado del diablo"; es decir, en el peor de los casos: ¿Qué sucede si una tabla de la base de datos se corrompiera? ¿Qué sucede si se "cae" la red? ¿Qué sucede si la fecha que pusieron en el archivo está mal? Dar una alternativa satisfactoria a esos eventos hace que el sistema disminuya en gran medida el impacto de alguna situación anormal. En este sentido, debe cuidarse que cada módulo que introduzca datos sea también responsable de verificar su validez.

Un ejemplo: cuando un determinado sistema de nómina detecta un error graba los datos inconsistentes en un archivo y continúa con el proceso. Al finalizar, informa de todas las incoherencias detectadas. Como contraejemplo: otro sistema que no tiene esas validaciones, tan solo se detiene al detectar fallas en la entrada de datos y con ello paraliza todo el proceso.

Para acercarse a los criterios mencionados, el mismo autor externa cinco reglas para tratar de cubrir las finalidades de la modularidad:

Correspondencia directa. La estructura modular obtenida en el proceso de construcción de un sistema de software debe hacer compatible con cualquier otra estructura modular obtenida en el proceso de modelado del dominio del problema.

En otras palabras, el modelo reflejado en la construcción del software debiera corresponder lo más posible al modelo que se manejó durante la especificación de requerimientos. Por ejemplo, imagine que se plantea el problema de un organizador: directorio, bloc de notas y agenda. No sería lógico crear un software que, aunque pudiera hacer todas las tareas, respondiera a un modelo conceptual ajeno por completo al ya señalado.

Pocas interfases. Cada módulo debe comunicarse con el menor número de módulos posibles.

Esta regla obedece a un criterio por demás lógico: si hubiera n número de módulos, cada módulo se podría comunicar con n-1 módulos, lo cual daría como resultado n (n-1) / 2 comunicaciones posibles. Si se emplearan casi todas las posibilidades, el sistema sería inmanejable. Por ejemplo, para 50 módulos habría 1 225 interacciones directas entre módulos. ¿Alguien podría entender el funcionamiento del sistema en una situación así?

Interfases pequeñas. Si dos módulos se comunican deben intercambiar la menor información posible.

En la medida que se intercambie menos información es más fácil realizar cualquier cambio y un error tiende a ser menos peligroso, con independencia de que el sistema gane en eficiencia.

Interfases explícitas. Siempre que dos módulos A y B se comuniquen, esto debe ser obvio a partir del texto de A, de B o de ambos.

Las llamadas explícitas se refieren a que un módulo llame en forma directa a otro. Las llamadas implícitas son los datos compartidos, como los archivos de uso común y las variables locales. Con una llamada explícita se sabe que A hace uso de B, con lo cual el seguimiento es más sencillo. Tener llamadas implícitas dificulta cualquier intento de comprensión, modificación o reutilización del código.

Ocultación de la información. El diseñador de cada módulo debe seleccionar un subconjunto de propiedades del módulo como información oficial sobre el módulo para ponerla a disposición de los autores de módulos clientes.

Para entender los beneficios de la ocultación de la información parta del hecho de que un módulo es conocido a través de una descripción oficial que señala sus propiedades públicas. Todo lo demás se desconoce. Cualquier uso de este módulo y cualquier prueba de su funcionamiento deben basarse solo en estas propiedades públicas. Por lo regular, las propiedades se refieren a la función del módulo y los datos de entrada y salida que requiere. El proceso interno de su tarea (el "cómo" lo hace) debe quedar como parte privada y nadie debe hacer uso de ella en forma directa.

Si se cumplen estas reglas se puede modificar el proceso interno (algoritmos, variables locales, etc.) sin que se afecte en modo alguno al sistema en su conjunto.

EJEMPLO

La rutina de raíz cuadrada en lenguaje C maneja la siguiente descripción oficial.[10]

```
#include "math.h"
double sqrt (double num);
La función sqrt() devuelve la raíz cuadrada de num. Si se llama a esta
función con un argumento negativo, se producirá un error de dominio.
```

Indica que se encuentra en la librería `math.h`, que requiere un número de tipo `double` como entrada y devolverá un número también de tipo `double`. El parámetro de entrada tiene que ser mayor o igual a cero. Los demás detalles quedan ocultos: si usa variables locales, el algoritmo que emplea, etc. Esta parte privada no se dice ni tiene porqué conocerla quien hace uso de este módulo.

[10] Schildt, Herbert,C. *Manual de Bolsillo*. McGraw-Hill, España, 1992, p. 180.

Tal vez estos criterios y reglas suenan en este momento abstractas e imprácticas. Sin embargo, suponemos que conforme se avance en el libro tomarán una forma más natural y se percibirán como una necesidad para un desarrollo de software más ágil y al que se le pueda dar mantenimiento.

EJERCICIOS SUGERIDOS

Ejercicios sugeridos con subrutinas

- Complete las cinco palabras faltantes, parta del hecho de que la siguiente subrutina devuelve el mayor valor de dos datos que llegan como parámetros. Cada inciso corresponde a una palabra. Después, pruebe la rutina y simplifique el código con la instrucción `?`.

```
_____(a)_____ mayor (float a, float b) {
   float ____(b)_________;
   if (a > b)
      aux = ____(c)_______;
   _____(d) _______
      aux = b;
   ____(e)_________ aux;
}
```

(a)	(b)	(c)	(d)	(e)

- Elabore una subrutina que reciba tres números flotantes y devuelva el mayor de estos números.
- Elabore la rutina sumatoria, que debe devolver la sumatoria de un número (1+2+3+...+n). Si se recibe un número negativo deberá devolver un -1. La rutina debe combinar en forma correcta con el siguiente programa principal.

```
int sumatoria (int a);
main() {
   printf("La sumatoria de 3 es: %d \n", sumatoria(3));
}
```

- Elabore la rutina factorial, que debe devolver la sumatoria de un número (1*2*3*...*n). Si se recibe un número negativo deberá devolver un -1. La rutina debe combinar en forma correcta con el siguiente programa principal y utilizar recursividad.

```
longint factorial (int a);
main() {
   printf("El factorial de 5 es: %d \n", factorial(5));
}
```

- ¿Qué desplegará el siguiente programa?

```
#include <conio.h>
#include <stdio.h>
void incremento();
   int main() {
   int p = 3;
   incremento();
   printf("El valor final de p es: %d \n", p);
   getch();
}
void incremento () {
   int p = 5;
}
```

- Elabore un programa que tenga dos parámetros (x y n) y devuelva el valor de xn, donde x es de tipo flotante y n es de tipo entero.
- La rutina siguiente intenta intercambiar dos números, pero no funciona bien. Corríjala con el uso de los parámetros por referencia en forma correcta:

```
#include <conio.h>
#include <stdio.h>
void intercambio (int a, int b);
int main() {
int cal1, cal2, cal3;
printf("Teclea 2 números:\n");
scanf("%d %d", &cal1, &cal2);
intercambio (cal1, cal2);
printf("Los números intercambiados son: %d %d", cal1, cal2);
printf("\n\nOprima cualquier tecla para continuar...");
getch();
}
void intercambio (int a, int b) {
int aux;
aux = a;
a = b;
 b = aux;
}
```

- Complete el siguiente código:

```
#include <conio.h>
#include <stdio.h>
float promedio (int cal1, int cal2, int cal3);
int main()
{
   int c1, c2, c3; _____indicar la palabra que falta_____ x;
   printf("Introduce los numeros a promediar:\n");
   scanf(«%d» «%d» «%d», &c1, &c2, &c3);
/* completar código, parte 1 */
printf("El promedio es: %6.2f\n", x);
}
float promedio (int cal1, int cal2, int cal3) {
   /* completar código, parte 2 */
}
```

- Elabore un programa que reciba los coeficientes de una ecuación cuadrática y devuelva las raíces de esa ecuación. Las raíces de la ecuación deberán ser manejadas a través de parámetros por referencia.
- Localice en Internet el problema y la solución recursiva de Torres de Hanoi y verifique que funciona de manera correcta. Realice la prueba de escritorio para el caso de tres discos.
- ¿Qué desplegará el siguiente programa?

```
#include <conio.h>
#include <stdio.h>
void cuadrado (int p);
void decremento (int *w);
int main() {
   int p = 3
```

```
   cuadrado(p);
   decremento(&p);
   printf("El valor final de p es: %d \n", p);
   getch();
}
void cuadrado (int p) {
   p = p * p;
}
void decremento (int *w) {
   *w = *w - 1;
}
```

- Elabore la subrutina que se combine de manera adecuada con el programa principal.

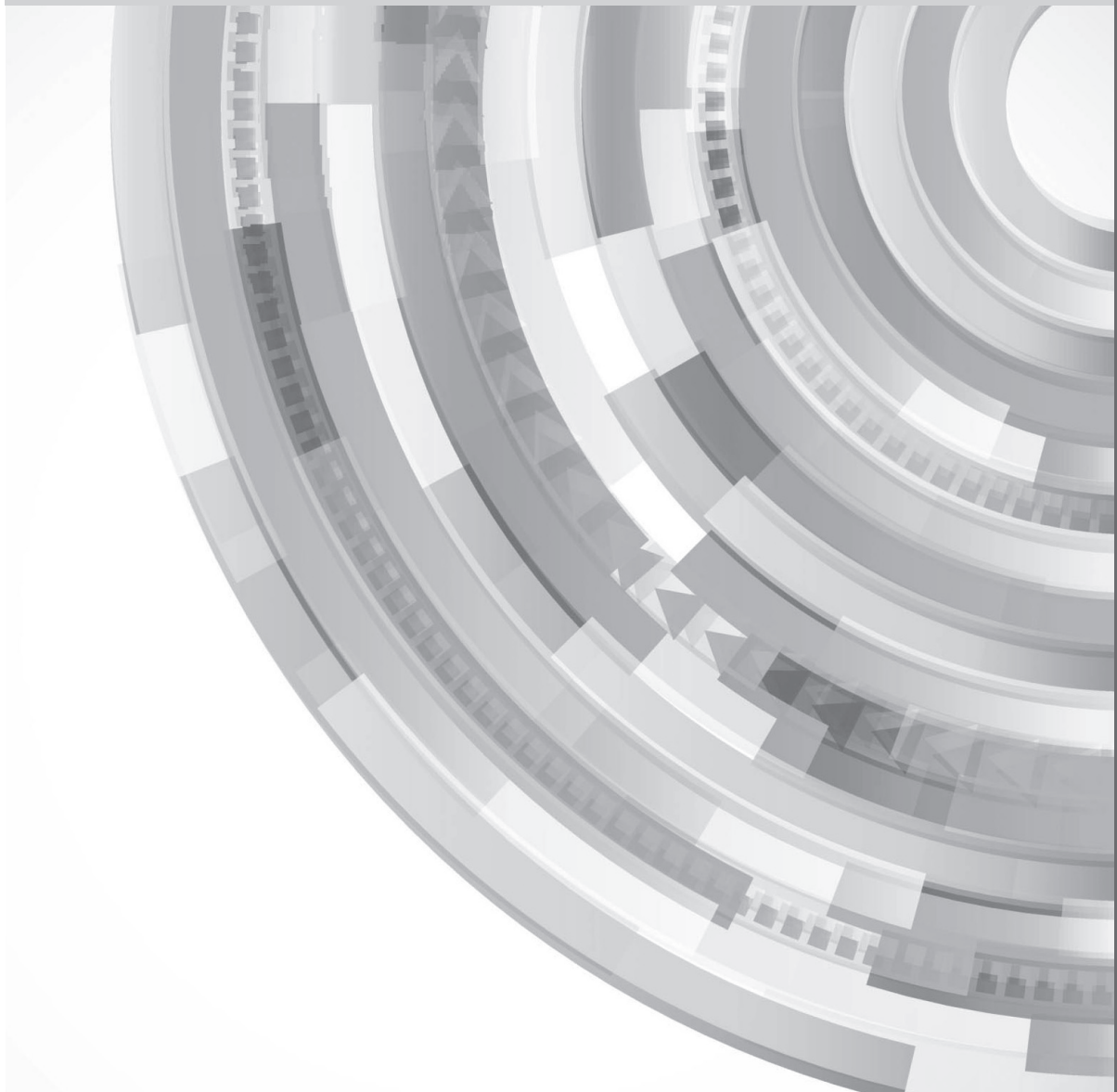
```
#include <conio.h>
#include <stdio.h>
main() {
   int x=3, y=4, z;
   suma (x, y, &z);
   printf("La suma de 3 + 4 es: %d \n", z);
   getch();
}
```

- Encuentre un error grave en el siguiente programa recursivo realizado en el producto Paradox (asuma que no hay errores de sintaxis).

```
method factorial (n: Smallint): number
Var
   i Smallint
begin
   x = n
   factorial = x * factorial (x-1) /* llamada recursiva */
end
```

Capítulo 4

El paradigma de la programación orientada a objetos



4.1 Un acercamiento intuitivo al paradigma de la POO

El desarrollo tecnológico pretende lograr que los sistemas estén integrados por componentes relativamente independientes, diseñados y probados antes de ser incorporados a un sistema global, de manejo sencillo y fácil construcción. Todo ello con la idea de que si algún componente presenta problemas solo hay que reemplazarlo por otro para solucionar la situación. A esta estrategia se le conoce como **desarrollo modular**. En general, se busca la reutilización de componentes o módulos de programas como lo hacen las bibliotecas de los propios lenguajes de programación.

En la programación estructurada, esta idea se cristalizó a través de subrutinas agrupadas en bibliotecas. Sin embargo, al paso del tiempo este avance fue insuficiente. Por ello, surgió la programación orientada a objetos (POO) como un intento por dominar la complejidad del desarrollo de software.

Para llegar a una buena práctica de la programación orientada a objetos hay que proseguir con el enfoque modular, que busca aplicar principios de **abstracción** al dividir un problema complejo en distintos módulos o partes. En otras palabras, se trata de agregar un nuevo tipo de módulo—en nuestro caso, clases y objetos— que permita manejar las variables que siempre es necesario manipular en los programas.

Al inicio es más fácil acercarse a este nuevo paradigma a través de ejemplos de la vida cotidiana. Si miramos a nuestro alrededor podemos percibir que estamos rodeados de objetos: con toda probabilidad en casa tenemos una televisión, un teléfono, una computadora, un automóvil, una bicicleta y muchas otras cosas más. Cada uno de estos objetos está compuesto por distintas partes o componentes que interactúan con otras. Para el caso de una bicicleta sabemos que tiene cadena, ruedas y varios objetos más, de manera que si los armamos o integramos en forma adecuada tendremos el objeto bicicleta.

En todos los objetos existen componentes que el usuario final puede manipular para hacerlos funcionar. No obstante, la forma en que trabajan está a nivel interno y debe ser desconocida e inaccesible para el propio usuario, lo cual hace más fácil su manejo y brinda un aspecto de seguridad (¿se sentiría bien si supiera que cualquier persona puede alterar la lógica interna del tablero del elevador?). En teoría, un componente solo debería comunicarse con otros componentes a través de los controles establecidos para ello. El foco de un cuarto únicamente debiera prenderse y apagarse a través de los contactos y no permitir que existiera un "cable oculto" en la sala que también pudiera cubrir dicha función, pues aunque da ciertas ventajas también podría ocasionar desde molestias hasta accidentes graves. ¿Suena exagerado? En muchos países miles de medidores de luz tienen "diablitos" que permiten que el hogar o el negocio reciban energía eléctrica sin que este consumo quede registrado para el cobro correspondiente (y muchos de estos "diablitos" improvisados han causado incendios).

Si los componentes funcionan en forma correcta es mucho más fácil su instalación y reemplazo. Un ratón de computadora que ya no sirva puede ser sustituido con suma facilidad siempre y cuando el nuevo ratón cuente con la misma interfase para interactuar con el sistema (en la actualidad, la entrada USB se ha convertido prácticamente en un estándar de entrada para este tipo de dispositivos).[1]

Esta forma de visualizar los objetos permite el diseño de aplicaciones orientadas a objetos y su posterior programación con lenguajes orientados a objetos, independientemente de cuál se utilice. Cuando un programa está elaborado en función de sus componentes, cada uno de ellos tiene una función predeterminada y puede interactuar con otros de diferentes formas predefinidas.

La programación orientada a objetos es una manera diferente de ver o atacar los problemas. De ahí la posible dificultad de aprender un lenguaje orientado a objetos sin entender las bases de la programación orientada a objetos.

Si se observa con cuidado, muchos de estos conceptos ya se aplican a subrutinas, pero la programación orientada a objetos crea una mayor independencia de los componentes con el programa principal.

[1] Un tema polémico acerca de la sustitución de componentes en el área electrónica y de cómputo es el impacto ecológico que trae consigo, sobre todo por la llamada obsolescencia programada. ¿Hasta dónde debe tirarse un componente y hasta qué punto debe ser reparado? El tema va más allá de lo teórico, pues también aplica a los componentes de software. Por desgracia, la discusión queda fuera de los alcances de este libro.

POO y objetos

Un primer acercamiento intuitivo a la programación orientada a objetos pasa justamente por los objetos, llamados también instancias de clase. Por ejemplo, mi computadora es una de las tantas que existen; representa una instancia de la clase de objetos conocida como computadoras. Como lo mencionamos antes, los objetos tienen estados o atributos: marca, procesador… y algunos métodos: lectura de archivos, escritura de archivos, etcétera.

El factor determinante es el estado muy particular de cada computadora, que es independiente del estado de las demás computadoras. La particularidad de cada atributo puede ser diferente en cada caso. Una computadora tiene un procesador, que es de alguna de las distintas velocidades de procesamiento que existen; también puede ser de una marca entre varias. De manera que podemos definir un grupo de variables y métodos para todas las computadoras.

En la programación orientada a objetos se abordan los problemas a través de objetos, situándolos en un escenario del "mundo real" del problema para implementarlo en nuestro programa. Por ejemplo, si hacemos referencia al objeto automóvil, podemos deducir que tiene características y comportamiento diferentes.

Características o atributos:	Marca, modelo, color, HP…
Comportamiento o métodos:	Frenar, acelerar, retroceder…

La realidad (perros, edificios, servicios, etc.) será representada a través de **objetos** dentro de un programa. Estos objetos poseen un conjunto de propiedades o atributos y un conjunto de métodos mediante los cuales muestran su comportamiento. Las posibles características (**atributos**) de un objeto y su comportamiento (**métodos**) quedan definidos por la **clase** a la cual pertenece.

Un objeto representa un concepto dentro de un programa y tiene la información necesaria para abstraerlo: los datos que describen su estado y las operaciones que pueden modificar este y determinan las capacidades del objeto. Los objetos tienen un tiempo de vida (existen mientras está en ejecución el programa) y un comportamiento, en el que los programas interactúan con los objetos mediante interfases de otros objetos.

Un objeto es un conjunto de atributos y métodos que se relacionan entre sí y que modelan objetos o entidades del mundo real. En otras palabras, un objeto es la representación en un programa de un concepto y contiene toda la información necesaria para su abstracción: datos o variables que describen sus atributos y operaciones que pueden realizarse sobre estos.

Suponga que existen dos objetos de la clase foco, llamados foco1 y foco2. Ambos tendrán los mismos atributos y métodos, aunque cada uno tendrá sus propios valores para los distintos atributos, como se muestra en los diagramas de clase de la figura 4.1. En resumen, un objeto es una unidad de código compuesto de atributos y métodos relacionados.

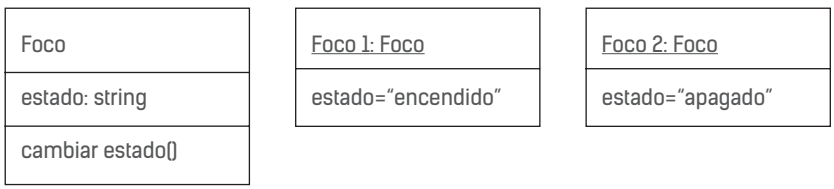


Figura 4.1 Una clase foco con un atributo y una rutina de comportamiento. A partir de ella se definieron los objetos foco1 y foco2, que tienen diferente estado.

Si retomamos el ejemplo de un automóvil, podríamos tener un Fiat 500, año 2014, color rojo y que alcanza una velocidad de 240 km/h. En otras palabras, tendremos al objeto automóvil con sus características definidas. Cuando a las características del objeto se les ponen valores entonces se dice que el objeto

tiene estados. Los atributos almacenan los estados de un objeto. Los datos anteriores los podríamos expresar de la siguiente manera:

Atributo	Valor
Marca	Fiat
Modelo	500
Año	2014
Color	Rojo
Velocidad	240 km/h

Los lenguajes de programación estructurados basan su funcionamiento en el concepto de subrutina (procedimiento o función). En el caso de los lenguajes orientados a objetos, como C++ y Java, el elemento básico no es la función, sino el objeto. Un objeto es la representación en un programa de un concepto que contiene la información necesaria para abstraerlo: elementos que describen sus atributos y operaciones que pueden realizarse sobre ellos. Un objeto de software mantiene sus características en uno o más atributos e implementa su comportamiento con métodos (subrutinas que se asocian a un objeto).

POO y clases

Para crear objetos es necesario primero crear las clases, que son tipos de datos que definen los atributos y los métodos que serán comunes para todos los objetos de esta clase. Por ejemplo, ya que se ha creado la clase computadora, podemos entonces crear una gran cantidad de objetos computadora a partir de la clase. Cuando creamos un objeto, este tendrá su propia copia de las variables miembro (atributos) definidas en la clase.

"Todo objeto es de algún tipo. Cada objeto es *un elemento de una clase*, entendiendo por *clase* un sinónimo de *tipo [de dato]*. La característica más relevante de una clase la constituyen 'el conjunto de mensajes que se le pueden enviar'."[2]

"Un programa es un cúmulo de objetos que se dicen entre sí lo que tienen que hacer mediante el envío de mensajes". Cada objeto puede recibir peticiones y, en razón de ellas, realizar una operación determinada. Esas peticiones solo pueden hacerse por la interfase que el mismo objeto ha diseñado.

Java es un lenguaje de objetos que incorpora el uso de la orientación a objetos como base fundamental del lenguaje, lo que representa una diferencia importante respecto a C++. Al contrario que en C++, en Java no se puede hacer mucho sin utilizar al menos un objeto.

Las clases son abstracciones que representan objetos con un comportamiento e interfaz común, que presentan el estado de los objetos a los que pertenecen mediante variables denominadas atributos. A las subrutinas que definen el comportamiento de una clase se les denomina métodos y a través de ellas se modifican los valores de los atributos y capacidades del objeto.

Algunos puntos específicos para trabajar con clases:

1. Existe un método especial en las clases que permite crear al objeto y asignar valores iniciales a sus datos. A este método se le conoce como **constructor**. El constructor tiene el mismo nombre que la propia clase.
2. Puede haber varios métodos con el mismo nombre en la misma clase. Se distinguirá entre ellos porque reciben diferentes parámetros, ya sea en número o tipo. A esto se le conoce como **sobrecarga**.

[2] Tomado de Eckel, Bruce. *Piensa en Java*. Pearson-Prentice Hall, España, 2002, p. 3.

3. Al método contrario, que realiza algunas tareas a la hora de eliminar ese objeto de la memoria, se le conoce como destructor. El destructor no existe de manera explícita en todos los lenguajes orientados a objetos (en Java el recolector de basura —Garbage Collection—se encarga de liberar la memoria de los objetos que ya no se usan).

En las clases existen elementos públicos y privados —y un nivel intermedio: el ámbito protegido. Por lo regular, los datos son de naturaleza privada y los métodos son públicos. Solo puede accederse a las partes privadas a través de los propios métodos de las clases. Con ello se impide que se modifique esa información desde fuera del objeto.

Encapsulamiento es la capacidad de ocultar algunos miembros de la clase, pero proporcionando una interfase pública para acceder a esos elementos. Esa interfaz fija una serie de reglas que no se pueden transgredir (véase figura 4.2).

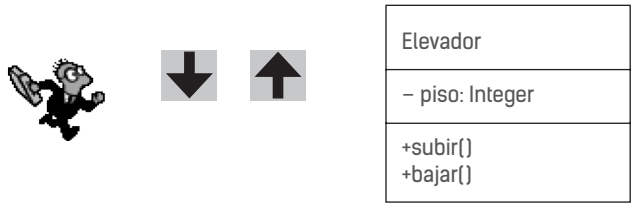


Figura 4.2 Un objeto elevador y su interfase externa para el usuario. A través de esta interfase solo se puede indicar: a) que se desea subir o b) que se desea bajar. Un atributo interno es el piso en que se halla el elevador.

Composición, herencia y polimorfismo

"Cada objeto tiene su propia memoria, constituida por otros objetos." Los objetos pueden contener atributos (datos), métodos (rutinas), o bien, otros objetos.

Se pueden crear objetos a partir de otros; es decir, objetos que incluyan a otros (**composición**). También es posible que un objeto herede las propiedades de otro, en cuyo caso el nuevo objeto tendrá los datos y mensajes de la clase padre, más los datos y mensajes que desee agregar (este mecanismo se conoce como herencia; véase figura 4.3). También existe la posibilidad de redefinir un método (**sobreescritura**).

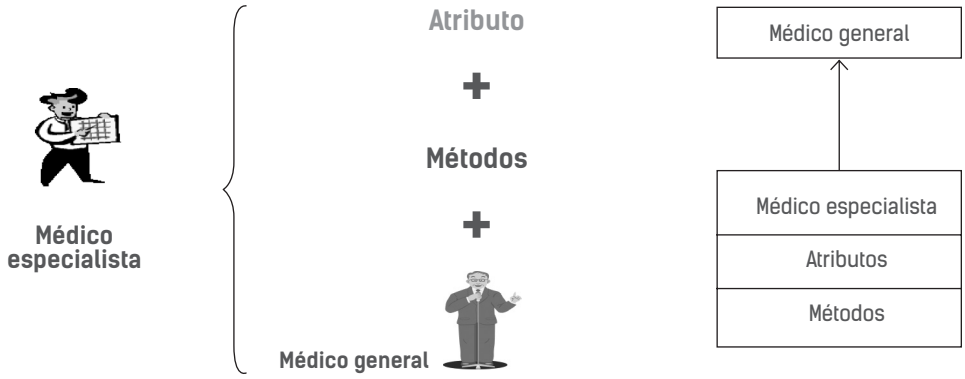


Figura 4.3 Una clase puede heredar atributos y métodos de otra clase. Sobre esa base puede añadir sus propios atributos y métodos. También puede sobreescribir los métodos recibidos.

La programación orientada a objetos nos permite definir clases a partir de otras clases ya construidas. Por ejemplo, las computadoras pueden ser de escritorio o portátiles; sin embargo, en términos generales,

al fin y al cabo son computadoras. Esto, en función de la programación orientada a objetos, representa subclases derivadas de la clase computadora. Las subclases heredan los estados de la superclase de la cual se derivan. En este caso, las computadoras de escritorio y las portátiles comparten algunos estados como, por ejemplo, la capacidad de almacenamiento, la velocidad de su procesador, de tal forma que cada subclase hereda los métodos de su superclase.

La herencia permite crear clases derivadas a partir de una clase base. Nos permite compartir en forma automática métodos y datos entre clases, subclases y objetos. Las clases que se derivan de otras heredan su comportamiento y además pueden introducir características particulares propias que las diferencian; de esta manera define una relación jerárquica entre todos los conceptos que se están manejando. Esta técnica, muy útil en el desarrollo de sistemas, se emplea para descomponer un problema en un conjunto de problemas subordinados a él, cuyo comportamiento es identificable con facilidad. El árbol de herencias o jerarquía de clases puede ser tan extenso como se necesite. Los métodos y las variables miembro se heredarán hacia abajo a través de todos los niveles de la jerarquía. Cuanto más abajo está una clase en la jerarquía, más especializado es su comportamiento.

En resumen, la herencia es la propiedad por la que una clase obtiene aquellos métodos y atributos de una segunda clase que se han especificado como heredables. La generalización es una perspectiva ascendente, mientras que la especialización es una perspectiva descendente.

"Todos los objetos de determinado tipo [de dato] pueden recibir los mismos mensajes". Esta es una afirmación de enorme trascendencia, como se verá más adelante. Dado que un objeto de tipo *triángulo* es también un objeto de tipo *polígono*, se garantiza que todos los objetos *triángulo* acepten mensajes propios de *polígono*. Esto permite la escritura de código que haga referencia a *polígonos*, y que de manera automática puede manejar cualquier elemento que encaje con la descripción de *triángulo*. Esta capacidad de suplantación es uno de los conceptos más potentes de la POO, el polimorfismo: la capacidad de que una referencia pueda apuntar a objetos de diferente tipo.

Debe recordarse que una clase tiene los datos y mensajes que heredó de la clase padre, más los datos y métodos que desee agregar. Si hay dos clases que heredaron de la misma clase padre y no se hizo ninguna redefinición, al menos tendrán en común los datos que heredaron.

Los objetos pueden tener ciertas relaciones entre ellos; tal es caso de la **asociación**, en donde un objeto realiza llamadas a los métodos de otro, interactuando de esta forma con él.

Otras formas de relaciones son la **agregación** y la **composición**. En ambas un objeto pertenece a otro objeto. En la agregación los dos objetos existen de manera independiente y, al mismo tiempo, uno forma parte de otro (por ejemplo, la relación de empresas con clientes). En la composición, un objeto solo tiene sentido si existe el otro objeto (por ejemplo, el departamento de una empresa). Al desaparecer el objeto "principal" (la empresa) de manera automática desaparecerá el otro objeto (el departamento de la empresa). La composición es duradera; es decir, permanece durante toda la vida del objeto que la compone; en la agregación no necesariamente es así: se puede implementar como un objeto compuesto, que cuenta entre sus atributos con otro objeto distinto.

EJERCICIOS SUGERIDOS

Ya se mencionó el ejemplo del elevador, cuyo atributo privado es el número de piso. En otras palabras, este dato que aparece en un recuadro del propio elevador no puede ser variado por el usuario; solo se modifica conforme se suben y bajan pisos. Por otra parte, existen funciones que el usuario puede manipular: ir a un determinado número de piso, abrir puertas, cerrar puertas y, en su caso, llamadas de auxilio.

El ejercicio consiste en localizar cinco elementos de la vida cotidiana de los estudiantes e identificar los atributos privados y las funciones que están disponibles para el usuario.

4.2 Un acercamiento al concepto de clases vía programación

Antecedentes de la POO

De manera paralela al desarrollo del hardware en los equipos de cómputo, se ha dado un desarrollo constante de los lenguajes de programación, que abarcan desde los lenguajes de bajo nivel como ensamblador hasta lenguajes de alto nivel, tanto para aplicaciones científicas, matemáticas, las orientadas a negocios y de aplicación general. Todos estos lenguajes desde su aparición marcaron el rumbo de la programación, muchos de gran trascendencia y otros que no fueron muy utilizados: Fortran, COBOL, Pascal, BASIC, C, Java, C#, Python…

No es tema de este libro hacer una descripción de la gran cantidad de lenguajes de programación que han aparecido a la par de las computadoras, pero es necesaria una referencia sencilla de algunos de los que se han utilizado más y que se han ido perfeccionando con el paso del tiempo para así poder apreciar las potencialidades y el enfoque de la programación orientada a objetos.

Los primeros lenguajes, llamados **ensambladores**, aparecieron a principios de la década de 1940 y se caracterizaban por el uso de símbolos o mnemónicos que correspondían a instrucciones de código de máquina.

Uno de los primeros lenguajes de alto nivel, llamado Fortran (FORmula TRANslator), aparece a mediados de la década de1950 y su utilización fue de cobertura muy amplia, incluyendo aplicaciones científicas y militares. Bajo este lenguaje se desarrollaron una gran cantidad de librerías o bibliotecas que a la fecha se han mejorado y se siguen utilizando.

A principios de la década de 1960 se desarrolla el lenguaje LISP, cuyo principal objetivo era la creación de programas relacionados con la inteligencia artificial, ya que estaba provisto de conceptos de programación funcional y recursividad.

En esos mismos tiempos también es desarrollado el lenguaje COBOL (Common Oriented Business Language), principalmente orientado a aplicaciones de negocios y al procesamiento de grandes volúmenes de información. Su mayor utilización fue en el ámbito bancario de grandes oficinas de negocios y, aunque parezca raro, hoy día aún existen empresas que lo siguen utilizando.

Casi a mediados de la década de1960 aparece el lenguaje BASIC (Beginner's All-purpose Symbolic Instruction Code), cuyo propósito principal era enseñar a programar a estudiantes que querían incursionar en el campo de la programación.

En la década de1970 es desarrollado el lenguaje Pascal con el objetivo de enseñar la programación estructurada (sin los saltos incondicionales de BASIC) de una manera sencilla y fácil, a tal grado que su utilización perduró hasta la década de 1990 y dio pie a la creación de un nuevo lenguaje llamado Delphi.

Por esas mismas fechas, el auge de los lenguajes de programación fue en aumento y, en 1972, Dennis M. Ritchie desarrolló el lenguaje C en los laboratorios Bell. De hecho, uno de los sistemas operativos más poderosos, UNIX, fue desarrollado en este lenguaje. Esto dio pie al desarrollo de muchos otros lenguajes de programación de entre los que sobresalen algunos actuales como C++, Java y C#.

La década de 1980 dio pie a la aplicación de la programación estructurada en entornos gráficos, sobre todo a raíz de la masificación del uso de ventanas en computadoras personales, lo cual se conoció como programación basada en objetos. De esta época data Visual Basic de Microsoft, que se combinaba por lo común con el sistema manejador de bases de datos (SMBD) Access y ObjectPal para explotar el SMBD Paradox, de Borland. Cabe aclarar que Access y Paradox no reunían todas las características de un SMBD, pero eran un acercamiento de bajo costo para aplicaciones en área local y muy pocos usuarios concurrentes.

La POO tiene sus orígenes en el lenguaje Simula utilizado a finales de la década de 1970, aunque el paradigma POO como tal se presenta a finales de la década de1980; ofrece una nueva forma de programar que da mayor facilidad para escribir y dar mantenimiento al código. Con el paso del tiempo estos lenguajes se fueron sofisticando más y más, y así apareció el C++, cuyas bases fueron tomadas del C, aprovechando potencialidades de Simula y Smalltalk, adicionando propiedades y características para el

manejo y trabajo con objetos. Al paso del tiempo, C++ se volvió uno de los lenguajes más representativos en lo que se refiere a la POO. Como un digno sucesor del C, C++ ha ido evolucionando a tal grado que tiene prácticamente todas las construcciones sintácticas para representar los tipos de reutilización necesarios, aunque esto lo ha vuelto un poco difícil de entender y de manejar para muchos principiantes de la programación orientada a objetos. La programación es un proceso en el que el concepto de abstracción tiene un valor muy significativo. C++ es un lenguaje que permite desarrollar programas eficientes, debido a que incorpora todas las construcciones sintácticas que se asocian con la programación orientada a los objetos.

A inicios de la década de 1990 aparece el lenguaje de programación orientada a objetos Java, creado por Mike Sheridan, Patrick Naughton y James Gosling, de la empresa Sun Microsystems. Por cierto, se dice que el origen del nombre del lenguaje Java se debe a que sus creadores discutían sobre cómo llamar al lenguaje mientras tomaban café, y pusieron el nombre de la marca del café que tomaban, el café Java. Java fue diseñado bajo un esquema en el que un programa escrito en este lenguaje pudiera ejecutarse en cualquier máquina, no solo en computadoras, sino en gran variedad de dispositivos electrónicos. Aunado a esto, la potencialidad de Java se ha incrementado debido a su convergencia con internet.

En la actualidad, muchos lenguajes de programación conservan un lugar importante debido al gran número de aplicaciones que tienen y al mercado que abarcan. Tal es el caso de lenguajes como Cobol, C y C++.

En lo que respecta a lenguajes de consulta a bases de datos relacionales, se tiene sin lugar a dudas el SQL. Para el desarrollo de aplicaciones orientadas a internet, hoy día uno de los lenguajes más utilizados es Java, aunque también se utilizan PHP y los que soportan en forma amplia la plataforma .Net, como son Visual.NET y C#.

Otro grupo de lenguajes que tienen mayor interés son los lenguajes como Perl, Phyton y Ruby, a los que no está por demás ponerles atención y hacer un análisis comparativo para determinar sus tendencias y potencialidades. JavaScript —técnicamente un lenguaje C para validaciones del lado del cliente en aplicaciones para internet— es otro lenguaje que está jugando un papel importante en el desarrollo de aplicaciones. También debemos mencionar tecnologías emergentes como Ajax, que facilitan la creación de ambientes RIA (*Rich Application Interface*).

El acceso a la información que ha permitido internet ha cambiado la forma en que nos comunicamos o interactuamos con nuestro entorno. Los lenguajes de programación para computadoras jugarán un papel importante y tendrán una relación estrecha con las aplicaciones y la evolución de la misma web en el mercado de la tecnología de la información. Tecnologías emergentes como el cómputo en la nube (*cloud computing*), los entornos virtuales, lenguajes para realidad aumentada, los lenguajes visuales o las tendencias en la mejora de la experiencia del usuario en aplicaciones web, son tecnologías que cambiarán la forma en la que interactuamos con las computadoras.

En resumen, al pasar de la programación estructurada a la programación orientada a objetos se desarrollaron los lenguajes como Simula y Smalltalk. Poco después aparece el lenguaje C++, cuyo compilador es capaz de reunir programas de C; es decir, un lenguaje capaz de interpretar dos estilos diferentes de programación. Hay que cuidar que esta ventaja no se vuelva contraproducente al paso del tiempo, ya que si no se necesita de un diseño orientado a objetos, quizá muchas de las aplicaciones hechas en C++ no tengan una orientación meramente orientada a objetos.

Las variables globales: un riesgo latente en la programación estructurada

Para analizar una de las situaciones que da origen a la programación orientada a objetos analizaremos el ejemplo concreto de una pila instrumentada a través de arreglos. Las pilas son estructuras de datos que se aplican a cálculos numéricos o subrutinas en computación. Por ello, su política inviolable de trabajo es últimas entradas primeras salidas (UEPS): en una expresión aritmética primero debe resolverse lo que está entre paréntesis; en la ejecución de un programa el módulo principal cede el control a una subrutina y

solo cuando termine la subrutina devuelve dicho control al programa principal. La política UEPS en una pila es innegociable.

La forma más sencilla de instrumentar una pila con base en arreglos es definir un arreglo y una variable entera que fungirá como índice. Este índice siempre señalará al último elemento que entró en la pila y que a su vez es el primero que tendrá que salir. Cuando se introduce un elemento se incrementa el índice; cuando se saca un elemento disminuye el índice (véase cuadro 4.1). Una pila estará llena si el índice señala al último elemento del arreglo. Para conservar la lógica del algoritmo, el índice valdrá –1 si la pila está vacía.

Cuadro 4.1 Instrumentación de una pila con arreglos

'h'	'o'	'l'	'a'						
0	1	2	3	4	5	6	7	8	9
			/ \						
			|						
			índice						

A nivel código, se requerirán al menos dos subrutinas: una que incorpore un elemento a la pila (llamada por lo común meter o *push*) y otra que saque un elemento de la pila (conocida como sacar o *pop*). Además, pueden existir otras subrutinas que permitan conocer si la pila está llena o vacía. Al haber más de una subrutina que hace uso del arreglo y del índice, es indispensable que estas variables sean de carácter global. Por último, se incluye un pequeño programa de prueba para mostrar que la pila trabaja en forma adecuada (en realidad, este tipo de programas de prueba deberían ir en un archivo aparte, pero por cuestiones pedagógicas se dejan en el mismo programa en este primer acercamiento). Bajo estas consideraciones, resultará un código muy similar al mostrado en el programa 4.1.

```
/* Pila en C con programación estructurada. */
Programa 4.1
#include <conio.h>
#include <stdio.h>
#define MAXPILA 50 /* tamaño máximo de la pila */
struct pila {
   int ultimo;
   char datos[MAXPILA];
};
/* inicializa el índice de la pila */
int inicializar (struct pila *pilaux)
{
 pilaux -> ultimo = -1;
}
/* retorna 1 si la pila está vacía. En caso contrario, devuelve 0.*/
int vacia (struct pila *pilaux)
{
   return(pilaux -> ultimo == -1);
}
/* retorna 1 si la pila está llena. En caso contrario, devuelve 0.*/
int llena (struct pila *pilaux)
{
   return(pilaux -> ultimo == MAXPILA-1);
}
```

```
void push (struct pila *pilaux, char dato) {
   (pilaux -> ultimo)++;
   pilaux -> datos [pilaux -> ultimo] = dato;
}
char pop (struct pila *pilaux) {
   char auxiliar;
   auxiliar = pilaux -> datos [pilaux -> ultimo];
   (pilaux -> ultimo)--;
return(auxiliar);
}
int main(void) {
   char dato;
   struct pila pila1;
   printf("Programa que despliega un mensaja al revés\n");
   printf("Ingresa la cadena para la pila. Termina con ENTER.\n");
   inicializar(&pila1);
   dato = getche();
   while (dato!=13) { /* el Enter es el carácter 13 en ASCII */
   push(&pila1,dato);
   dato = getche();
   }
   printf("\n\nLa cadena invertida es:");
   while (!vacia(&pila1)) {
   printf("%c",pop(&pila1));
   }
   printf("\n\nOprima cualquier tecla para continuar...");
   getch();
}
```

```
Programa que despliega un mensaje al revÚs
Ingresa la cadena para la pila. Termina con ENTER.
hola

La cadena invertida es:aloh

Oprima cualquier tecla para continuar...
```

Figura 4.4 Una pila en lenguaje C instrumentada con arreglos.

Todo parece normal y deseable. Sin embargo, queda un problema de mantenimiento y seguridad que por lo normal pasa desapercibido: los datos de la pila son variables globales separadas del algoritmo, con lo cual se abre la posibilidad de que cualquier otra rutina pueda hacer uso de ellos al infringir sus políticas de acceso en forma maliciosa o por error. En principio, pareciera que esto es muy difícil. Sin embargo, imagine que el sistema utiliza muchas estructuras (pilas, colas, árboles, etc.): se llenaría de variables globales que complicarían tanto la programación como el mantenimiento. Por ello, la programación orientada a objetos crea clases bajo las siguientes consideraciones:

1. Las clases reúnen en un mismo componente la lógica y los datos.
2. Las clases protegen a los datos declarados como privados de accesos no autorizados (los "encapsulan"). En otras palabras, solo las rutinas de la propia clase pueden tener acceso a estos.

3. Las propias clases definen aquellos atributos o métodos **públicos**; es decir, que pueden ser llamadas desde otros programas. Estas subrutinas serán la forma de comunicación con los otros componentes.

Las clases se especifican en forma gráfica a través de los diagramas de clase del Lenguaje Unificado de Modelado (UML, por sus siglas en inglés). Las partes privadas se señalan con un guión (–) y las partes públicas con un signo de más (+). La sintaxis que describe los métodos no pertenece a ningún lenguaje de programación en particular, por lo cual debe ser codificada en el lenguaje correspondiente. Bajo las consideraciones anteriores, la clase Pila se muestra en la figura 4.5.

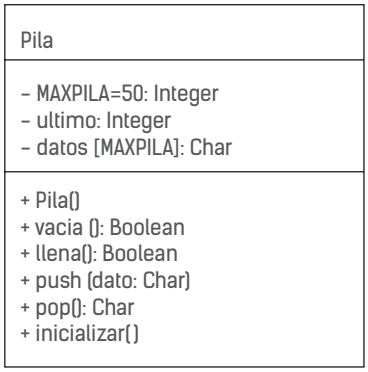


Figura 4.5 Diagrama UML de la clase Pila.

La clase Pila tiene tres variables privadas: una constante MAXPILA, último y el arreglo datos. Puesto que son variables a nivel de toda la clase, se conocen como atributos. Tienen una rutina que se llama igual a la clase que permite crear todas las pilas que se requieran (a esta clase se le conoce como constructor y las variables de la clase se conocen como objetos). Por último, hay cinco subrutinas que pueden ser llamadas por otros componentes (a estas subrutinas se les conoce como métodos).

Como se verá, existe un cambio conceptual entre la programación estructurada y la programación orientada a objetos. ¡Y apenas comenzamos!

La reacción natural a esta altura es querer ver el código construido en el nuevo paradigma. Sin embargo, al igual que en su momento sucedió con la programación estructurada, conviene detenerse un momento para tratar de asimilar los nuevos conceptos en forma abstracta.

Como puede observarse, para llegar a una buena práctica de la programación orientada a objetos primero hay que pasar por la programación modular. Ambos paradigmas buscan aplicar principios de abstracción al dividir un problema complejo en distintos módulos: en la programación estructurada, se trata de subrutinas; en la POO se agregan las clases. Al definir una clase se define un tipo de dato y los objetos son variables de ese tipo de datos. Un programa o una clase no pueden realizar trabajo alguno sin antes ser alimentados por datos, que llegarán a través de los métodos de la propia clase.

Con base en el principio de clase, la POO incorpora otros principios; entre ellos, la herencia, la composición y el polimorfismo. Por ello representa un gran avance con respecto a la programación estructurada. Por una parte, aprovecha la potencialidad de condicionales, ciclos y subrutinas; por otra, integra los datos y procedimientos, que antes estaban por separado, en componentes llamados clase. De hecho, el cambio principal tanto a nivel conceptual como metodológico es que ahora se piensa en componentes de código relativamente independientes con estructuras de comunicación claramente definidas a través de las cuales se integran al sistema completo.

La programación orientada a objetos facilita herramientas para poder tener buenos niveles de calidad, mejora el mantenimiento, proporciona bibliotecas de clases y fragmentos de código para su reutilización en otros programas. Los principios de: abstracción, encapsulamiento, modularidad y jerarquía son fundamentales en el modelo orientado a objetos. Se busca abatir la complejidad de la programación para que esta sea más fácil y sencilla. C++ y Java incorporan la orientación a objetos como uno de los pilares básicos de su lenguaje.

EJERCICIOS SUGERIDOS

- Instrumente el programa de pila e intente hacer el paso siguiente para demostrar su vulnerabilidad: intente cambiar un dato en forma directa en el arreglo sin llamar a las rutinas de meter y sacar.
- Agregue una clase cola al programa que utilice un arreglo de la misma dimensión. Ahora intente llamar la función meter de cola, pero indicando el arreglo de la pila. Podrá observar que errores de ese tipo pueden generar una gran cantidad de problemas.

REFLEXIÓN 4.1

¿Cómo iniciar un curso de programación orientada a objetos (POO)?

El inicio de los cursos de programación orientada a objetos no es una tarea sencilla. Se requiere que los estudiantes conozcan el paradigma de la programación estructurada y tengan práctica en el manejo de subrutinas y arreglos, así como claridad en el concepto de apuntadores. En este caso, recomendamos, si es necesario, hacer un repaso de lenguaje C, pero haciendo hincapié en que se sigue en programación estructurada. Con esa base, intentar que se capte la idea de un objeto como entidad abstracta; es decir, que al inicio solo se programarán componentes independientes con su respectivo programa de prueba ("focos" con su respectivo "probador de focos").

Estas ideas no se tienen del todo explícitas en cursos y materiales de referencia. Hemos observado libros de C++ bien escritos desde el punto de vista técnico en que la primera mitad se refiere a programación estructurada y en la mitad restante hablan de la orientación a objetos, pero nunca hacen explícitos los paradigmas ni el cambio en la forma de trabajo.

Hacer hincapié en lo que implica cada paradigma es una necesidad pedagógica muy importante.

4.3 Panorama general del lenguaje Java

Java fue creado por Mike Sheridan, Patrick Naughton y James Gosling a inicios de 1990, quienes trabajaban para Sun Microsystems. En sus inicios, el enfoque de aplicación de Java fue prácticamente para el control de aparatos electrodomésticos; a estos trabajos se les denominó "Proyecto Green". Otro de los campos iniciales en los que se aplicó Java fue llamado VOD (Video On Demand), que se utilizaba como interfaz para la televisión interactiva. Estas aplicaciones se desarrollaron por completo en un Java primitivo; sin embargo, no llegaron a ser sistemas que se comercializaran. Cabe mencionar que en sus orígenes el lenguaje se llamó Oak (que significa roble), pero por alguna razón cambió de nombre y hoy se conoce como Java.

El lenguaje al fin tomó cierta fortaleza al ser orientado a plataformas web y en 1995 Netscape incluyó en sus navegadores el soporte para Java, con lo que se reposiciona como lenguaje con muchas potencialidades.

Se dice que entre las principales razones por las que se llegó a la creación de Java estaba la importancia de tener interfases cómodas e intuitivas y la necesidad de contar con herramientas poderosas y fáciles de utilizar en el proceso de desarrollo de sistemas, situación que en C y C++ demandaban grandes esfuerzos por parte de los programadores. Además, se requería flexibilidad para correr en distintas plataformas e interactuar con diferentes dispositivos periféricos. Por ello desde un principio se planteó la situación de código abierto.

Bajo este contexto, el equipo liderado por Gosling hace la propuesta de crear un nuevo lenguaje de programación lo más sencillo posible con la finalidad de adaptarse a cualquier entorno de ejecución sin mayor problema. No está por demás mencionar que ya se tenía un panorama claro de la problemática que en un momento dado se presentaba con algunos otros lenguajes de programación como C y C++. Como ya se ha dicho, Java es un lenguaje orientado a objetos, sencillo pero a la vez potente, lo que le permite tener ciertas facilidades al abordar diferentes tipos de problemas, y con capacidad de ejecución en distintas plataformas, incluyendo su gran incursión en los ambientes de internet.

Respecto a C, Java puede catalogarse como un lenguaje lento. Esto se debe a que Java utiliza la representación llamada **código de byte**, que tendrá que ser traducido al código de máquina donde se corra la aplicación. Sin embargo, esto no ha sido un impedimento para seguir con su crecimiento.

Java es una plataforma que ha ido madurando con el paso del tiempo con un futuro prometedor; desde hace tiempo cuenta con un variado número de herramientas que fortalecieron su entorno de desarrollo. Entre ellas sus librerías, su facilidad para lograr una buena conectividad y su fácil interacción con ambientes web. Java proporciona elementos para la web con una interactividad casi total entre usuario y aplicación, basada en diseño orientado a objetos. Su sintaxis es entendible con facilidad, además que facilita un conjunto de clases potente y flexible.

Algunas de las características principales del lenguaje de programación Java son:[3]

Simple. A nivel general, Java conserva gran parte de las reglas sintácticas de C, aunque elimina algunas que creaban confusión y ya no emplea apuntadores explícitos. En su orientación a objetos, el concepto de objeto resulta sencillo y fácil de ampliar y entender. Se conservan tipos de datos simples como números, caracteres y otros tipos de datos simples. Gracias a su simplicidad semántica, cuando se definen tareas, estas se pueden realizar en un número reducido de formas, consiguiendo con esto un gran potencial de expresión e innovación por parte de los programadores.

Distribuido. "Java tiene una extensa biblioteca de rutinas para realizar copias con los protocolos de TCP/IP como HTTP y FTP. Las aplicaciones de Java pueden abrir y acceder a objetos a través de la red vía URL con la misma facilidad con la que se accede al sistema local de ficheros."

Robusto. La robustez de Java radica en su capacidad para verificar la buena escritura del código. Esto se logra en la etapa de compilación, al verificar la escritura correcta en cuanto a tipos y declaraciones. En lo que respecta a la administración de memoria, Java posee una gestión avanzada de memoria llamada gestión de basura y un manejo de excepciones orientado a objetos integrados, lo que libera al programador de tareas tediosas con relación a la administración de la memoria.

Seguro. En relación con la seguridad, una amenaza palpable está representada por los malware, caballos de Troya y una gran variedad de virus que viven en la red. Al respecto, Java ha establecido niveles de seguridad desde su diseño: restricción de acceso a memoria al no utilizar punteros explícitos; la inclusión de funciones de verificación a nivel de códigos de byte; el aseguramiento de que son consistentes los nombres de clase y sus restricciones de acceso al momento de la carga del programa (se sabe si el acceso a los campos de un objeto es legal mediante las palabras reservadas public, private y protected).De manera paralela se pueden utilizar mecanismos alternos para la encriptación de código, datos e información y así tener niveles de seguridad que cumplan con los estándares establecidos.

Portable. Java está diseñado para que un programa escrito en este lenguaje sea capaz de ejecutarse en diferentes plataformas: Macintosh, Windows o UNIX. Esto se consigue mediante la utilización de una representación intermedia denominada código de byte (*bytecode*), capaz de interpretarse en diferentes sistemas operativos. A diferencia de C y C++ no hay aspectos de la especificación "dependientes de la implementación"; el intérprete Java puede ejecutar bytecode de Java en forma directa en cualquier máquina a la que se haya trasladado el intérprete. Están especificados los tamaños de los datos básicos, así como su comportamiento aritmético.

Orientado a red. Java tiene la posibilidad de crear programas interactivos bajo una plataforma en red, ya que es capaz de ejecutar varios procesos a la vez sin perder de vista lo que sucede con otras aplicaciones. Esto se logra mediante la utilización de múltiples hilos de programación (*multithread*).

Java es capaz de generar aplicaciones para animación de páginas web, teniendo además la posibilidad de distribuir contenidos ejecutables, de manera que los gestores de información de la web puedan crear páginas de hipertexto con una interacción continua al mismo tiempo. Cumple con los requisitos para entrega de contenidos interactivos mediante el uso de applets que se incluyen en páginas HTML. Las clases de Java admiten estos protocolos y formatos. El envío de las clases de Java a través de internet se realiza con facilidad, debido a la existencia de una interfaz unificada.

Una applet es una pequeña aplicación en Java transferida a través de internet e insertada en páginas HTML, que hacen posible que los programadores ejerzan control sobre los programas ejecutables de Java; esto es, una funcionalidad que no fácilmente se encuentra en otros lenguajes.

De aplicación general. Java cuenta con un gran paquete de librerías que cubren una amplia gama temática. Entre ellos, paquetes gráficos como el AWT (Abstract Window Toolkit) y Swing, que brindan componentes de interfaz gráfica de usuario, que por lo general se hace común en la mayoría de las compu-

[3] Las partes textuales de este apartado fueron tomadas de: Horstmannn, Cay S., Cornell, Gary, *Java 2 Fundamentos*, Prentice Hall, España, 2003, pp. 6-13. Los autores lo sintetizaron a su vez del "Libro blanco", disponible en http://java.sun.com/doc/language_environment

tadoras. Las utilidades de Java son un conjunto de estructuras de datos complejas y sus métodos asociados son de gran ayuda para desarrollar e implementar applets y otras aplicaciones más complejas. También se consideran estructuras de datos las pilas y tablas hash, que son clases ya implementadas. Java hace mucho énfasis en las clases, la encapsulación, la herencia, el polimorfismo, la gestión de memoria, la gestión de excepciones y la concurrencia.

Dinámico. En varios aspectos Java es un lenguaje más dinámico que C o C++. Las bibliotecas pueden añadir métodos nuevos e instanciar variables sin que afecte a los clientes.

Se suele hablar de Java 1, Java 2 y Java 3D. En realidad, la numeración tiene que ver más con una cuestión comercial que técnica. El nombre de Java 2 se emplea para hacer hincapié en el crecimiento de Java como una plataforma amplia y probada a partir de la versión 1.2 del compilador. Java 3D, por su parte, contiene una serie de clases para creación de imágenes gráficas. Se puede decir que es independiente por completo de Java 2.

"Sun desarrolló la primera versión de Java a principios de 1996. La gente se dio cuenta con rapidez de que Java 1.0 no iba a servir para desarrollar aplicaciones serias. Es verdad que se podía utilizar Java 1.0 para hacer un applet en el que apareciese el texto moviéndose en forma temblorosa de acá para allá en un lienzo. Pero ni siquiera se podía imprimir en Java 1.0. Para ser francos, Java 1.0 no estaba listo para distribuirse en forma masiva. Su sucesor, la versión 1.1, arregló la mayor parte de los defectos obvios, mejoró de manera notable la capacidad de reflexión y añadió un modelo nuevo de eventos para la programación de la GUI (interfaz gráfica de usuario). Aun así, todavía tenía muchas limitaciones.

"La gran noticia de la conferencia JavaOne de 1998 fue la inminente versión 1.2 de Java, que reemplazaba esa especie de juguete que era la GUI y las herramientas gráficas con versiones sofisticadas y escalables que se acercaban más a la promesa de 'Escribir el código una sola vez, ejecutarlo en cualquier sitio' (*Write Once, Run Anywhere*) que lo logrado por sus predecesores. Tres días más tarde (!), esto ocurría en diciembre de 1998, el departamento de marketing de Sun cambió el nombre al término pegadizo **Java 2 Standard Edition Software Development Kit Version 1.2.**" El cuadro 4.2 indica el crecimiento de Java a lo largo del tiempo hasta la versión 1.4; el ritmo de crecimiento no se ha detenido.[4]

Cuadro 4.2 El crecimiento de Java a lo largo del tiempo

Versión	Número de clases e interfases	Número de métodos y campos
1.0	212	2 125
1.1	504	5 478
1.2 (Java 2)	1 781	20 935
1.3	2 130	23 901
1.4	3 020	32 138

El entorno de desarrollo Java

Java es un lenguaje de programación orientado a objetos provisto de un entorno para desarrollo, un entorno de ejecución de aplicaciones y un entorno de despliegue de aplicaciones. Es un lenguaje adecuado para aplicaciones en internet que puede ejecutarse en varias plataformas. Se basa en librerías al igual que el lenguaje C. Sin embargo, incorpora un concepto nuevo: la máquina virtual. El compilador no produce en forma directa el código ejecutable. En su lugar deja un código de bytes que —en un segundo paso— un intérprete transformará al código máquina de la computadora. Por eso se dice que la compilación es hacia una "máquina virtual". En otras palabras, compilamos para un intérprete que aún no se conoce (véase figura 4.6).

[4] Ibid, p. 17.

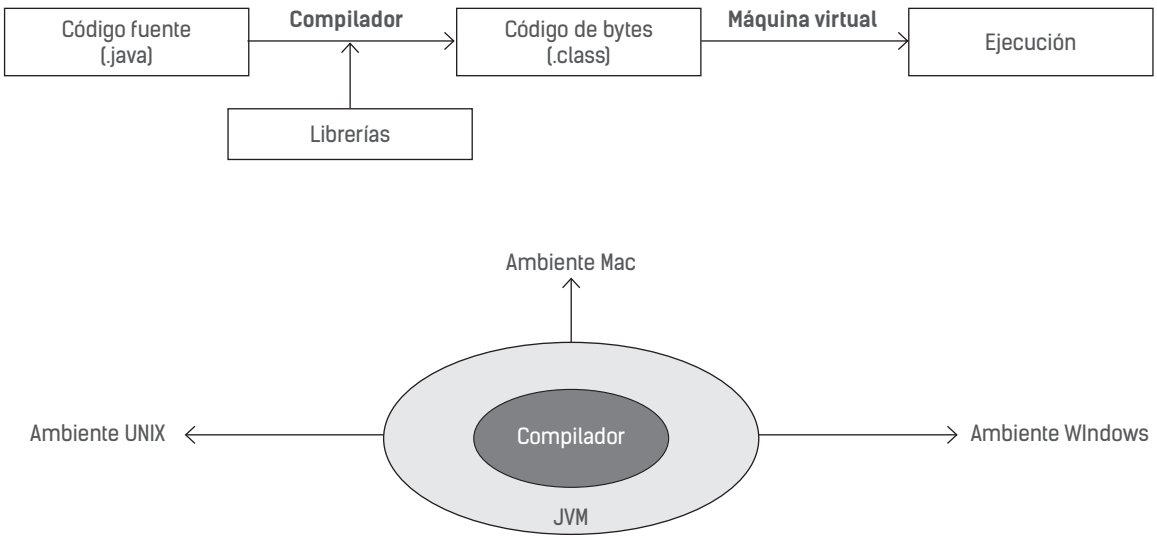


Figura 4.6 El entorno de desarrollo de Java (imagen simplificada).

Los programas escritos en lenguaje Java no se traducen a archivos ejecutables (.exe), sino a archivos (.class), interpretables solo por la Máquina Virtual de Java (JVM). El código de los programas Java se escribe en archivos de texto que se guardan con la extensión (.java). El compilador traduce uno a uno estos archivos .java a archivos binarios en un lenguaje intermedio llamado Java Bytecode (*.class), muy cercano al lenguaje máquina. La JVM interpreta los archivos binarios, ejecutando la aplicación de manera normal sobre la plataforma de destino (UNIX, Windows, Mac…).

La idea no era nueva y había fracasado en intentos anteriores. Sin embargo, la situación técnica de la computación hacia finales de la década de1990 y la aplicación masiva de internet hicieron que Java tuviera una difusión que ningún otro lenguaje había logrado.

Esta forma de trabajo tiene grandes ventajas. Por un lado, ni el programa fuente ni el código compilado dependen de la computadora en específico. Por otro, es mucho más difícil vulnerar la seguridad en Java. Pero tiene costos: se requiere un intérprete que realice el paso final y hace que los programas sean menos eficientes por ese último paso de traducción. Sin embargo, con la tecnología actual de Java, ese costo se ha reducido cada vez más.

La figura 4.6 presenta un entorno de desarrollo simplificado, ideal para un primer acercamiento didáctico. No obstante, también es útil tener una visión más detallada (véase figura 4.7).

La **Interfaz de programación de aplicaciones (Java API)** facilita a los programadores herramientas para el desarrollo de aplicaciones. Está provisto de un grupo de clases utilitarias para efectuar diferentes tareas al momento de la programación. Las API están organizadas en paquetes que contienen un conjunto de clases relacionadas en forma semántica. La serie de bibliotecas de software (Core Java) son subprogramas que manejan cadenas, archivos, procesos ysupervisan acciones entrada/salida, así como la seguridad del sistema.

El Java Development Kit (JDK) contiene una serie de herramientas de desarrollo como el compilador, el depurador y el generador de documentación. El Entorno de ejecución (JRE) es el encargado de cargar el archivo que genera el compilador **.class** y verifica los bytecodes que validan, interpretan y ejecutan el código. Luego traducen las instrucciones a lenguaje máquina para que puedan ser ejecutadas por la computadora.

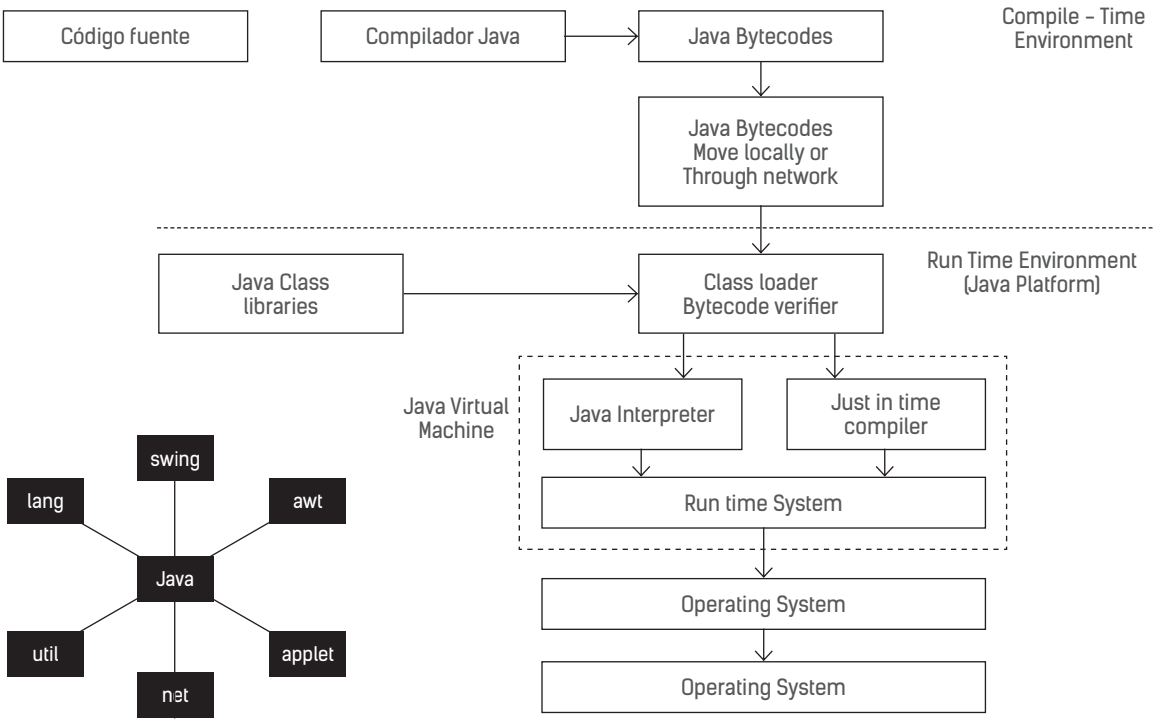


Figura 4.7 El entorno de desarrollo de Java (imagen general).

Java es independiente de la plataforma; es decir, cuenta con herramientas poderosas que permiten programar con una independencia total de la plataforma tanto de hardware como del sistema operativo en que se desarrolle o programe. La **Máquina Virtual de Java** (JVM) es una máquina implementada por software en una máquina real. El código para la máquina virtual se almacena en archivos con extensión **.class**, mismos que contienen el código necesario para una clase pública:

- Paquete **lang**. Clases con funcionalidades básicas (arrays, cadenas de caracteres, entrada/salida, excepciones, hilos...).
- Paquete **útil**. Utilidades (números aleatorios, vectores, propiedades del sistema...).
- Paquete **net**. Conectividad y trabajo con redes (sockets, URL...).
- Paquete **applet**. Desarrollo de aplicaciones ejecutables en forma directa en navegadores web.
- Paquetes **awt** y **swing**. Desarrollo de interfases gráficas de usuario y muchos más aspectos orientados a objetos.

Una aplicación por lo regular está integrada por archivos, bibliotecas y recursos que utiliza el programa (de manera opcional se incluyen los archivos de código fuente). Los directorios o carpetas que suelen usarse en un proyecto se muestran a continuación:

•src	•bin	•lib	•test	•doc
Fuentes (.java)	Binarios (.class)	Bibliotecas (.class o .jar)	Pruebas	Documentación

El JDK contiene una herramienta llamada .jar que permite empaquetar todos estos archivos en uno solo (.jar) para facilitar su distribución y ejecución.

El Java Development Kit (JDK)

Para entrar de lleno a la parte aplicativa de Java es necesario instalar el compilador o intérprete Java. Sun Microsystems provee en forma gratuita un kit de desarrollo en el Java Development Kit (JDK), que provee y facilita una serie de herramientas, documentación y utilidades para desarrollar aplicaciones en Java. El JDK es fácil de instalar y se puede obtener en la dirección de internet: **http://java.sun.com**. Es importante resaltar que aquí se puede encontrar una gran cantidad de versiones del JDK para las diversas plataformas de sistemas operativos que existen, por lo cual debe elegirse la versión que se necesite de acuerdo con el entorno o plataforma del sistema operativo en el que se quiera instalar.

Para el caso de los ejemplos que aquí se muestran se utilizó el instalador: jdk-7u45-windows-i586, que al ejecutarse solicita una serie de acciones sencillas que hay que seguir y al terminarlas quedará instalado el JDK, y solo faltará definir las variables de entorno para que el sistema operativo ejecute el programa de Java desde cualquier directorio. En caso contrario, solo se podrá ejecutar el compilador desde la ubicación <directorio de Java>\bin.

PATH: Variable que indica la ruta para ejecutar los programas.
CLASSPATH: Indica al compilador la ruta donde se encuentran los archivos de clase (*.class).

En el caso de Windows la opción para lograr esto se encuentra de la siguiente forma: a) ir a Panel de control; b) ir a la opción sistema; c) dentro de opciones avanzadas ir a variables de entorno; d) agregar el directorio donde se encuentra el compilador de Java en la variable Path, cuidando de no borrar las otras variables de entorno.

Una de las formas para verificar que se ha instalado el compilador en forma correcta es invocando el comando java —versión desde cualquier directorio en la línea de comandos (véase figura 4.8).

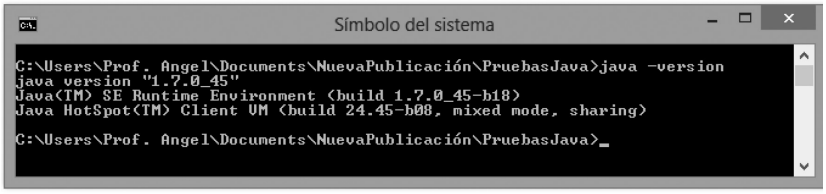


Figura 4.8 Verificando la instalación del compilador de Java.

El JDK provee herramientas para realizar diferentes acciones apegándose a su sintaxis definida con anterioridad. Por ejemplo, si utilizamos la consola **DOS** o **cmd** para ejecutar o compilar un programa en Java y queremos ver su sintaxis, solo tendremos que invocar la herramienta de ayuda en la línea de comandos. En este caso:

```
C:>java -?      oC:>java -help
```

Como se puede apreciar, la sintaxis para su utilización es la siguiente:

`java [-options] class [args…]`
Opciones (-options): forma en que el intérprete Java ejecuta el programa.
Clase (class): define la clase cuyo método main() se desea ejecutar como programa.
Argumentos (args…): define los argumentos a recibir en el parámetro del método **main(String s).**

Para compilar archivos de código fuente —identificados por la extensión **.java**— y convertirlos en archivos de clases —identificados por la extensión **.class**— es necesario crear un archivo de clase para cada clase de naturaleza pública.

Para el caso del compilador de Java, tenemos la siguiente sintaxis:

javac [options] sourcefiles
Opciones (-options): define cómo creará el compilador las clases ejecutables.
Archivos a compilar (sourcefiles): archivo fuente a compilar, con extensión .java.

Entre otras de las herramientas del JDK está el **depurador**, que permite liberar de errores un programa o aplicación escrita en Java. Es similar al depurador **gdb** utilizado por C/C++, su sintaxis es:

jdb [options]
Opciones (options): define los ajustes necesarios durante la depuración.

El **desensamblador** facilita desensamblar un archivo de clase **.class**. Su utilidad se manifiesta cuando no se tienen los códigos fuente de una clase. Su sintaxis es:

`javap [options] [classes]`
Opciones (options): define cómo han de desensamblar las clases.
Clase (clases): define la ruta de las clases a desensamblar.

El **visualizador de applets** permite realizar pruebas de los mismos applets, facilitando su previsualización como si se tuviera algún navegador de internet. La sintaxis para su activación es:

`appletviewer [options] applet`
Opciones (options): especifica cómo ejecutar la applet Java.
Applet: indica que contiene una página HTML con una applet Java.

El **generador de cabecera** se utiliza para crear archivos de cabecera C/C++ (en código nativo) e implementar en esos lenguajes los métodos nativos que presente un programa Java. La sintaxis del generador de cabeceras **javah**, es:

`javah [options] classes`
Opciones (options): forma en la que se generarán los archivos fuente.
Clase (clases): clase desde la que se van a generar archivos fuente C.

El **generador de documentación** facilita la generación de documentación desde el código fuente Java. Se pueden obtener páginas HTML en función de declaraciones y comentarios definidos en las líneas de comentarios. La sintaxis que tiene es:

`javadoc [options] packagenames`
Opciones (options): se refiere a qué tipo de documentación se generará.
Nombre de paquete (packagenames): Paquete de código fuente en Java del que se generará la documentación.

Cabe destacar que conforme se fueron actualizando las versiones del JDK también aumentaron de manera significativa las capacidades del lenguaje. Por mencionar algunas:

- Ampliación del paquete de clases de JFC (Java Foundation Classes).
- Aparecen nuevos paquetes de la API de Java:
  - **Swing:** paquete de gráficos.
  - **Java 2D:** ampliación de AWT.

- **Java Collections:** incluye clases que representan estructuras de datos: Vector, ArrayList, ArraySet, TreeMap...
- Compatibilidad entre versiones en código y en ejecución.
- El compilador genera código con mayor optimización.
- El entorno de ejecución es más rápido.
- Solución a errores de versiones pasadas.
- IDL (Interface Definition Language): Interacción con CORBA en forma simple y práctica que utiliza Java.
- JCE (Java Criptography Extensions): mejores posibilidades de seguridad.
- RMI (Remote Method Invocation): realiza acciones sobre objetos remotos, incluso mediante SSL (Security Socket Layer).

Un primer programa de ejemplo

Ya tenemos las bases conceptuales. ¡Es hora de compilar e interpretar nuestro primer programa escrito en lenguaje Java! En este caso el bloc de notas que se encuentra en la carpeta de accesorios será la herramienta que utilizaremos para escribir nuestro código fuente de Java (recuerde que el archivo debe llevar extensión .java). Ahí escribiremos el código de Java del programa 4.2, cuidando de colocar comillas, llaves y puntos y coma en el lugar adecuado.

```
/* Programa 4.2 Un primer programa en Java
Codificación del programa de bienvenida en Java */
public class Bienvenida {
public static void main (String[] args) {
System.out.println("Bienvenidos al mundo" +
   "de la programación en lenguaje Java");
   }
}
```

Una breve descripción de lo que se está escribiendo: la palabra `public class` señala que se va a escribir una clase pública (accesible para todas las demás clases); `Bienvenida` indica el nombre de la clase, que será de manera automática el nombre del archivo por tratarse de una clase pública; la palabra `static` señala un elemento estático; es decir, un solo elemento `main` sin importar cuántos objetos se creen en memoria; `void` indica que la rutina `main` no tiene ningún valor de retorno; `String[] args` permite que esta rutina reciba parámetros tipo cadena al ser invocados desde el entorno del sistema operativo; `System.out.println` da la posibilidad de desplegar textos en la pantalla, por lo común se combina con el operador para concatenar (+).

Una vez que se escribió el código en nuestro bloc de notas procederemos a guardarlo con el nombre Bienvenida.java en alguna carpeta previamente definida para este fin (recuerde que debe tener extensión .java y llamarse igual a la clase, incluyendo el uso de mayúsculas y minúsculas). Con esto estaremos listos para compilar nuestro primer programa Java, para lo cual tendremos que utilizar la consola **DOS** o **cmd**, como ya se explicó con anterioridad. Una vez activada la consola del DOS, en la línea de comandos se invocará al compilador a través de la instrucción `javac` seguida del nombre del archivo que contiene nuestro programa en Java.

Sintaxis: **`javac Bienvenida.java`**

Si hay errores, tendrían que corregirse en bloc de notas. En caso contrario, aparecerá de nueva cuenta la línea de comandos, lo cual quiere decir que todo es correcto y se generó en forma interna el archivo Bienvenida.class, que contiene el programa en código de bytes.

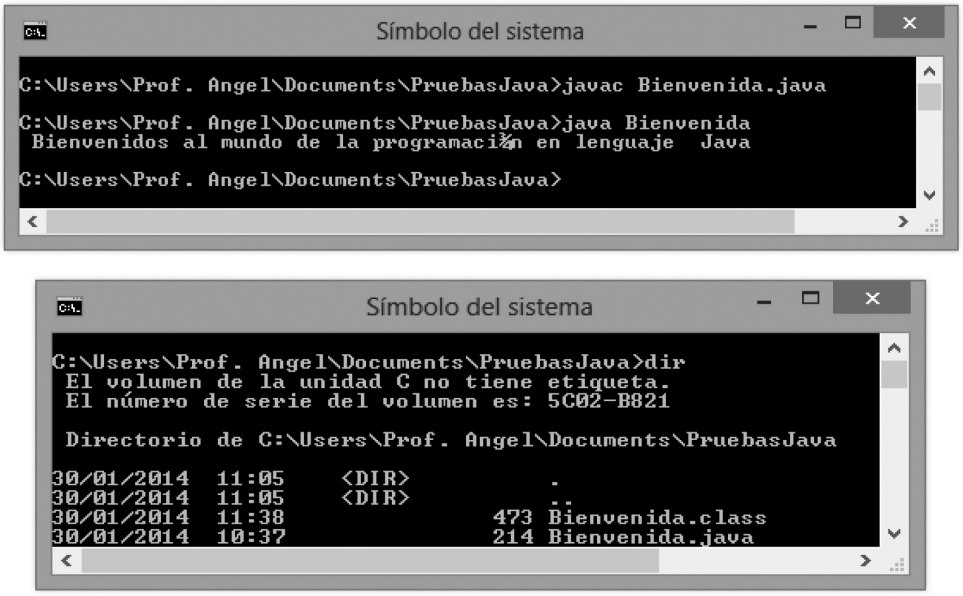


Figura 4.9 Compilación y ejecución del primer programa en Java.

Para verificar que el programa efectivamente funciona, en la línea de comandos llamaremos a la máquina virtual de Java a través de la instrucción: **`java Bienvenida`** y nuestro primer programa enviará el mensaje:

Bienvenidos al mundo de la programación en lenguaje Java

Con estos sencillos pasos, comprobamos que el compilador Java funciona **correctamente** y por supuesto hemos hecho nuestro primer programa en lenguaje Java (véase figura 4.9).

Entornos integrados de desarrollo para Java

Es conveniente trabajar en modo consola para un primer acercamiento con Java, pues ahí se notan con toda claridad los pasos de compilación y ejecución por separado, así como los archivos .java y .class.

No obstante, los entornos integrados de desarrollo facilitan en gran medida las labores y las hacen más amigables a través de una amplia gama de ayudas. Los IDE para Java pueden ser hacia el uso básico o para usos más específicos (como el Visual J + + hacia tecnología Microsoft). Cabe aclarar que en casi todos los casos el compilador es el mismo —el JDK— y que los IDE para Java en ocasiones se han discontinuado (como el Sun Java Studio Creator), son de poco uso en estos momentos (como JBuilder) o aplican a casos muy específicos (como J#, un compilador de versión propietaria de Java que respeta las convenciones del lenguaje pero únicamente funciona para plataforma Windows).

Entre los IDE vigentes y aplicables para un curso de Java básico se encuentran JCreator, Eclipse y Netbeans. El primero es más sencillo y menos exigente en el consumo de recursos, aunque al mismo tiempo es menos potente. Los otros dos siempre pelean por ser el referente de mercado para desarrollos profesionales en este nicho de mercado. La batalla a nivel comercial sin duda la ha ganado Eclipse, aunque parece ser que eso se ha debido más a mejores estrategias de penetración de mercado que a una valoración técnica, en la cual ambos terminan bien evaluados.

JCreator (http://www.jcreator.com/), de Xinox Software. Es práctico, amigable y consume pocos recursos, gratuito en su versión básica, aunque no tan completo como Eclipse o Netbeans. Sin lugar a dudas, una buena opción para comenzar en Java (véase figura 4.10).

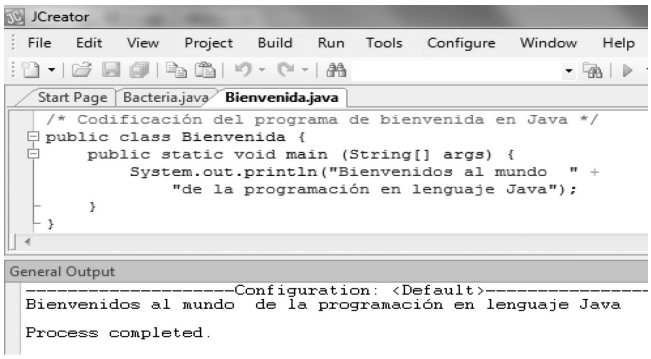


Figura 4.10 Primer programa en Java, ahora utilizando JCreator.

Eclipse (https://www.eclipse.org/), originalmente impulsado por IBM y ahora por la Fundación Eclipse sin fines de lucro. Es un entorno de Desarrollo Integrado (IDE) multiplataforma para desarrollo y compilación de aplicaciones Java, cuenta con una interfaz que lo hace fácil y amigable. Eclipse está construido por completo en Java, por lo que sus alcances son muy extensos. Comenzó como un proyecto de IBM y su código fue liberado en 2001. Eclipse IDE for Java Developers es una distribución concreta de Eclipse para desarrollar aplicaciones Java.

NetBeans (https://netbeans.org/) es un IDE utilizado por muchos desarrolladores y programadores. Permite escribir, compilar, depurar y ejecutar programas. Al igual que Eclipse, está escrito en Java y es un producto libre, gratuito y sin restricciones de uso. En esta plataforma, es posible elaborar aplicaciones de escritorio, para ambientes web y dispositivos portátiles como móviles, tablets u otros, sin que cambie la forma de programación.

EJERCICIOS SUGERIDOS

- Instale el compilador, verifique que haya quedado instalado en forma correcta y ejecute el primer programa de ejemplo.
- Realice un programa similar al expuesto (puede ser el clásico "Hola mundo") sin consultar materiales de apoyo. Trate de depurar los errores que surjan.
- Instale el Entorno Integrado con el que trabajará y verifique que ambos ejemplos hayan corrido en forma adecuada.

REFLEXIÓN 4.2

La constante confusión entre la máquina virtual, el compilador y el IDE

En muchos estudiantes que ya tomaron el curso de Java existe una confusión entre la máquina virtual de Java, el compilador y el IDE, confusión que permanece desde la instalación de Java hasta la compilación. Por eso recomendamos que los primeros programas se ejecuten desde la línea de comandos, para que se noten con claridad los dos pasos que se dan por separado: la compilación y la ejecución, así como los archivos que se van creando en el proceso. Después, ya es conveniente ocupar el IDE elegido.

4.4 Una primera clase en C++ y Java

Una primera clase en C++

El primer programa realizado sirvió solo para verificar que el entorno de desarrollo trabaja en forma adecuada. Aunque está en Java, no explota en lo absoluto el paradigma de la programación orientada a objetos.

Ahora incorporaremos un segundo ejemplo para entender cómo trabaja una clase, conforme el diagrama del cuadro 4.3. Conviene recordar que la sintaxis corresponde a UML y no coincide con la de C++ ni Java, por lo cual hay que adecuarla a las reglas del lenguaje de programación en que se realizará la codificación.

Cuadrado
lado: float –
(Cuadrado (dato: float + perimetro ( ) : float + area ( ) : float + consultalado ( ) : float + (cambialado (dato: float +

Cuadro 4.3 Diagrama de la clase cuadrado.

Hay otro aspecto sobre el cual se seguirá insistiendo a lo largo del libro. No estamos trabajando sobre un requerimiento del usuario propiamente dicho, sino creando un componente y haciendo un programa principal de prueba para verificar que este trabaja en forma adecuada. De manera metafórica, construimos un "foco" y un "probador de focos". Ignoramos cómo será la instalación eléctrica de la casa que se está construyendo, pero nuestro foco servirá para esa casa y mediante la clase que diseñamos se podrán crear los focos que sean necesarios. Por eso nuestro procedimiento de trabajo en este caso será a través del diagrama de clase, al cual se llegó después de un procedimiento de requerimientos y análisis, que queda esbozado de manera general en el capítulo 7.

```
Programa 4.3 Programación de una clase cuadrado en C++ (versión 1)
#include "iostream.h"
#include "conio.h"
class cuadrado {
   private:
      float lado;
   public:
      cuadrado (float dato) { lado = dato; }
float perimetro () { return lado * 4.0; }
float area () { return lado * lado; }
void cambialado (float x) { lado = x; }
float consultalado () { return lado; }
};
main(void) {
float auxiliar;
   cout << "Este programa crea una clase cuadrado.\n";
   cout << "Teclee el valor del lado: ";
   cin >> auxiliar;
   cuadrado cuadro(auxiliar); /* creación del objeto */
   cout << "El perímetro es: " << cuadro.perimetro() << endl;
   cout << "El área es: " << cuadro.area() << endl;
   cuadro.cambialado(8);
   cout << "Después de cambiar el valor del lado a 8" << endl;
   cout << "El perímetro es: " << cuadro.perimetro() << endl;
   cout << "El área es: " << cuadro.area() << endl;

   printf("\nOprima cualquier tecla para terminar...\n");
   getch();
}
```

```
Este programa crea una clase cuadrado.
Teclee el valor del lado: 7
El perýmetro es: 28
El Órea es: 49
DespuÚs de cambiar el valor del lado a 8
El perýmetro es: 32
El Órea es: 64

Oprima cualquier tecla para terminar...
```

Figura 4.12 Programación de una clase cuadrado en C++ (versión 1).

El código en lenguaje C quedaría como se expresa en el programa 4.3, elaborado en el compilador Dev C++ (que marcará una advertencia, la cual dejaremos pasar por el momento). Observe que la palabra `class` identifica a la clase. Las partes privadas y públicas quedan identificadas con claridad a través de las palabras reservadas `private` y `public`, respectivamente; los elementos privados solo son accesibles desde la misma clase (quedan "encapsulados"), mientras que los elementos públicos pueden invocarse desde otras clases. El constructor se identifica con el mismo nombre de la clase y en este caso recibe el valor del lado como parámetro de inicialización. Cabe aclarar que si no hubiera un constructor explícito, el compilador creará un constructor implícito sin parámetros.

Como la clase es un tipo de dato, la declaración de un objeto es similar a la de cualquier otra variable. En nuestro caso: `cuadrado cuadro(auxiliar)`. En C++ el `printf` y el `scanf` se sustituyen por `cout` y `cin`, respectivamente. Sin embargo, se pueden emplear las opciones anteriores. En este caso, el detalle de los métodos se da en forma directa al definirlos, lo cual resulta muy práctico. No obstante, si hubiera un código muy grande para cada subrutina, entonces se perdería la visión general de la clase. En esas situaciones, tal vez sea preferible emplear una sintaxis alternativa, que se muestra en el programa 4.4. Como es lógico suponer, el cambio de sintaxis no modifica en ningún momento a las rutinas que hacen uso de la clase; en nuestro caso, el programa principal.

```
Programa 4.4Programación de una clase cuadrado en C++ (versión 2).
#include "iostream.h"
#include "conio.h"
class cuadrado {
   private:
      float lado;
   public:
      cuadrado (float dato);
   float perimetro ();
   float area ();
   void cambialado (float x);
   float consultalado ();
};
float cuadrado::consultalado() {
   return lado;
}
void cuadrado::cambialado (float x) {
   lado = x;
}
float cuadrado::perimetro() {
   return lado * 4.0;
}
float cuadrado::area() {
   return lado * lado;
}
cuadrado::cuadrado(float dato) {
   lado = dato;
}
main(void) {
   float auxiliar;
   cout << "Este programa crea una clase cuadrado.\n";
   cout << "Teclee el valor del lado: ";
   cin >> auxiliar;
   cuadrado cuadro(auxiliar); /* creación del objeto */
   cout << "El perímetro es: " << cuadro.perimetro() << endl;
   cout << "El área es: " << cuadro.area() << endl;
```

```
 cuadro.cambialado(8);
 cout << "Después de cambiar el valor del lado a 8" << endl;
 cout << "El perímetro es: " << cuadro.perimetro() << endl;
 cout << "El área es: " << cuadro.area() << endl;
 printf("\nOprima cualquier tecla para terminar...\n");
 getch();
}
```

El programa 4.4, aunque se ejecuta en forma adecuada, no tiene la mejor estructura. Podríamos suponer que el diseñador del sistema encargó la construcción de la clase Cuadrado. El programa principal (main) que se presentó tiene la finalidad de probar que dicha clase funciona de manera adecuada. No es deseable juntar en una clase ambas partes, pues mientras la clase se integrará al código del sistema final, la sección que corresponde a la prueba será desechada al cumplir su cometido o, en el mejor de los casos, quedará colocada en una carpeta de pruebas para tener una evidencia objetiva de la calidad del sistema. Por ello es conveniente dividir el código; lo haremos, por cuestiones pedagógicas, en Cuadrado2 y TestCuadrado2, para así conservar intacta la clase Cuadrado. Como es lógico, la clase Cuadrado2 será compilada, mientras la clase TestCuadrado2 será compilada y ejecutada. El código se muestra en los programas 4.5 y 4.6.

Si el compilador no puede abrir los programas con extensión .h, como es el caso de Dev C++, recurra a la siguiente estrategia: nombre el archivo como cuadrado2.cpp, compílelo para garantizar que no tiene ningún error y después cambie la extensión a cuadrado2.h.

```
//Programa 4.5 La clase cuadrado2.h en C++.
class cuadrado2 {
   private:
      float lado;
   public:
      cuadrado2 (float dato) { lado = dato; }
      float perimetro () { return lado * 4.0; }
      float area () { return lado * lado; }
      void cambialado (float x) { lado = x; }
      float consultalado () { return lado; }
};
```

```
Programa 4.6 El programa textCuadrado2 en C++.
#include "cuadrado2.h"
#include "iostream.h"
#include "conio.h"
main(void) {
float auxiliar;
   cout << "Este programa crea una clase cuadrado2.\n";
   cout << "Teclee el valor del lado: ";
   cin >> auxiliar;
   cuadrado2 cuadro(auxiliar); /* creación del objeto */
   cout << "El perímetro es: " << cuadro.perimetro() << endl;
   cout << "El área es: " << cuadro.area() << endl;
   cuadro.cambialado(8);
   cout << "Después de cambiar el valor del lado a 8" << endl;
   cout << "El perímetro es: " << cuadro.perimetro() << endl;
   cout << "El área es: " << cuadro.area() << endl;

   printf("\nOprima cualquier tecla para terminar...\n");
getch();
}
```

El ejemplo anterior no requirió una de las posibilidades comunes en la creación de clases en C++: el uso del destructor, que sirve para eliminar de la memoria los elementos dinámicos que fueron creados vía apuntadores durante la implementación de un objeto.[5] El destructor lleva el mismo nombre de la clase antecedida por una tilde (~).

Para ello retomaremos una idea del libro de Ernesto Peñaloza,[6] a la cual le haremos adaptaciones para hacer más notoria la labor del destructor, conforme puede verse en el programa 4.7. La clase desplegar Mensaje tiene la declaración de un apuntador hacia una cadena como atributo privado; el constructor recibe una cadena como parámetro y crea una copia hacia el atributo privado; el método desplegar muestra esa cadena en pantalla, mientras que el destructor elimina ese objeto de la memoria, que de otra forma quedaría latente y sin ningún uso posible.

Observe en la pantalla que muestra la ejecución que, justo cuando se da por finalizado el programa principal, se destruye el objeto. Planteado de otra forma: un instante antes de eliminar de la memoria la subrutina main que ha concluido es "disparado" el destructor.

```
Programa 4.7 Ejemplo del uso de la clase destructor de C++.
#include "iostream.h"
#include "conio.h"
class desplegarMensaje {
   private:
      char *mensaje;
   public:
      desplegarMensaje (char *texto) {
         int longitud;
         if (texto != NULL) {
            longitud = strlen(texto);
            mensaje = new char[longitud+1];
            strcpy(mensaje, texto);
         }
         else {
            texto = new char[1];
            texto[0] = '\0';
         }
      }
      void desplegar() {
         cout << mensaje << endl;
      }
      ~desplegarMensaje() {
         delete mensaje;
         cout << "Se destruyó el objeto";
         getch(); // una pausa al destruir el objeto
      }
};
int main() {
   cout << "Crearemos el objeto con el mensaje hola." << endl;
   desplegarMensaje auxiliar ("hola");
   cout << "Desplegaremos el mensaje que se almacenó: ";
   auxiliar.desplegar();
   cout << "Ha finalizado el programa. Oprima cualquier tecla...\n";
   getch(); // una pausa antes de terminar el programa
}
```

[5] Una de las diferencias entre C++ y Java es que C++ utiliza el destructor, mientras que Java deja esa labor a un programa recolector de basura (*garbage collection*).

[6] Peñaloza, Ernesto, *Fundamentos de programación C/C++*, Alfaomega, cuarta edición, México, 2004.

```
Crearemos el objeto con el mensaje hola.
Desplegaremos el mensaje que se almacen%: hola
Ha finalizado el programa. Oprima cualquier tecla...
```

Figura 4.13 Ejemplo del uso de la clase destructor de C++.

Una primera clase en Java

Ahora mostraremos la forma en que quedará el programa en Java. Primero, cambiaremos el diagrama de clase pues, aunque desde el punto de vista conceptual es el mismo, existen diferencias en las convenciones para nombrar los identificadores (véase cuadro 4.4): las clases comienzan con letra mayúscula; las variables y subrutinas comienzan con letra minúscula; se emplea la notación de camello, en la cual la primera letra a partir de la segunda palabra siempre se pone con mayúscula; por último, las subrutinas para consultar y cambiar el valor de un atributo serán `getAtributo` y `setAtributo`, respectivamente (en nuestro caso: `getLado` y `setLado`). También existe cierta costumbre pedagógica de usar más el tipo flotante para textos en C y el tipo double para textos en Java, aunque cabe recalcar que el tipo de dato adecuado en un sistema real debe ser el que corresponde a nuestra necesidad específica.

Cuadrado
– lado: double
+ Cuadrado (dato: double) + perimetro ( ) : double + area ( ) : double + getLado ( ) : double + setLado (dato: double

Cuadro 4.4 Diagrama de la clase cuadrado (versión 2).

En el programa 4.4 se muestra el código en lenguaje Java. La estructura es similar a la del lenguaje C++. Las palabras `private` y `public` identifican a las partes privadas y públicas y luego viene un programa de pruebas.

Las diferencias entre el código en Java con respecto al de C++:

- Comienza con `publicclassCuadrado`, lo cual implica que el archivo fuente se llamará Cuadrado.java y al ser compilado se creará Cuadrado.class.
- Existe una serie de reglas para el nombre de identificadores, establecidos por Sun como un estándar para toda la programación de Java.
  - Todas las clases de naturaleza pública se guardan en un archivo NombreDeLaClase.java.
  - El nombre del constructor siempre será el mismo que el nombre de la clase.
  - Los nombres de las clases comienzan con mayúsculas, al igual que las interfases.
  - Los nombres de atributos y métodos comienzan con minúscula.
  - Se recomienda el uso de verbos para los métodos.
  - La primera letra de cada palabra intermedia debe ser mayúscula. Por ejemplo: nombre Del Metodo.
  - El acceso a los atributos se realiza a través de las palabras `get` y `set`.
  - Las constantes se especifican con mayúscula. Por ejemplo: VALOR_MAXIMO.
  - La ubicación de los paquetes se hace todo con letras minúsculas.
  - Recuerde que tanto C, como C++ y Java, hacen diferenciación entre minúsculas y mayúsculas.
- La rutina principal tiene otra sintaxis: la palabra `static` señala un elemento estático: implica que habrá un solo programa principal `main` sin importar cuántos objetos se creen en memoria; `void` indica que la rutina `main` no tiene ningún valor de retorno; `String[]args` permite que la rutina `main` reciba parámetros tipo cadena al ser invocado desde el entorno del sistema operativo.
- El despliegue varía. `System.out.println` da la posibilidad de desplegar textos a la pantalla; por lo común se combina con el operador para concatenar (+).

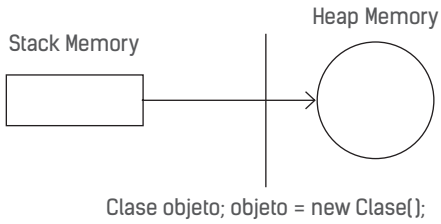


Figura 4.14 Stack Memory y Heap Memory en Java.

- La sintaxis para la creación del objeto difiere. Primero se declarará la variable a utilizar.

```
Cuadrado mosaico;
```

Desde el punto de vista técnico, lo que se define es un apuntador que señalará al objeto mosaico cuando este sea creado en memoria. Después se creará en memoria el objeto propiamente dicho.

```
mosaico = new Cuadrado(7.0);
```

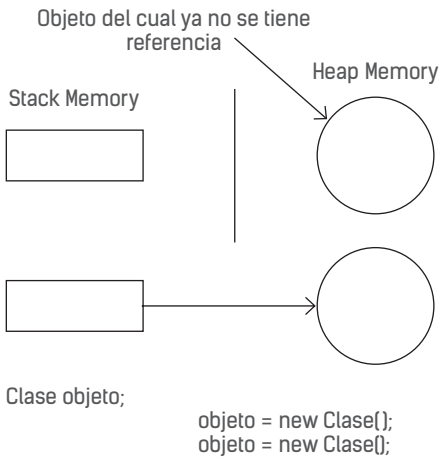


Figura 4.15 El objeto que ya no tiene referencia es eliminado en forma automática por el recolector de basura (*Garbage Collection*).

En Java, todos los objetos se manejan a través de referencias (apuntadores). Cuando se crea un objeto, el nombre de este queda en un espacio de memoria llamado stack. Esta variable, a su vez, señala al objeto, el cual queda alojado en otro espacio de memoria llamado heap (véase figura 4.4).

Cuando se declara la variable objeto, este se da de alta en el stack. Esta variable almacena la dirección de memoria del objeto. El objeto se crea en el heap cuando se da de alta en forma explícita mediante la palabra **new**.

Suponga que se define y crea un objeto. ¿Qué sucedería si el objeto se crea de nuevo sin ser eliminado el primero? Se pierde la primera referencia del primer objeto, el cual ya no es accesible por ninguna vía. En otros lenguajes —como lenguaje C— este objeto seguiría consumiendo memoria hasta que se apague la máquina. En el caso de Java, la memoria se liberará después a través del recolector de basura (*Garbage Collection*). El objetivo del recolector de basura es eliminar de la memoria las referencias y objetos que ya no se usan (véase figura 4.15).

```
//Programa 4.8 Programación de una clase Cuadrado en Java (versión 1).
public class Cuadrado {
   private double lado;
   public Cuadrado(double dato) {
      lado = dato;
   }
   public double getLado() {
      return lado;
   }
   public void setLado(double dato) {
      lado = dato;
   }
   public double area() {
      return lado * lado;
   }
   public double perimetro() {
      return lado * 4;
   }
   public static void main(String[] args) {
      Cuadrado mosaico;
      mosaico = new Cuadrado(7.0);
      System.out.println("Este programa crea una clase cuadrado.\n " +
         "El valor del lado es 7.0, \n el perímetro es: " +
```

```
      mosaico.perimetro() + "\n y el área es: " + mosaico.area());
      mosaico.setLado(8.0);
System.out.println("Después de cambiar el valor del lado a 8\n " +
        "el perimetro es " + mosaico.perimetro() +
        "\n y el área es: " + mosaico.area());
      mosaico.setLado(8.0);
   }
}
```

En forma muy parecida al caso que ya vimos en C++, el programa 4.8, aunque se ejecuta en forma adecuada, no tiene la mejor estructura. Podríamos suponer que el diseñador del sistema encargó la construcción de la clase Cuadrado. El programa principal (main) que se presentó únicamente tiene la finalidad de probar que dicha clase funciona de manera adecuada. No es deseable juntar en una clase ambas partes, pues mientras la clase se integrará al código del sistema final, la sección que corresponde a la prueba será desechada al cumplir su cometido o, en el mejor de los casos, quedará colocada en una carpeta de pruebas para tener una evidencia objetiva de la calidad del sistema. Por ello es conveniente dividir el código; por cuestiones pedagógicas lo haremos en Cuadrado2 y TestCuadrado2, para conservar intacta la clase Cuadrado. Como es lógico, la clase Cuadrado2 será compilada, mientras la clase TestCuadrado2 será compilada y ejecutada. El código se muestra en los programas 4.9 y 4.10.

Observe que hemos incorporado una instrucción para leer datos del teclado, contenida en el paquete `javax.swing` (el paquete es equivalente a una librería y la palabra `import` a la instrucción include; el asterisco indica que puede utilizar todos los métodos de ese paquete). Aunque por el momento la aplicaremos sin conocer el porqué de su funcionamiento, el cual será explicado en el siguiente capítulo.

REFLEXIÓN 4.3

La lectura de Java en programas de prueba

No es necesario leer datos del teclado para crear programas de prueba, pero hemos observado que es más interactivo y motivador para los estudiantes. Sin embargo, debe recordarse que no se trata de un programa para el usuario final sino un programa de pruebas unitarias. Por otra parte, para entender a cabalidad la lectura de datos en Java es necesario saber el tema de herencia y excepciones, que no se han tocado. Leamos datos del teclado para que los estudiantes no sientan el tema árido, pero con las precauciones del caso.

```
//Programa 4.9 La clase Cuadrado2 en Java.
public class Cuadrado2 {
   private double lado;
   public Cuadrado2(double dato) {
      lado = dato;
   }
   public double getLado() {
      return lado;
   }
   public void setLado(double dato) {
      lado = dato;
   }
   public double area() {
      return lado * lado;
   }
   public double perimetro() {
      return lado * 4;
   }
}
```

```
//Programa 4.10 La clase TestCuadrado2:
   programa de prueba de la clase Cuadrado2.
import javax.swing.*;
public class TestCuadrado2 {
public static void main(String[] args) {
   Cuadrado2 mosaico;
   String entrada = JOptionPane.showInputDialog("¿Cuál es el lado " +
      "del triángulo?");
   double lado = Double.parseDouble(entrada);
   mosaico = new Cuadrado2(lado);
   System.out.println("Este programa crea una clase cuadrado.\n " +
      "El valor del lado es " + mosaico.getLado() +
      "\n el perímetro es: " +
      mosaico.perimetro() + "\n y el área es: " + mosaico.area());
   entrada = JOptionPane.showInputDialog("Dé un nuevo valor: ");
   lado = Double.parseDouble(entrada);
   mosaico.setLado(lado);
   System.out.println("Después de cambiar el valor del lado \n " +
      "el perimetro es " + mosaico.perimetro() +
      "\n y el área es: " + mosaico.area());
   }
}
```

En ocasiones, es confuso si el nombre de un identificador se refiere a un atributo del objeto, a una variable o a un parámetro. Para romper esta ambigüedad se utiliza el operador `this`. Este operador señala que el atributo se refiere al objeto; se emplea como `this.atributo`. En este ejemplo no se le verá mucha utilidad. Sin embargo, es indispensable su uso en diversas ocasiones, además de que algunos ambientes de desarrollo generan el código en forma automática como se muestra en la segunda versión. Por ejemplo, la clase `Cuadrado`, que originalmente está escrita de la siguiente forma:

```
public class Cuadrado {
private double lado;
public void setLado(double dato) {
lado = dato;
 }
 /* aquí va más código */
}
```

También podría escribirse de la siguiente forma:

```
[ENTRA CÓDIGO]
public class Cuadrado {
private double lado;
public void setLado(double lado) {
this.lado = lado;
 }
 /* aquí va más código */
}
```

Sobrecarga en C++ y Java

Tanto C++ como Java admiten hacer sobrecarga de métodos, incluyendo el constructor. De hecho, en C++ también puede hacerse una sobrecarga de operadores. La sobrecarga es una situación muy práctica, pues permite que varios métodos con diferentes parámetros reciban el mismo nombre, lo que da la

impresión que fueran el mismo método. Esto evita que se tengan que programar distintas rutinas con una ligera variación del nombre para hacer lo mismo conceptualmente. Un ejemplo sencillo se proporciona en el programa 4.11, que permite crear un triángulo y obtener su área a partir de sus lados. Observe que el constructor se encuentra sobrecargado: hay una rutina que crea el triángulo a partir de sus tres lados y otra que lo hace a partir de un único dato, para cuando se trata de triángulos equiláteros (en los cuales los tres lados son iguales).

```
//Programa 4.11 Clase triángulo con sobrecarga en el método constructor.
#include<conio.h>
#include<math.h>
#include<iostream>
using namespace std;
class triangulo {
   private:
      double x, y, z;
   public:
      triangulo (double a) {
         x = a; y = a; z = a;
      }
      triangulo (double a, double b, double c) {
         x = a; y = b; z = c;
      }
      double area () {
         double s;
         if ( (x + y >= z) && (x + z >= y) && (y + z >= x) ) {
         s = (x + y +z) / 2;
         return sqrt (s * (s-x) * (s-y) * (s-z));
      }
      else
         return -1;
   }
};

int main(void) {
   triangulo x(4, 4, 4);
   triangulo y(3, 4, 5);
   cout << "El área del triángulo 4-4-4 es: " << x.area() << endl;
   cout << "El área del triángulo 3-4-5 es: " << y.area() << endl;
   cout << "Oprima cualquier tecla para continuar...";
   getch();
}
```

¿Qué pruebas aplicar a las clases, tanto en C++ como en Java?

Las pruebas al software pueden ser bajo diferentes enfoques:

a) **Pruebas de caja negra.** Parten del supuesto de que no se tiene acceso al código y por ello solo se trabaja con lo que el usuario pueda realizar. Incluye pruebas de carga máxima o estrés, en paralelo, de usabilidad, de consistencia de datos, de datos límite, casos normales, casos excepcionales, casos "imposibles".
b) **Pruebas de caja blanca.** La premisa es que se puede ver el código y por tanto es posible verificar cada uno de los casos distintos que involucra el desarrollo (librerías, excepciones, decisiones, etcétera).

La clase de prueba —en este caso Test Cuadrado— debe hacer una mezcla de ambos enfoques. En términos más prácticos, se deben considerar:

- Todos los métodos de la clase.
- Los cálculos que responden a lógica diferente. Por ejemplo, el factorial normal es factorial(n) = 1 * 2 * 3 * …. * (n-1) * n, pero el factorial de 0 tiene su propia lógica: es 1 por definición. Ambas situaciones deben ser consideradas en el programa de pruebas.
- Los casos límite; es decir, cuando se cambian las reglas de cálculo. Citemos un caso representativo: en calificaciones aprobatorias los decimales se redondean al entero más cercano (se dice de modo coloquial que "suben"), pero en calificaciones reprobatorias los decimales se truncan ("bajan"). Si un estudiante tiene las calificaciones parciales de 9, 10 y 10 el promedio final es 10, pero si tiene 5, 6 y 6 el promedio final es 5. El último caso que obedece a una "lógica reprobatoria" es 5, 6 y 6, con promedio final de 5, mientras que el primer caso de una "lógica aprobatoria" es 6, 7 y 7, con un promedio final de 7. Ambas deben formar parte del lote de pruebas.
- Si uno tiene acceso a los algoritmos o al código se debiera incluir al menos un caso para cada decisión, así como los casos límite en donde termina un ciclo o una recursividad.
- Llamar a los métodos con datos inconsistentes, que no son aplicables al cálculo. Por ejemplo, si se está revisando una clase que calcula un factorial, debemos incluir una llamada con números negativos para observar lo que sucede, pues no existen factoriales de números negativos.

REFLEXIÓN 4.4

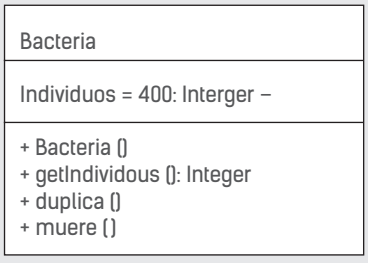
La necesidad de programas de prueba

En términos generales, hemos observado que los estudiantes tienen problemas con las pruebas en programas sencillos: entre 75 y 90% no logran pasar todas las pruebas de escritorio si realizan el código solo en papel (no resuelven bien todos los casos planteados). Cuando se les da acceso al compilador sin poner límite de tiempo el porcentaje disminuye a 23%. Aun así, es demasiado costoso que casi la cuarta parte de los programas tengan que ser reprocesados por no pasar todas las pruebas de "caja negra" que se diseñaron.

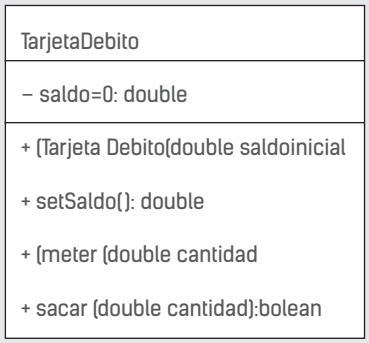
Por ello debe insistirse en que los programas de prueba son el "control de calidad" de las clases y que si no se detectan los errores en esta etapa será mucho más costoso que se perciban cuando estas clases se combinen con otras dentro del sistema.

EJERCICIOS SUGERIDOS

- Capture la clase Cuadrado, compílela y ejecútela en el compilador de C++ de su elección.
- Capture la clase Cuadrado, compílela y ejecútela tanto desde línea de comandos como desde el IDE de su elección.
- Realice el código de la clase Bacteria, con las siguientes especificaciones.
- Esta clase asume que las bacterias se duplican en un periodo determinado (es un comportamiento típico de seres microscópicos); el fallecimiento se da de modo individual. En este caso la población inicial es, de manera forzosa, de 400 individuos (ciertamente es un supuesto demasiado rígido, pero ejemplifica la forma de manejar constructores sin parámetros).



- Suponga que en el programa principal existe un objeto **TarjetaDebito ejemplo1** y otro **double x**, con base en el siguiente diagrama.



a) Indique cuáles instrucciones serían erróneas, suponga que se dan desde otra clase:

```
a) meter(ejemplo1,1000);
b) ejemplo1.meter(1000);
c) x = ejemplo1.meter(1000);
d) ejemplo1.saldo = ejemplo1.saldo + 6000;
```

Respuesta: _____

b) Señale nombre y extensión de los archivos que se crearían en Java al capturar y compilar el programa:

_____

- Complete las palabras que **hacen** falta. Cada inciso corresponde a una palabra.

```
public _____(a)_____ Equilatero {
   private _____(b)_____ lado;
   public _____(c)_____ (double dato) {
      lado = dato;
      System.out.println("Construyendo un triángulo equilátero.");
   }
   public String toString() {
      return "Triángulo equilátero. Lado = " + lado;
   }
   public static void main (String [] args) {
      Equilatero _____(d)_____;
      ejemplo = _____(e)_____ equilatero(8.5);
      System.out.println(ejemplo.toString());
   }
}
```

- Elija, entre los enunciados siguientes, el que corresponde más al término de encapsulamiento.

a) Que una clase tenga atributos y métodos.

b) Que no tengamos acceso al código en un archivo .class.

c) Que no pueda accederse a un atributo privado desde fuera de la clase.

Respuesta: _____

4.5 Diferencias principales entre C++ y Java

En términos generales, Java tiene la sencillez que no posee C++ y de esta manera reduce en forma considerable el número de errores que se cometen en C++. Esto se debe a que los creadores de Java tomaron

en gran medida como base a C++, pero intentaron no trasladar a un nuevo lenguaje lo que consideraron sus deficiencias o debilidades; procuraron hacer un lenguaje portable y que diera facilidad para el desarrollo. Algunas diferencias entre ambos lenguajes ya se mostraron en este capítulo; otras se manejarán en los capítulos siguientes.

Java incorpora una máquina virtual. Como ya se mencionó, existen varios compiladores de C++ que trabajan sobre una plataforma específica y producen códigos ejecutables. En Java solo hay un compilador con distintas versiones y produce código de bytes (extensión .class), que será interpretado por la máquina virtual que corresponda a la plataforma específica; no hay archivos ejecutables. Se dice que Java tiene una arquitectura neutra ya que su código compilado va a un archivo de formato independiente de la arquitectura de la máquina en que se ejecutará.

Java es totalmente orientado a objetos y por ello no utiliza variables globales (en todo caso se utilizan como atributos estáticos de una clase). Por otra parte, Java no intenta conectar todos los módulos que integran una aplicación sino hasta el tiempo de ejecución, mientras C++ enlaza todos los módulos cuando está en la fase de compilación.

Los **apuntadores se manejan en forma distinta.** C y C++ expresan apuntadores de manera explícita (&variable o *variable, según el contexto) y crean variables en forma dinámica con la palabra new, las cuales deben ser destruidas también de manera explícita (con la palabra `delete` y, en su caso, el uso del destructor). En C los apuntadores pueden dirigirse a aspectos de bajo nivel, con una gran potencialidad que no tiene Java, pero también con algunos riesgos: si los apuntadores no son bien controlados pueden propiciar errores de difícil detección al tiempo de ejecución. Por su parte, Java permite en forma automática una gestión ágil de la memoria y tiene operadores nuevos para reservar memoria para los objetos, pero no hay función alguna explícitamente para liberarla: la recolección de basura es una parte integral de Java durante la ejecución de sus programas. Una vez que se ha almacenado un objeto en el tiempo de ejecución y se detecta que en determinado momento ya no tiene ninguna referencia, el sistema vacía ese espacio de memoria para un uso posterior a través del recolector de basura. En síntesis, en Java los nombres de los objetos son en realidad apuntadores implícitos (parámetros por referencia), desaparecen los símbolos & y *, y los objetos no referenciados son liberados de la memoria en forma automática a través del recolector de basura.

Existen diferencias entre los tipos de **datos primitivos** que se manejan. En Java, los tipos de datos primitivos pueden ser numéricos, booleanos o caracteres (véase 4.16). Java maneja tipos de datos numéricos: byte de 1 byte, short de 2 bytes, int de 4 bytes, y los long de 8 bytes; el byte viene a sustituir el tipo char de C++. Los tipos por omisión son **double** e **int** y no existe un tipo sin signo (unsigned) para los números en Java. Los tipos numéricos decimales son el float (8 bytes) y el double (16 bytes). En Java se pueden definir variables con valores Verdadero/Falso o Sí/No parecidos a los de C++. Aunque, a diferencia de C++, estas variables no pueden ser convertidas a datos numéricos.

Cuadro 4.5 Tipos de datos primitivos en Java

Tipo	Descripción	Valor mínimo/máximo
byte	entero con signo	-128 a 127 (1 byte)
short	entero con signo	-32768 a 32767 (2 bytes)
int	entero con signo	-2147483648 a 2147483647 (4 bytes)
long	entero con signo	-922117036854775808 a 92117036854775807 (8 bytes)
float	real de simple precisión	± 3.40282347e+38 a ±.40239846e-45
double	real de doble precisión	± 1.79769313486231570e+308 a ±4.94065645841246544-324
char	caracteres unicode	\u0000 a \uFFFF (2 bytes)
boolean	verdadero o falso	true o false

Para indicar que una cantidad se trata de un tipo de dato flotante, se señala el valor y a continuación se coloca una f (minúscula o mayúscula). Por ejemplo, 0.45f.

Los tipos de datos en Java son diferentes de los de cualquier otro lenguaje:[7]

En Java existe el **tipo de dato cadena** (`String`), con lo cual las cadenas se manejan de manera natural. Habría que aclarar que en términos estrictos las cadenas no son un tipo de dato primitivo, sino una clase. Pero, en términos prácticos, su manejo se da con la misma facilidad que los demás tipos de datos.

En cuanto a los **operadores** relacionales y aritméticos, se permiten los mismos operadores que en C++, con la salvedad de >>> y la utilización del operador + para concatenar cadenas de caracteres.

Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución, lo que hace que se detecten errores lo antes posible. Por ello facilita la **comprobación de la compatibilidad** de conversión de diferentes tipos de datos al momento de ejecutar un programa. En caso de error utiliza procesos de control por excepciones, que pueden hacer comprobación en tiempo de ejecución de la compatibilidad de tipos y emitir una excepción cuando falla. En el caso de C y C++ tiene mecanismos que permiten cambiar, por ejemplo, el tipo de un puntero. Sin embargo, se requiere precaución al hacerlo, ya que si esto sucede es difícil su detección al momento de la ejecución del programa.

Otras diferencias a nivel de lenguaje Java, son:

- No considera los tipos **struct**, **union** ni las directivas **typedef** ni **#define**;
- no permite una sobrecarga de operadores;
- hace uso de las sentencias **break** y **continue** que sustituyen de alguna forma el uso de la sentencia **goto** utilizada en C++ para transferir el control de la ejecución de un programa que carezca de estructura en su código;
- no admite el operador de asignación dentro de la condición del `if`; solo el operador de comparación;
- no soporta herencia múltiple;
- maneja de manera natural los tipos de datos enumerados;
- maneja argumentos en la línea de en de forma diversa a como lo hacen C o C++;
- utiliza una clase **String** que es parte del paquete **java.lang** y se diferencia de la matriz de caracteres terminada con un nulo que usan C y C++;
- Java gestiona en forma automática el uso de memoria, con lo que no se hace necesario utilizar las funciones previstas para este fin en C y C++;
- utiliza primitivas similares a las de C++, pero más elaboradas, permitiendo una mejor gestión de archivos, sockets, teclado y monitor, de tal forma que se pueden utilizar dichas primitivas para cualquier operación de Entrada/Salida.

Java tiene potencialidades con las que se pueden generar y crear diferentes tipos de aplicaciones, algunas de ellas son:

- Las que trabajan sin necesidad de utilizar algún navegador para internet.
- Pequeñas aplicaciones applet que es posible descargar de internet y luego ser visualizadas.
- Componentes de Java, que se incorporaran en forma gráfica a otros componentes JavaBeans.
- Aplicaciones del lenguaje Java que se codifican sobre cualquier documento HTML JavaScript.
- Módulos Servlets que permiten sustituir o utilizar el lenguaje Java en el lugar.

En resumen, existen muchas coincidencias entre C++ y Java a nivel conceptual y sintáctico. Pero también hay aspectos distintos muy significativos: se pueden esperar mejores posibilidades a bajo nivel y una mayor

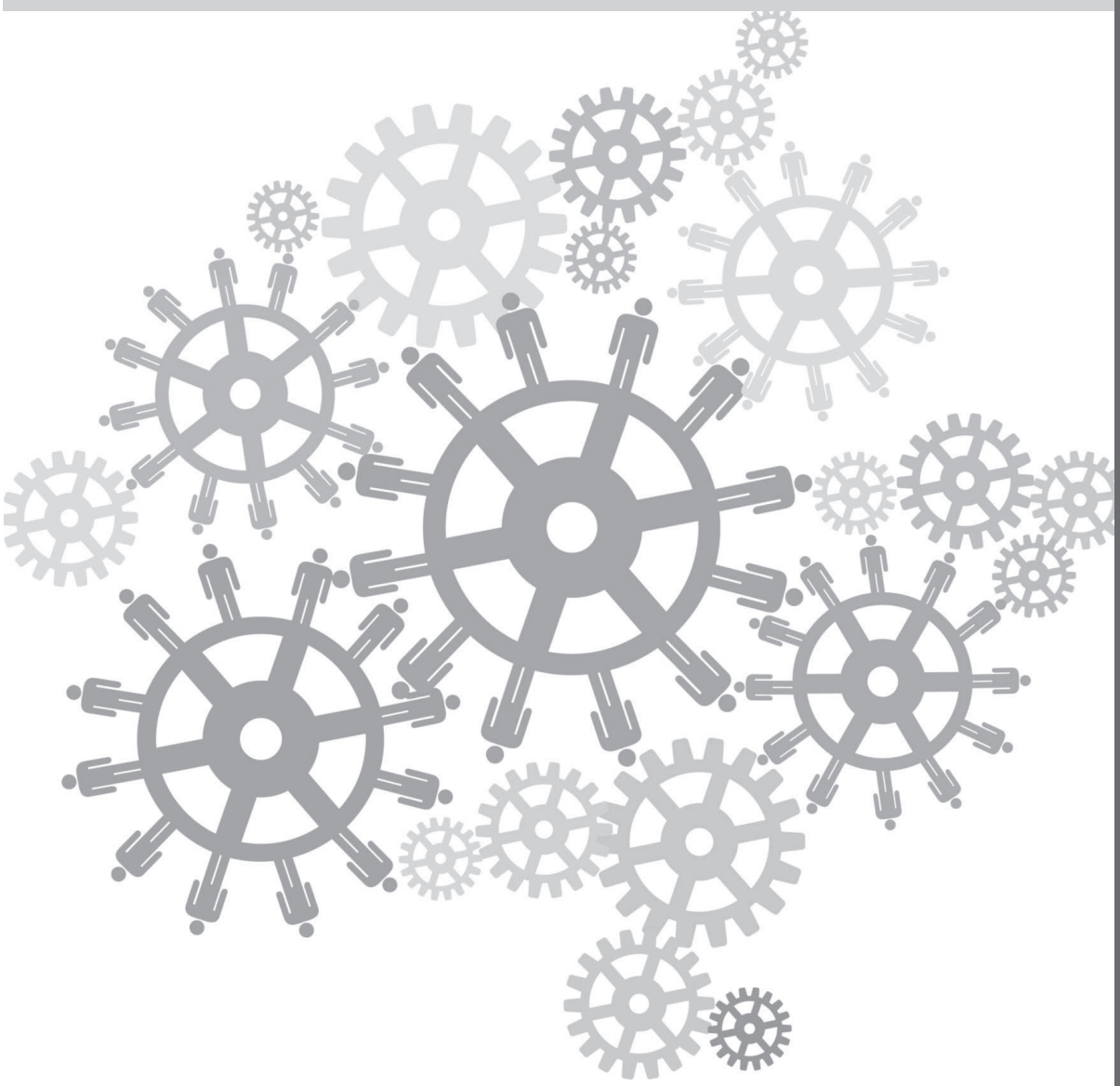
[7] Varios autores. *Programación en Java 2*. McGraw-Hill (Schaum), España, 2005, p. 12.

eficiencia por parte de C++, mientras el campo de Java es más general y conlleva una mayor portabilidad. De manera obvia, ambos comparten el paradigma de la programación orientada a objetos.

¿Cuál es el mejor lenguaje para enseñar POO, C++ o Java? Aunque nosotros nos inclinamos por Java, debemos admitir que en ese sentido existen fuertes argumentos para defender a ambos (e incluso a favor de Python o .NET, cuyo análisis queda fuera de esta obra). Es difícil que pueda darse un consenso entre los docentes. La única recomendación que podríamos dar de manera tajante es que resulta indispensable que los estudiantes perciban que el paradigma de la programación orientada a objetos aprovecha los elementos de la programación estructurada y le da mayor potencialidad. Se basa en la construcción de componentes llamados clases, que son en realidad tipos de datos definidos por el desarrollador, los cuales deben ser probados de manera unitaria. Tratar a C++ o a Java para crear únicamente programas sencillos de entrada-proceso-salida como los primeros que vimos en este libro, distorsiona en gran medida al mundo de la POO e impide ver el enorme potencial de este paradigma.

Capítulo 5

Uso de múltiples clases de programación orientada a objetos



5.1 Introducción al lenguaje Java

Trataremos de dar una breve introducción al lenguaje de programación Java, antes de proseguir con aspectos más avanzados de la orientación a objetos. Por el momento dejaremos de lado trabajar con clases. El criterio es sencillo, aunque pudiera resultar polémico: hemos notado que a los estudiantes se les facilita más trabajar con clases si tienen previamente alguna familiaridad con la sintaxis de Java, por ello reproducir los ejemplos y hacerles pequeñas adaptaciones ahorrará contratiempos posteriores.

También recomendamos su lectura para quienes trabajen con un lenguaje C++, pues les servirá para tener una visión más amplia y repasar mentalmente la forma en que se dan estos conceptos en el propio C++.

Como ya se mencionó, Java es un lenguaje orientado a objetos, con orígenes en C++, al que se le hicieron mejoras significativas. Empecemos por entender algunas definiciones básicas del lenguaje de programación Java.

El concepto de token en C, C++ y Java

Un **token** es el elemento más pequeño y significativo para el compilador. En el proceso de compilación de programas el texto es reconocido, analizado y se eliminan los espacios en blanco y los comentarios, teniendo al final tokens individualizados que se traducirán a código byte Java si se cumplen las reglas léxicas y sintácticas del lenguaje. Los **token** se establecen bajo cinco categorías: identificadores, palabras clave, constantes, operadores y separadores.

El objetivo de los identificadores es asignar nombres en forma única y fácil manejo para el programador a variables, métodos y clases. Los identificadores son sensibles al manejo de mayúsculas y minúsculas, inician con una letra, un subrayado o guión bajo (_) o símbolo de pesos ($); pueden incluir las cifras del 0 al 9. No se pueden usar palabras clave del lenguaje de programación como por ejemplo nombres de identificadores.

Se conocen como literales o constantes las declaraciones que definen un valor, el cual no cambia durante la ejecución de un proceso.

Es una buena práctica estándar de Java emplear la "notación de camello"; es decir, que la letra inicial de las palabras sea con mayúscula, exceptuando la primera palabra. Si se trata de clases y constantes se comenzará el identificador con mayúscula; para métodos y variables se iniciará con minúscula; por último, los paquetes utilizarán solo minúsculas.

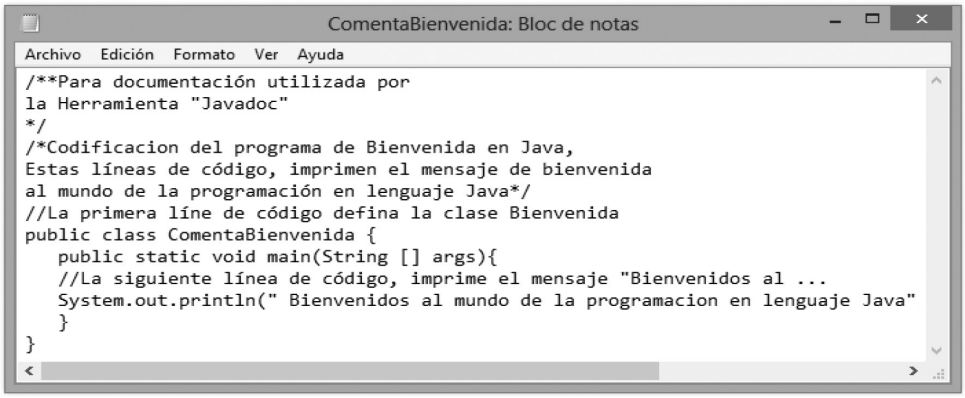
- Las clases: Clase o UnaClase.
- Las interfases: Interfaz o UnaInterfaz.
- Los métodos: metodo() o metodoLargo().
- Las variables: calif o MiCalif.
- Las constantes: Constante o Constante_Larga.
- Los paquetes: java.paquete.subpaquete.

En C todo se maneja con minúsculas, a excepción de las constantes que se trabajan por completo con mayúsculas. Por último, está el caso de C++, que heredó el manejo de C; no obstante, algunos autores sugieren que se aproveche la "notación de camello" para una mayor claridad, aunque no existe consenso en este sentido.

Comentarios en C, C++ y Java

Una buena práctica al escribir código en algún lenguaje de programación es documentarlo y describir qué acción se pretende en cada sección o segmento de código. Sin embargo, es una práctica que muchos deja-

mos de lado. Utilizaremos el programa 5.1 para ejemplificar las formas que se utilizan en Java para hacer comentarios al momento de escribir el código.



Programa 5.1 Formas de manejar comentarios en Java.

La forma de comentar /* */ proviene de C, C++ incorpora // y a su vez Java agrega el formato /** */. Cuando aparece un texto después de // no se considera el comentario hasta el fin de la línea. Como podemos apreciar, el texto o comentario que se encuentra entre /***/se utiliza para que la herramienta **Javadoc** genere documentación de manera automática (tiene dos asteriscos al inicio del texto o comentario). El compilador ignora el texto que se encuentre entre /* */, por lo que solo se utiliza para efectos de comentario sin que se tenga que ejecutar alguna acción. Tanto las forma de comentar /** */ como /* */ pueden utilizar una o más líneas.

En C++ y Java, se suele usar el comentario de doble diagonal para indicar la finalidad de algunas variables. Por ejemplo:

```
int edadMinima = 65; // edad mínima para jubilación
```

EJERCICIO SUGERIDO

Profundizar en el tema de la generación automática de documentación a través de comentarios y probar que funciona en forma adecuada.

Palabras reservadas de Java

Los identificadores en Java se representan por una secuencia de caracteres que inician normalmente con una letra (a-z, A-Z) y pueden contener números (0-9), así como los caracteres $ y _ (subrayado). No hay un límite definido respecto al número de caracteres que puede contener un identificador.

La siguiente línea de código muestra dos ejemplos de declaración de una variable entera (int) con su correspondiente identificador:

```
int velocidad; o int maxVelocidad;
```

Las palabras reservadas de Java son, en conjunto, más amplias que las de C o C++ y, al igual que en todos los lenguajes de programación, su utilización es limitada y específica; por ello, no pueden utilizarse como identificadores.

El cuadro 5.1 muestra las palabras reservadas en Java.

Cuadro 5.1 Palabras reservadas en Java				
abstact	boolean	break	byte	case
catch	char	class	continue	default
Do	double	else	extends	false
final	finally	float	For	if
implements	import	instanceof	Int	interface
long	native	new	null	package
private	protected	public	return	short
static	super	switch	syncroniced	this
throw	throws	transient	true	try
Void	volatile	while		
byvalue	cast	future	const	generic
goto	inner	operator	outer	rest
var				

Tipos de datos y variables en Java

En Java, igual que en C y C++, todas las variables deben declararse antes de usarse. Se declara el tipo, el nombre de la variable y de manera opcional su valor de inicio. Los tipos de datos primitivos son predefinidos por el lenguaje y se hace referencia a ellos con una palabra reservada. Los tipos de datos **primitivos** soportados por Java son:

Enteros (byte, short, int, long). Se utilizan para representar números enteros con signo. Puede tratarse de uno de los siguientes casos: **byte**: de un byte, con un valor mínimo de —128 y un valor máximo de 127; **short**: de 2 bytes, con un valor mínimo de —32,768 y un valor máximo de 32,767; **int**: de 4 bytes, con un valor mínimo de —2,147,483,648 y un valor máximo de 2,147,483,647; **long**: de 8 bytes, con un valor mínimo de —9,223,372,036,854,775,808 y un valor máximo de 9,223,372,036,854,775,807. El tipo por omisión es **int.** Algunos ejemplos de declaración:

```
int miEdad = 55; // edad del autor
long miPeso = 88L; // mi peso en gramos, observe la L
```

Decimales (float y double). Se usan para representar números con cantidades fraccionarias. Puede tratarse de **float**: número de punto flotante de 4 bytes que representan números decimales con cantidades fraccionarias, ya sea en notación estándar (563,84) o científica (5.63784e2); **double**: número de punto flotante de 8 bytes (64 bits). El tipo por omisión es double. Se muestran algunos casos con variables decimales.

```
float velocidad = (float) 60.10
double valPi = 314.16e-2 ;
```

Lógicos (boolean). Tipo de datos que contiene solo dos posibles valores **true** (verdadero) o **false** (falso). Por ejemplo:

```
boolean acreditado = false; // pasó o no el semestre
```

Carácter (char). Tiene un valor mínimo de'\u0000'(o 0) y un valor máximo de'\uffff'(o 65,535 inclusive), pues se basa en la convención Unicode, a 4 bytes. También sirve para enviar caracteres especiales en el despliegue (véase cuadro 5.2)

Cuadro 5.2 Tabla de tipos de datos carácter

Descripción	Representación	Valor Unicode
carácter Unicode	\udddd	
número octal	\ddd	
barra invertida	\\	\u005C
Continuación	\	\
Retroceso	\b	\u0008
retorno de carro	\r	\u000D
alimentación de formularios	\f	\u000C
tabulación horizontal	\t	\u0009
línea nueva	\n	\u000A
comillas simples	\'	\u0027
comillas dobles	\"	\u0022
números arábigos ASCII	0-9	\u0030 a \u0039
alfabeto ASCII en mayúsculas	A.-Z	\u0041 a \u005A
alfabeto ASCII en minúsculas	a.-z	\u0061 a \u007A

Cadena (String). Representada por una combinación de caracteres que se acota con comillas dobles. La clase `String` cuenta con métodos que facilitan combinar y modificar cadenas. Por ejemplo:

```
String c = "Cadena de caracteres";
```

Cabe aclarar que el tipo de datos String en realidad no es un tipo de datos primitivos, sino una clase que permite la creación de objetos para crear y manipular cadenas de caracteres (tema que se explicará después con más detalle). No obstante, es conveniente que los estudiantes estén familiarizados con la clase String desde un inicio.

EJERCICIO SUGERIDO

De seguro habrá observado que cuando se indica un número que no coincide con las opciones por omisión es necesario poner una letra adelante del número para que sea consistente con el tipo de dato especificado y el compilador no marque un error de sintaxis (por ejemplo: `long miPeso = 88L`). Realice un programa sencillo que trabaje con todos y cada uno de estos de tipo numérico para verificar que puede declararlos en forma adecuada.

El ámbito de las variables en C++ y Java

Las variables en C++ y Java pueden tener diferentes ámbitos: de instancia, de clase, locales y de ciclo. A diferencia de otros lenguajes, Java no tiene variables globales; es decir, variables que son vistas en cualquier parte del programa. Desde el punto de vista conceptual, la programación orientada a objetos no maneja variables globales, pero todos los compiladores de C++ también compilan programas en C, por lo que abren esta posibilidad. Por ello, en términos prácticos, puede decirse que C++ sí puede manejar variables globales, aunque debieran usarse lo menos posible.

- **Variables de instancia.** Se utilizan para definir los atributos de un objeto.
- **Variables de clase.** Son similares a las variables de instancia, con la excepción de que sus valores son los mismos para todas las instancias de la clase.
- **Variables locales.** Se declaran y se utilizan dentro de las definiciones de los métodos.
- **Variables de ciclo.** Se declaran para un ciclo específico y solo tienen vida dentro de este. Por ejemplo:

```
for (int j = 0; j < n; j++) {
   // bloque de instrucciones
 }
```

Operadores y expresiones en Java

Las expresiones son una combinación de variables, operadores y llamadas a métodos que se construyen de acuerdo con la sintaxis del lenguaje y que devuelven un valor. El tipo de dato del valor regresado por una expresión depende de los elementos que se usen en ella.

Una expresión es una combinación de variables, operadores y llamadas a métodos, representada según la sintaxis del lenguaje, y como resultado devuelve un valor; es decir, realiza el cálculo indicado por los elementos de la expresión y devuelve el valor obtenido como resultado del cálculo.

Si tomamos como referencia el signo =, este no significa estrictamente una igualdad, representa una asignación a una variable de una expresión, aunque conviene aclarar que el resultado también puede devolverse en forma directa desde una rutina, o combinarse con un despliegue en pantalla. Si agrupamos las posibilidades de operadores, entonces podemos crear instrucciones. Por ejemplo:

```
nombreVariable = expresión;
residuo = dividendo % divisor;
```

Los operadores son símbolos especiales utilizados en expresiones que indican una evaluación que se realizará en objetos, datos y constantes. Se dividen en cuatro categorías: a) aritméticos, b) de comparación y condicionales, c) a nivel de bits y lógicos, d) de asignación. El cuadro 5.3 resume los operadores admitidos por Java.

Cuadro 5.3 Operadores de Java

Operador	Significado	Operador	Notas
+	Suma	., [], ()	corchetes utilizados en arreglos
–	Resta	++, --, !, ~	! es el NOT lógico y ~ es el complemento de bits
*	Multiplicación	new (tipo)expr	para crear instancias de clases
/	División	*, /, %	multiplicativos
%	Módulo	+ -	aditivos
=	asignación	<<, >>, >>>	corrimiento de bits
+=	suma y asignación	<>, <=, >=	relacionales
–=	resta y asignación	==, !=	igualdad
*=	multiplicación y asignación	&	AND (entre bits)
/=	división y asignación	^	OR exclusivo (entre bits)
%=	módulo y asignación	|	OR inclusivo (entre bits)

==	Igualdad	&&	AND lógico
!=	Distinto	||	OR lógico
<	menor que	?, :	condicional
>	mayor que	=, += ,-=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=	asignación
<=	menor o igual que		
>=	mayor o igual que		
++	incremento		

Operadores aritméticos. Los números enteros y decimales incluyen la suma (+), resta (–), multiplicación (*), división (/) y módulo (%); es decir, el residuo de una división entre enteros.

Operadores de comparación y condicionales. Los operadores de comparación trabajan dos valores y determinan su relación. Por ejemplo, el operador != devuelve verdadero **(true)** si los operandos son distintos. Estos operadores de comparación en general se usan junto con los operadores condicionales para construir expresiones un poco más complejas para la toma de decisiones. El operador **&&** ejecuta una operación lógica **and**. Por ejemplo, se pueden utilizar dos operaciones diferentes de comparación con **&&** para determinar si ambas relaciones son ciertas. Por ejemplo:

```
if ((80 < velocidad) && ( velocidad < velocMax ))
```

Operadores de bit. Los operadores de bit permiten realizar operaciones de bit sobre los datos. Hay dos tipos: de desplazamientos de bits y operadores de lógica de bit. Los de desplazamiento recorren los bits del operando de la izquierda las veces indicadas por el operando de la parte derecha. La dirección del desplazamiento es la indicada por el operador. Los operadores de lógica de bits (lógica de Boole) se utilizan para modelar condiciones cierto/falso. El operador de complemento invierte el valor de cada bit del operando, convierte el falso en cierto y el cierto en falso. La manipulación de bit es útil, entre otras posibilidades, para gestionar indicadores booleanos (banderas).

Operadores de asignación. El operador de asignación = se utiliza para asignar un valor a otro (por ejemplo: **int suma Calificacion = 0;**). Además existen operadores de asignación que realizan un atajo en la escritura de código y aplican en operaciones aritméticas, lógicas, de bit y de asignación con un único operador.

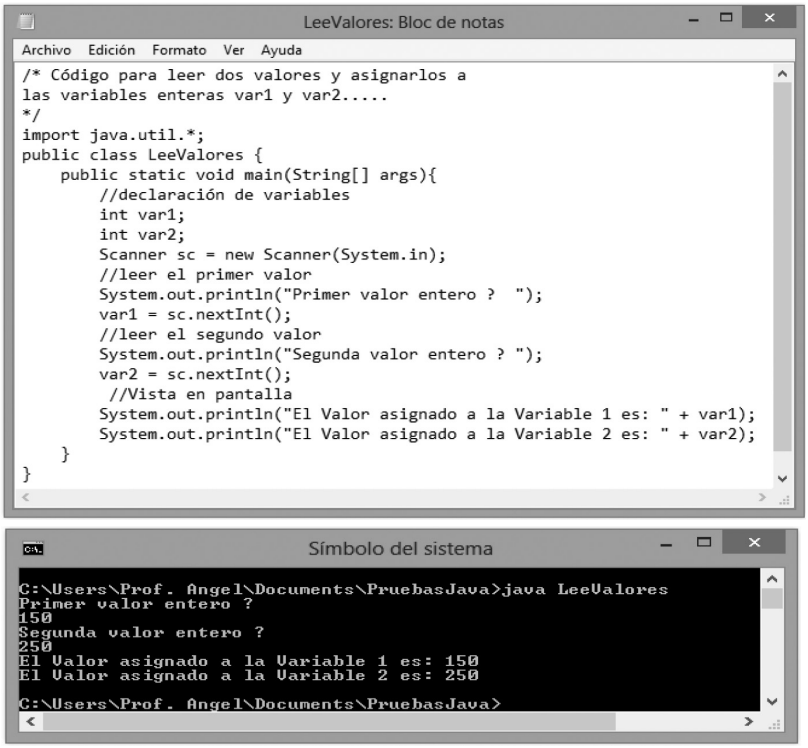
Precedencia de operadores. Se le llama precedencia al orden que aplica el compilador a los operadores al momento de su ejecución. C, C++ y Java obedecen a las mismas reglas generales de precedencia: los operadores de mayor precedencia son evaluados antes que los operadores con una precedencia menor. Si en una sentencia aparecen operadores con la misma precedencia, los operadores de asignación son evaluados de derecha a izquierda y si existen operadores binarios (menos los de asignación) serán evaluados de izquierda a derecha, el uso de paréntesis también puede definir el orden en que se desee evaluar una expresión. Por ejemplo: si tenemos la siguiente expresión:

```
x = var1 + var2 * var3;
```

entonces el compilador decide en función de la precedencia asignada a los operadores; es decir, el operador de multiplicación tiene mayor precedencia que el operador de suma, por ello se evaluará primero `var2 * var3`. Si deseáramos que la suma se llevara a cabo, primero sería necesario utilizar paréntesis y la expresión sería

```
x = (var1 + var2) * var3.
```

A manera de ejemplo, véase el programa 5.2, que lee dos variables y expresa su suma.[1] Claro que es un programa alejado de la orientación a objetos, pero nos permitirá observar semejanzas y diferencias entre las sintaxis de C y Java.



Programa 5.2 Un programa sencillo en Java: la suma de dos números.

EJERCICIO SUGERIDO

Confirme que el programa trabaje en forma adecuada. Después, realice un programa que reciba la base y la altura de un triángulo y despliegue su área sin tener ningún programa de Java a la vista (podrá buscar ayuda o asesoría hasta después de intentar solucionar los probables errores por sí mismo).

Estructuras de control en C, C++ y Java

Las sintaxis de C y Java son muy similares. En muchas ocasiones se requiere que una determinada acción sea en forma secuencial, trabaje con base en una condición o se repita un número determinado de veces. En tal situación se pueden utilizar las estructuras de control, que se dividen en tres categorías: secuencial, condicional o selectiva, e iterativa o repetitiva. El orden natural de ejecución de las expresiones por defecto es el secuencial; es decir, una después de otra de acuerdo con el orden en que aparecen a lo largo del programa. Según la sintaxis del lenguaje, las expresiones son separadas por el carácter (;), y se agrupan en **bloques** delimitados por llaves (`{ }`). Las llaves pueden omitirse si el bloque es de una única instrucción.

[1] Tenemos que hacer hincapié en que la lectura de datos en estos momentos da una mayor interactividad para los estudiantes, pero no se está aplicando con todo su potencial, pues falta robustez (supone que el usuario dará un número entero y si no es así el programa se interrumpe). El tema de la robustez se abordará en el apartado de Excepciones.

El código fuente en Java se divide en diferentes partes que pueden estar separadas entre ellas por llaves; por lo regular se les denomina bloques. Los bloques son independientes y pueden contener otros bloques, de manera que se establecen niveles o jerarquías que permiten su anidamiento. La agrupación de sentencias de código con el uso de llaves da un sentido lógico al momento de la compilación, ya que facilita la identificación de inicio y fin de cada bloque de código, con lo que proporciona un fácil entendimiento y seguimiento del programa. Se puede obtener un mejor control e identificación de los diferentes bloques mediante la utilización de sangrías y alineación del código. Por ejemplo:

```
// programa con sangrías y alineación de código
{
   // bloque Principal
   …
   {
   // bloque
   …
   }
}
```

Si durante la ejecución de un programa tenemos acciones que se repiten un número determinado de veces, estas se pueden estructurar de distintas formas. Por ejemplo, leer dos calificaciones en Java podría hacerse con las instrucciones:

```
var1 = sc.nextInt();
var2 = sc.nextInt();
```

¿Que pasaría en el caso de que se requiriera leer más calificaciones? Es aquí donde se utilizan las estructuras de control, que permiten realizar ciertas acciones, en tanto se cumpla o no una condición. Si se quiere modificar el orden en que se ejecutan determinadas instrucciones dentro de un programa se pueden utilizar las estructuras condicionales y repetitivas. Las estructuras de control soportadas por el lenguaje Java se sintetizan en el cuadro 5.4.

Cuadro 5.4 Palabras que manejan estructuras de control en C, C++ y Java

Sentencia	Clave
Decisión	if-else, switch-case
Bucle	for, while, do-while
Misceláneo	break, continue, label:, return, goto

La estructura de decisión condicional indica la ejecución de un grupo de instrucciones o de otro alterno, según se indique en el programa. La instrucción `if` puede ser una estructura condicional simple `if` o una condicional doble `if …else` para ser evaluada en forma correcta. La condición deberá ser una expresión booleana; es decir, debe tener como resultado de su evaluación el valor true o false. En el caso de una condición simple, se evalúa la condición y si esta se cumple, entonces ejecuta una determinada acción. En caso contrario se ignora.

Es necesario aclarar que Java no soporta la sentencia `goto`, por lo que en su lugar se pueden utilizar otro tipo de sentencias.

La sentencia `if-else` permite realizar la ejecución de diferentes conjuntos de sentencias según el criterio seleccionado. Al evaluarse la condición, si esta es verdadera, entonces se realiza un grupo de instrucciones; de lo contrario ejecutará otro bloque de instrucciones.

```
if (expresión booleana){
   instrucciones 1 // si la condición es verdadera
}
else {
   instrucciones 2 // si la condición es falsa
}
```

Si agregamos un par de líneas más a nuestro código del ejemplo anterior, entonces tendremos algo similar a la estructura antes descrita.

```
Programa 5.3 Ejemplo de una instrucción if-else.
// Código en Java que pregunta el promedio de calificaciones
// y emite un mensaje si se cumple la condición.
import java.util.*;
public class Condiciones {
   public static void main( String[] args ){
      int miPromedio;
      Scanner sc = new Scanner( System.in );
      System.out.print("¿Cuál es tu promedio? ");
      miPromedio = sc.nextInt();
      if (miPromedio >= 6 ) //Si la condición es verdadera
         System.out.println("Curso acreditado ....");
      else //si la condición no se cumple o es falsa
      System.out.println("Curso no acreditado ... ¡ recursar !");
   }
}
```

Es muy probable que se quieran ejecutar diferentes grupos de instrucciones dependiendo de ciertas condiciones, lo cual puede lograrse mediante la utilización de expresiones `if-else` anidadas. A esto se le denomina sentencias `else if`. Si se sigue con el ejemplo del promedio de calificaciones, si se quiere asignar un determinado peso, según el promedio obtenido, entonces este pudiera ser una buena aproximación al código.

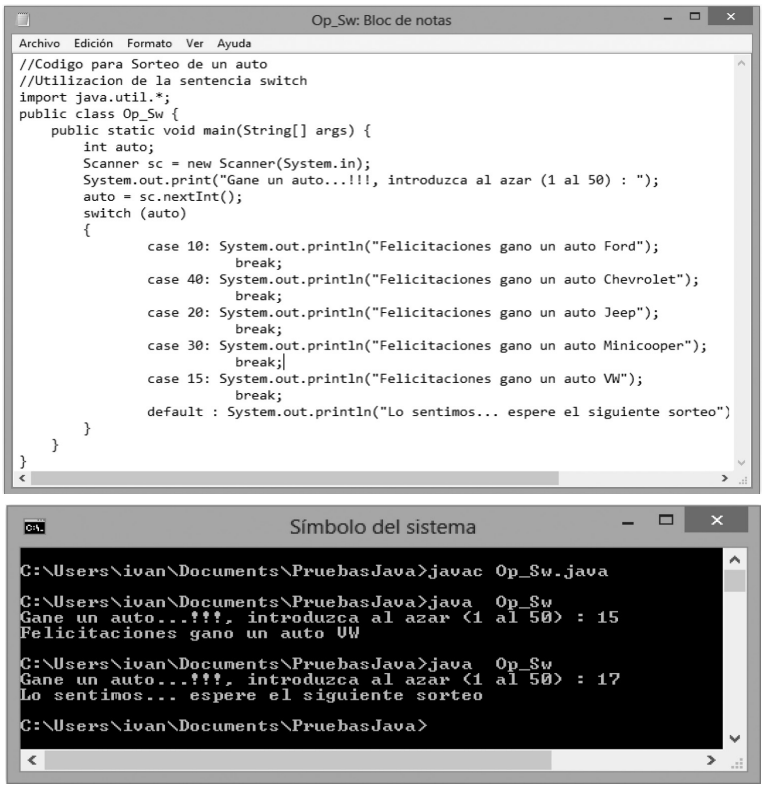
```
int miPromedio;
char peso;
if (miPromedio > 90)
   {peso = '1';}
else
   if (miPromedio > 80)
      {peso = '2';}
   else
      if (miPromedio > 70)
         {peso = '3';}
      else
         if (miPromedio > 60)
            {peso = '4';}
         else
              {peso='0';}
```

Esta forma de programar sentencias `else if` es poco recomendable y en ocasiones puede generar confusiones. Sin embargo, C, C++ y Java tienen la sentencia `switch` que facilita estas acciones de condición; es recomendable utilizar `switch` para sentencias con más de tres o cuatro posibilidades. La sentencia `switch` hace posible seleccionar entre varias alternativas un determinado conjunto de sentencias según el valor de alguna expresión, en general de tipo entero. Se evalúa la condición y, dependiendo del valor obtenido, se ejecuta el `case` que coincida con el valor de la expresión. Se ejecutan las sentencias que

siguen al `case` seleccionado hasta encontrar un `break` o hasta el final del `switch`; el `break` produce un salto a la siguiente sentencia a continuación del `switch`. Si ninguno de estos casos se cumple se ejecuta el bloque `default`, el cual es opcional. Por ejemplo:

```
switch ( expresión_entera ) {
   case val_1 :
      sentencias;
      break;
   case val_2 :
      sentencias;
      break;
   case val_3:
      sentencias;
      break;
   default:
      sentencias;
}
```

Para ejemplificar el `switch`, supongamos que queremos realizar un programa en el cual se capture un número del teclado entre 1 y 50 y si coincide con valores asignados con anterioridad, se gana como premio un automóvil de determinada marca. En este caso, se toma como base la variable **auto** para conocer, en su caso, el nombre de la marca del ganador. El programa 5.4 muestra el código Java para este ejemplo.



Programa 5.4 Ejemplo de uso de `switch`.

C, C++ y Java cuentan con otro tipo de sentencias que permiten ejecutar una serie de instrucciones en forma repetida mientras se cumplan ciertas condiciones: `while`, `do – while` o `for`. Mientras la expresión Booleana tenga un valor verdadero se ejecutará el bloque de instrucciones definido. En el `while` la condición se evalúa desde el inicio del ciclo o bucle, con lo cual se facilita que el grupo de instrucciones se ejecute las veces que sean consideradas en la condición. Su forma general es como sigue:

```
while ( expresiónBooleana ) {
   bloque de instrucciones;
};
```

Por ejemplo, en este bucle se itera hasta que el valor de la variable **cuenta** es mayor a 55.

```
int cuenta = 5;
while ( cuenta <= 55 ) {
   cuenta = cuenta + 5;
}
```

Al evaluar la ejecución del ciclo `while`, primero se evalúa la condición y si el resultado es verdadero se ejecuta el bloque de instrucciones definido, regresando el control al inicio del ciclo; si el resultado es falso, el bloque de instrucciones no se ejecuta y el programa seguirá con otra serie de instrucciones según esté definido en el programa.

En las sentencias del ciclo `do-while`, el proceso que sigue es similar al ciclo `while`, con la diferencia de que en este la expresión se evalúa al principio del ciclo, y la del ciclo `do-while` se evalúa al final. La forma general del ciclo `do-while` es la siguiente:

```
do {
   bloque de instrucciones;
} while ( expresiónBooleana );
```

Este tipo de sentencias `do-while` se utiliza poco. Sin embargo, tiene ciertas ventajas cuando se trata de programar, como es el caso en que al menos un bloque de instrucciones tenga que ejecutarse una vez.

El ciclo `for` es de uso amplio y aplica por lo general cuando se conoce de antemano el número de veces que se realizará un bloque de instrucciones. En la sentencia `for` se resume un ciclo `do-while` con una inicialización previa. La forma general de la sentencia `for` es:

```
for (inicialización; terminación ; incremento)
 …instrucciones;
```

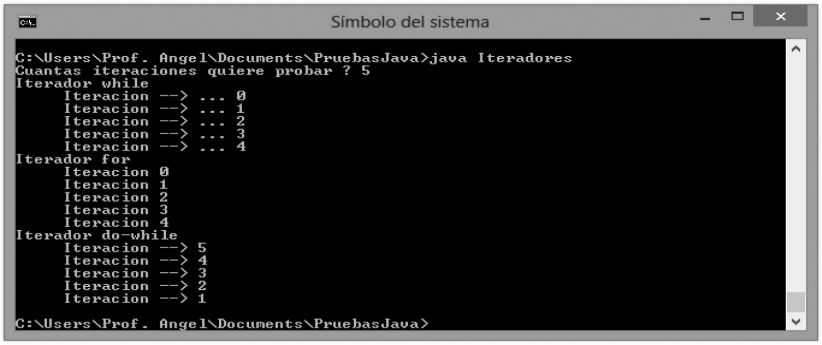
La inicialización es una sentencia que se ejecuta una vez antes de entrar en el ciclo. La terminación determina cuándo debe concluir la ejecución de instrucciones del ciclo y se evalúa al final de cada iteración del ciclo. Cuando la expresión se evalúa como falsa, el ciclo termina. El incremento se realiza en cada iteración; suele utilizarse para incrementar una variable contador:

```
for (j = 0 ; j < 10 ; j++)
```

En general se utilizan los ciclos `for` cuando se conocen sus restricciones: inicialización, terminación e incremento. Uno de los casos en los cuales se emplea con mucha frecuencia es en el manejo de matrices. El programa 5.5 muestra la posible utilización de sentencias para el manejo de ciclos.

```
//Programa 5.5 Ejemplo de ciclos con while, do-while y for.

// Código para ejemplificar el uso de sentencias de iteración
import java.util.*;
public class Iteradores {
   public static void main (String[] argumentos) {
      int inicial = 0;
      int numiter = 0;
      Scanner sc = new Scanner (System.in);
      System.out.print("¿Cuántas iteraciones quiere probar?");
      numiter = sc.nextInt();
      salida("Iterador while");
      while (inicial < numiter) {
         salida(" Iteracion --> ..." + inicial);
         inicial++;
      }
      salida ("Iterador for");
      for (int j = 0; j < numiter; ++j) {
         salida(" Iteracion --> ..." + j);
      }
      salida("Iterador do-while");
      do {
         salida(" Iteracion --> ..." + inicial--);
      } while (inicial > 0);
      System.exit(0);
   }
   public static void salida (String cadena) {
      System.out.println(cadena);
 }
}
```



De la misma forma en que se puede desear romper la iteración en un ciclo con la sentencia `break`, también es posible desear continuar con el ciclo, pero después de dejar pasar una determinada iteración. En este caso se utiliza `continue`. La diferencia entre ambos radica en que la sentencia `break` detiene la ejecución del ciclo y ejecuta la primera línea del programa tras el ciclo, mientras que la sentencia `continue` detiene la iteración actual y pasa a la siguiente iteración del ciclo sin salir de este. Por ejemplo:

```
for (int i = 0; j < 5; j++) {
   System.out.println("Acción dentro del ciclo");
   continue;
   System.out.println("No se ejecuta acción alguna");
}
```

Sentencia `return`

La sentencia `return` se puede usar para salir del método en curso y retornar a la sentencia dentro de la cual se realizó la llamada. Tan solo se debe poner el valor a continuación de la palabra `return`. El valor devuelto por `return` debe coincidir con el tipo declarado como valor de retorno del método. No obstante, puede utilizarse `return` para métodos que no devuelven valores (declarados como `void`); esto evita la ejecución de todo el código de la subrutina. A continuación se muestra una sección de código totalmente independiente para ejemplificar lo expuesto:

```
int aumentado(){
   int cuenta;
   // aquí va el código de la subrutina aumentado
   return ++cuenta;
}

void metodoReturn(){
   boolean pregunta;
   // código de la subrutina metodoReturn
   if ( pregunta == true )
   /* con el return se para la subrutina y se
      devuelve el control al programa principal */
   return;
}
```

Arreglos en Java

Un arreglo es una construcción que proporciona almacenaje a una lista de elementos de un mismo tipo, sean simples o compuestos. Los arreglos de una dimensión se conocen como vectores.[2] Por ejemplo, un arreglo puede declararse de cualquiera de las dos formas siguientes (la primera es la que recomienda Sun, la empresa que creó Java):

```
int[ ] listaElem; // vector de lista de elementos
int listaElem[]; // vector de lista de elementos
```

En Java no se indica la dimensión de un arreglo cuando se declara. En lugar de ello, el tamaño se determina en forma explícita en el programa cuando se crea en memoria con el operador new. Entre corchetes se indica el tamaño del vector y puede utilizarse `length` para obtener el número de columnas que tiene cada fila. El tipo de datos deberá coincidir con el tipo declarado en el **vector**, que a su vez deberá ser una variable declarada como `tipo [ ]`. Al utilizar la forma new se asignará el valor 0 a cada elemento del vector. Valgan algunos ejemplos para darnos a entender mejor.

```
listaElem = new int[10];
```

También se pueden asignar elementos que conforman la lista de otras formas:

```
listaElem = new int[variableConocida];
int listaElem [ ] = {3,5,7};
```

En el caso de una matriz bidimensional:

```
int [ ] [ ] = new tipo [filas] [columnas]
int matriz [ ] [ ] = {1, 3, 5}, {2, 4, 6}
```

[2] Recalcamos que la expresión de vectores en programación se refiere a arreglos unidimensionales y no tiene que ver con los vectores aplicados al campo de la física, cuyo concepto es formal y más amplio.

Las sintaxis anteriores también aplican en el caso de cadenas o cualquier otro objeto. Por ejemplo, puede crearse un arreglo de cadenas paso a paso o en forma corta. Cualquiera de los dos bloques siguientes es equivalente.

```
String[] nombre;
nombre = new String[3];
nombre[0] = "Adán";
nombre[1] = "Eva";
nombre[2] = "Otro";
String[] nombre = {"Adán", "Eva", "Otro" };
```

La forma corta también se puede aplicar a objetos. Por ejemplo:

```
Fecha[] celebraciones = {
   new Fecha(1,1,2013),
   new Fecha(10,5,2013),
   new Fecha(25,12,2013)
   };
```

Este manejo permite varias posibilidades que no estaban presentes en el lenguaje C tradicional:

a) El tamaño del arreglo se puede crear en tiempo de ejecución.

```
int numeroDeElementos;
  …
  /* en esta parte del código se calcula el valor de x */
  …
char[] arreglo;
arreglo = new char [x];
```

b) Pueden crearse arreglos bidimensionales tradicionales, pero también arreglos de arreglos, cada uno con una longitud diferente.

```
int[][] dosDimensiones = new int [2][];
dosDimensiones[0] = new int[5];
dosDimensiones[1] = new int[4]
```

De hecho, si todos fueran de la misma longitud se recomiendaría asignarlas en un solo paso.

```
int[][] dosDimensiones = new int [2][5];
```

c) Si se vuelve a crear el arreglo se pierde el contenido anterior y se regresa a un arreglo vacío. El arreglo anterior será destruido luego por el recolector de basura.
d) Por supuesto, también puede haber arreglos de objetos.
e) La palabra `lenght` sirve para averiguar el tamaño del arreglo.
Por último, cabe citar que la función `arraycopy` permite copiar el contenido de un arreglo a otro. Su sintaxis es:

```
System.arraycopy(arreglo1, posicion1,
arreglo2, posicion2, elementosACopiar);
```

A manera de ejemplo, el programa 5.6 muestra la creación y despliegue de un arreglo unidimensional.

```
//Programa 5.6 Creación y despliegue de un arreglo unidimensional.
//Código para listar los elementos de un vector
import java.util.*;
public class LeeElem {
   public static void main(String[] args){
      int[] listElem = {100,50,25,50,100};
      for(int j=0; j<listElem.length; j++)
         System.out.println(listElem[j]);
   }
}
```

Con esta pequeña introducción a los conceptos básicos del lenguaje de programación Java, así como de su sintaxis, podemos estar listos para abordar las potencialidades con que cuenta.

EJERCICIOS SUGERIDOS

Elabore en Java alguno de los siguientes programas, que ya se habían sugerido para lenguaje C. Nos servirán para trabajar con condicionales, ciclos y arreglos.

- Realice un programa que reciba n números enteros y despliegue cuántos de esos números son mayores al último número proporcionado. La pantalla sería similar a la siguiente.

```
Bienvenido.
Este programa le dirá cuántos números son mayores
al último número que capture.
¿Cuántos números va a capturar? 5
Indique los números:
6
7
9
10
8
Hay 2 números mayores al último número capturado.
```

- Suponga que hay un camión con 10 lugares. Todos están disponibles al principio. Realice un programa que permita al usuario reservar n lugares, e indique cuáles. Al final, despliegue los lugares aún disponibles. Nota: si el usuario da un lugar mayor de 10 simplemente se hará caso omiso de este dato. La pantalla final debe ser como sigue:

```
Este programa reserva lugares en camión.
¿Cuántos lugares desea reservar?
3
¿Cuáles
4 7 9
Situación al final:
1 2 3 4 5 6 7 8 9 10
D D D - D D - D - D
```

Uso de paquetes en Java

¿Cómo se logra que los diferentes desarrolladores de Java realicen sus clases y puedan usarse en forma masiva? La respuesta son los paquetes. Un paquete (`package`) es una agrupación o conjunto de clases afines; se refiere a las librerías (bibliotecas) de las que se vale el lenguaje para optimizar ciertos procesos; algunos lenguajes de programación se complementan con el uso de librerías propias de su lenguaje. Una

clase puede definirse como perteneciente a un `package` y puede usar otras clases definidas en ese o en otros paquetes. El nombre de una clase debe ser único dentro del paquete donde se define y cada uno corresponde a un directorio físico dentro del equipo de cómputo.

Mediante el uso de paquetes, Java permite la agrupación de clases, interfases, excepciones y constantes. Así tendremos conjuntos de estructuras de datos y de clases con algún tipo de relación en común. Es conveniente que las clases y las interfases contenidas en las mismas tengan cierta relación funcional.

Las clases se compilan por omisión en el subdirectorio `bin` de Java o en el directorio especificado para el proyecto en el ambiente integrado de desarrollo que se esté usando, pero es posible emplear otro paquete (otro subdirectorio) siempre y cuando se especifique mediante la palabra `package`. Esta indicación debe ir en la primera línea de código (sin considerar comentarios ni líneas en blanco) de modo similar al siguiente ejemplo:

```
packagemegasinapsis;
public class Ejemplo /* prosigue el código */
```

De manera general, al estructurar un programa en Java, se debe considerar lo siguiente:

- Una única sentencia de paquete (opcional).
- Las sentencias de importación deseadas (opcional).
- La declaración de una (y solo una) clase pública (`public`).
- Las clases privadas del paquete (opcional).

Haremos hincapié en que la sentencia de declaración de paquete debe ser la primera en un archivo fuente Java.

Si se quiere crear un subpaquete, habrá que almacenar el paquete dependiente (hijo) en un directorio paquete/subpaquete. Así, una clase dentro de un subpaquete como paquete.subpaquete Clase estará codificada en el archivo `paquete\subpaquete\Clase.java`. Cuando se instala el JDK se pide que se definan las variables de entorno. En este caso la variable de entorno CLASSPATHindica la(s) ruta(s) en que se buscarán los subpaquetes.

Cuando se utilizan paquetes establecidos con anterioridad o que han sido desarrollados, se hace uso de la sentencia `import`, seguida del nombre del paquete o grupo de paquetes que se necesiten. Se pueden incluir todos los elementos de un paquete o parte de ellos. Para importar todas las clases o interfases de un paquete se utiliza el metacarácter (*). En algunas ocasiones solo necesitamos importar algunas clases de un determinado paquete o subpaquete. Por ejemplo:

```
import megasinapsis.*;            // importando todas las clases
import megasinapsis.Ejemplo;      // importando solo una clase
import paquetesalida.*;
import paquete.subpaquete_1.subpaquete_2.Clase_1;
```

La API de Java provee un conjunto de paquetes que se pueden incluir en cualquier aplicación (o applet) Java.De hecho, Java incorpora por omisión el paquete `java.lang.*` a todas las clases para que reconozcan instrucciones de uso común, como `System.out.println`.

Los ambientes integrados de desarrollo utilizan una estructura de subdirectorios ya definida. Dentro de esa estructura, el subdirectorio `src` se ha reservado de *facto* para los códigos fuentes de las clases. Por ejemplo, al crear ProyectoEjemplo en el IDE JCreator se establecen los directorios `classes` para los códigos de bytes y `src` para los códigos fuente. En nuestro caso, dimos de alta el proyecto, hicimos clic derecho sobre su nombre y agregamos un nuevo folder `joseluislopez` (lo que equivale a agregar un nuevo paquete). A continuación hicimos lo mismo para el paquete `angelgutierrez`. José Luis López, usando su propio paquete, creará las clases. Ángel Gutiérrez, empleando el paquete que creó, realizará los programas de prueba. Como cada paquete corresponde a un directorio, bastará con copiarlo a la otra máquina

para integrar ambos desarrollos. La finalidad de hacer esto es lograr que mientras un estudiante programa la clase, otro codifique el programa de prueba, considerando que deben trabajar juntos y cualquier desviación de los estándares provocará errores en el tiempo de compilación o ejecución. La conformación de los paquetes puede verse en la figura 5.1 y el código en el programas 5.7.

```
//Programa 5.7a Codificación de la clase usando paquetes.
package joseluislopez;
public class Rectangulo2 {
   private double base, altura;
   public Rectangulo2(double base, double altura) {
      this.base = base;
      this.altura = altura;
   }
   public double area() {
      return base * altura;
   }
}
```

```
// Programa 5.7b Codificación del programa de prueba usando paquetes.
package angelgutierrez;
import javax.swing.*;
import joseluislopez.Rectangulo2;
public class TestRectangulo2 {
   public static void main(String[] args) {
      Rectangulo2 r; String entrada; double b, h;
      entrada = JOptionPane.showInputDialog("¿Cuál "
      + "es la base del rectángulo?");
      b = Double.parseDouble(entrada);
      entrada = JOptionPane.showInputDialog("cuál "
      + "es la altura del rectángulo?");
      h = Double.parseDouble(entrada);
      r = new Rectangulo2(b, h);
      System.out.println("El área es " + r.area());
   }
}
```

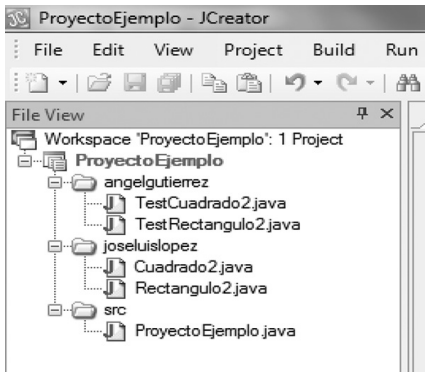


Figura 5.1 Utilización de paquetes para combinar clases y programas de prueba.

A nivel masivo, las clases de Java son realizadas por un sinnúmero de empresas. Una organización pudiera optar por utilizar el nombre del dominio de la empresa en orden inverso (com.miempresa) como el nombre del paquete. Así se tendría cierta garantía de evitar paquetes con el mismo nombre a nivel mundial.

REFLEXIÓN 5.1

Hay que aprovechar los paquetes para combinar clases y programas de prueba realizados por distintos estudiantes

Como ya hemos mencionado, muchos alumnos no detectan todos los errores de sus propios programas. Al codificar "en papel", el número de programas con algún error sintáctico, de lógica u omisión de un caso especial, sobrepasa 85%. Cuando se les deja usar el equipo de cómputo, los programas con alguna falla significativa de lógica oscilan en 24% aproximadamente. Por otra parte, en un primer intento cerca de 80% no obedece los estándares de programación. Por ello, si un mismo estudiante codifica la clase y el programa de pruebas se puede esperar que no detecte la omisión de casos especiales ni la programación fuera de estándares. Por eso se vuelve indispensable que los programas se sometan a un programa de pruebas de otro desarrollador. Aún más, si cada estudiante trabaja en su propio paquete, es posible combinar la clase con el programa de prueba de dos estudiantes al azar, como el ejemplo que pusimos.

¡La única forma de percibir la necesidad de pruebas y estándares es vivirla! Y los paquetes nos brindan una gran posibilidad a nivel pedagógico.

Bibliotecas de clases (API de Java)

Cuando se instala el JDK en cualquiera de sus últimas versiones se incluyen bibliotecas o librerías con clases de tipos estándar a las que pueden referenciar los desarrolladores y programadores en lenguaje Java. Sin temor a falla alguna, se pueden incluir en los programas asegurando su portabilidad. Hay una muy buena documentación relacionada con estas clases, ya sea impresa o con acceso mediante direcciones de internet. A este conjunto de paquetes (o bibliotecas) se le conoce como API (Application Programming Interface) de Java.

Los paquetes API sirven para agrupar clases, interfases y tipos a manera de unidades estructuradas en librerías, con lo que se mejora la organización y estructura de las clases, con la seguridad de poder emplearlas en forma homogénea en cualquier aplicación que desarrollemos. También se facilita el intercambio o reutilización de código entre programadores. Por ejemplo, cuando utilizamos las sentencias:

```
import java.util.*;
import java.util.ArrayList;
```

importamos la biblioteca de utilidades, que en este caso forma parte de la biblioteca que se distribuye o se instala como parte del JDK estándar de Java.

Los paquetes básicos de la API de Java se pueden clasificar como sigue:

Paquetes de utilidades

- *java.lang:* fundamental para el lenguaje. Incluye clases como String o StringBuffer.
- *java.io:* para la entrada y salida a través de flujos de datos y archivo del sistema.
- *java.util:* contiene colecciones de datos y clases, el modelo de eventos, facilidades horarias, generación aleatoria de números y otras clases de utilidad.
- *java.math:* clases para realizar aritmética con la precisión que se desee.
- *java.text:* clases e interfases para manejo de texto, fechas, números y mensajes de una manera independiente a los lenguajes naturales.
- *java.security:* clases e interfases para seguridad en Java.

Paquetes para el desarrollo gráfico

- *java.applet:* para crear applets y clases que estas utilizan para comunicarse con su contexto.
- *java.awt:* para crear interfases con el usuario y dibujar imágenes y gráficos.

- *javax.swing:* conjunto de componentes gráficos que funcionan igual en todas las plataformas que Java soporta. En general, se combina con el paquete awt.
- *javax.accesibility:* para dar soporte a clases de accesibilidad para personas discapacitadas.
- *java.beans:* para el desarrollo de Java Beans.

Paquetes para el desarrollo en red

- *java.net:* clases para aplicaciones de red.
- *java.sql:* paquete que contiene el JDBC para conexión de programas Java con bases de datos.
- *java.rmi:* paquete RMI, para localizar objetos remotos, comunicarse con ellos e incluso enviar objetos como parámetros de un objeto a otro.
- *org.omg.CORBA:* facilita la posibilidad de utilizar OMG CORBA para la conexión entre objetos distribuidos, aunque estén codificados en distintos lenguajes.
- *org.omb.CosNaming:* da servicio al IDL de Java, similar al RMI pero en CORBA.

La utilización de las API de Java está en función de las necesidades del programador, por lo que su implementación en los programas depende de lo que se quiera hacer. Es importante resaltar que existe una gran documentación al respecto, por lo que aquí solo se mencionan algunas de ellas.

Los modificadores `static` y `final` de Java

Existen dos modificadores de uso común en Java: `static` y `final`.

La palabra `static` se emplea cuando un atributo de valor constante o un método de cálculo deben existir en forma independiente del número de objetos que se crearán de esa clase. Si se coloca la palabra `static` para los atributos, estos se convierten en atributos de clase o variables estáticas: en este caso la variable es única para todas las instancias de la clase, pues no tendría sentido crear un nuevo lugar de memoria por cada objeto de la clase. Cuando usamos `staticfinal` se dice que creamos una constante de clase. Al utilizar los atributos estáticos, las instancias de una clase, además del espacio propio para variables de instancia, comparten un espacio común.

La referencia a una variable estática se hace con el nombre de la clase y el nombre del atributo; no se utiliza el nombre del objeto.

Modificador `final` indica que las clases hijas no pueden cambiar el atributo o método. Por su naturaleza, es común ver juntos a ambos modificadores, aunque desde el punto de vista conceptual son independientes por completo.

Cuando se utiliza la palabra clave `final`, se indica que una variable es de tipo constante; es decir, que no admitirá cambios después de su declaración, y la asignación de valor final determina que un atributo no puede sobreescribirse: no funcionará como una variable tradicional, sino como una constante; por ello, toda constante declarada con final deberá ser inicializada al momento de declararla. También puede emplearse este modificador para las clases, así no pueden ser heredadas, o para los métodos, de esta manera no podrán ser sobreescritos.Es una buena práctica de codificación utilizar identificadores en mayúsculas para las variables que sean `final`. Por ejemplo:

```
static int x = 5;
static void metodoGlobal() {
public static final String NOMBREPROG = "Java";
public static final int COSTO_INICIAL = 100;
```

A manera de ejemplo, realicemos un programa que reciba el radio de un círculo y devuelva su área. El valor de π se encuentra en la clase `Math` con el encabezado:

```
public static final double PI
```

lo cual significa que es un atributo público (accesible desde cualquier clase), estático (se invoca con el nombre de la clase y el atributo sin que sea necesario crear ningún objeto, es decir, `Math.PI ), final` (la clase hija no puede modificar su valor) y, por último, es de tipo double. Por otra parte, se mantiene la costumbre de usar mayúsculas cuando se trata de un valor constante. El siguiente código muestra el programa de prueba para corroborar que π está definido en forma adecuada en la clase Math.[3]

```
import javax.swing.*;
public class PruebaDePi {
   public static void main(String[] args) {
      String entrada; double diametro, perimetro;
      entrada = JOptionPane.showInputDialog("¿Cuál es el diámetro: ");
      diametro = Double.parseDouble(entrada);
      perimetro = Math.PI * diametro;
      System.out.println("El perímetro del círculo es " + perimetro);
   }
}
```

EJERCICIOS SUGERIDOS

1. Cambie en C++ el programa Cuadrado2 y observe si hubo algún efecto en el programa ejecutable TestCuadrado2. Haga lo propio en Java y verifique si existieron repercusiones al trabajar con TestCuadrado2.class. ¿Coincide lo que observó con lo expresado en párrafos anteriores?
2. Reproduzca el ejemplo del perímetro del círculo para verificar que puede accederse al valor de π con la sintaxis Clase.ATRIBUTO (con el nombre de la clase por ser de tipo static y con el atributo con mayúsculas porque existe la convención de que las constantes se escriben con estas). También confirme que se accede al valor sin necesidad de crear ningún objeto; justamente, porque se trata de un atributo estático.

5.2 Asociación, agregación y composición

La asociación en clases implica la utilización de una clase pública por otra que también es pública.[4] Por ejemplo, un programa de Java puede emplear la clase `Math` para calcular la raíz cuadrada de un número a través del método `sqrt`. De hecho, todos los programas de prueba que hemos realizado hasta al momento mandan llamar a la clase que están verificando. De esta manera, `TestCuadrado` *usa* la clase `Cuadrado` para confirmar que trabaja en forma adecuada.

La agregación y la composición implican que una clase tiene como atributo un objeto de otra clase. La diferencia radica en que en la composición el objeto "pertenece" a la clase y desaparece si se elimina el objeto principal; la relación es subordinada y de tipo 1:N. Por ejemplo, una tarjeta de débito pertenece a un determinado banco; si desapareciera el banco se cancelarían de una u otra forma todas sus tarjetas de débito. La composición se puede abordar de manera natural si uno se acostumbra a que un objeto se puede pasar como argumento. Para ese momento —claro está— el objeto ya debe estar creado.

En la agregación las dos entidades existen por separado y se pueden relacionar al mismo tiempo con otros objetos, por ello la relación es N:M. Por ejemplo, la relación de una empresa con sus proveedores. Si la empresa se declarara en bancarrota, el proveedor seguiría existiendo porque también trabaja con otras organizaciones (o por lo menos ha tenido la posibilidad de hacerlo). Es importante ubicar si se trata de una agregación o una composición, porque el diseño de clases y bases de datos son distintos para cada una de ellas.

[3] La idea original del programa fue tomada de http://www.w3api.com/wiki/Java:Math.PI

[4] En un archivo pudiera haber definidas varias clases. En tal caso debería haber solo una clase pública. En estas situaciones no suele hablarse de asociación.

Un ejemplo con asociación, composición y agregación en Java

Especificación de requerimientos: se necesita codificar las clases Llanta, Coche, Cliente y PosiblesClientes, con los atributos mostrados en el diagrama de clases de la figura 5.2, obtenido a través del software de uso gratuito StarUML. Del mismo diagrama se puede notar que la clase Coche guarda una relación de composición con respecto a la clase Llanta. La clase PosiblesClientes tiene una relación de agregación con respecto a Coche y Cliente. Por último, TestPosiblesClientes conlleva una asociación con PosiblesClientes y servirá como una clase de pruebas para verificar que las otras clases funcionan en forma correcta y que PosiblesClientes permite guardar una relación de los clientes y los coches que les interesan.

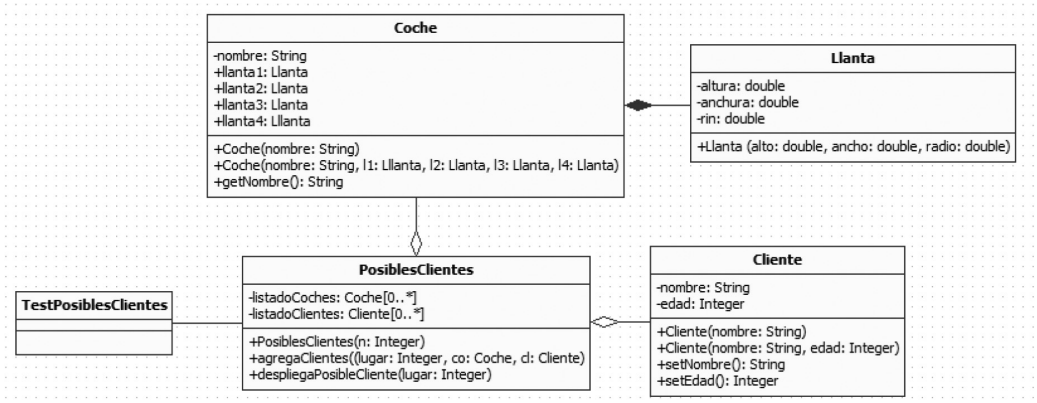


Figura 5.2 Ejemplo de asociación, composición y agregación.

Como requisitos no funcionales, se requiere que para efectos didácticos se despliegue un mensaje cada que se cree un objeto tipo Llanta, Cliente y Coche, así como una relación de los posibles clientes.

La clase Llanta tiene tres atributos privados, los que corresponden a la especificación del tipo de llanta, más su respectivo constructor. Como se comentó en las especificaciones, se desplegará un letrero cada que se cree un objeto Llanta (véase programa 5.8a).

La clase Coche tiene como atributo privado al nombre del coche y a cuatro objetos tipo Llanta con ámbito público (por el momento, hemos pasado por alto la llanta de refacción). La clase Llanta y la clase Coche tienen una relación de composición. Observe que tiene dos constructores: uno asigna únicamente el nombre y otro, además, recibe a los cuatro objetos tipo Llanta. Si se usa el primero, luego se pueden asignar las llantas de manera directa, pues son de ámbito público (véase programa 5.8b).

REFLEXIÓN 5.2

Metodología pedagógica para la programación orientada a objetos

En este momento, hay que hacer hincapié en que el diagrama de clases proviene de una especificación de los requerimientos del sistema realizada a través de alguna metodología de desarrollo de software y expresada con diagramas de casos de uso, historias de usuario o alguna otra técnica (véase el capítulo 7). De estos requerimientos se obtuvo, justamente, el diagrama de clases como un "puente" entre las especificaciones del sistema y la estructura interna del mismo. Hasta aquí seguimos con la idea de fabricar componentes con su prueba respectiva ("focos" y "probadores de focos"), aunque se comienza a esbozar que los programas de prueba —TestClase— tienen cierta similitud con las pantallas finales hacia el usuario).

Por otra parte, es conveniente que los estudiantes utilicen algún software para la diagramación de UML. Después de un análisis general, recomendamos **StarUML** por ser de uso libre, fácil manejo y tener una estructura muy apegada a Proceso Unificado, pero reconocemos que puede haber otras opciones también válidas.

La clase Cliente tiene dos atributos privados: el nombre y la edad. En un constructor se asigna el nombre y la edad se asigna a cero. En otro, se solicita el nombre y la edad (véase programa 5.8c).

La clase PosiblesClientes permite dar de alta una relación que identifica los clientes que están interesados en determinados coches; guarda una relación de agregación con Cliente y Coche. En el constructor se establece el número de datos a proporcionar; el método agrega Cliente permite dar de alta un cliente y el coche que le interesa, mientras despliega PosibleCliente también lo hará con los datos del cliente y el coche con respecto al lugar indicado, que se recibe como parámetro (véase programa 5.8d).

Por último, TestPosiblesClientes es un programa de prueba que da de alta tres combinaciones de coche y clientes, y despliega en pantalla esa relación de manera similar a un reporte. Observe que se combinan objetos Coche y Cliente con diferentes parámetros; en otras palabras, invocando a constructores diferentes. En la pantalla se muestran mensajes conforme se crean los objetos; es decir, a la par que se dan de alta en memoria desde su respectivo constructor (véase programa 5.8e).

```
//Programa 5.8a La clase Llanta, con sus atributos y constructor.
public class Llanta {
   private double altura;
   private double anchura;
   private double rin;
   public Llanta (double alto, double ancho, double radio) {
      altura = alto; anchura = ancho; rin = radio;
      System.out.println("Construyendo una llanta altura = "
         + alto + ", anchura = " + ancho + ", rin = " + rin);
   }
}
```

```
//Programa 5.8b La clase Coche, con sobrecarga en constructor.
public class Coche {
   private String nombre;
   public Llanta llanta1, llanta2, llanta3, llanta4;
   public Coche(String nombre) {
   System.out.println("Construyendo un coche " + nombre + ".");
      this.nombre = nombre;
   }
   public Coche(String nombre,
      Llanta l1, Llanta l2, Llanta l3, Llanta l4) {
      this.nombre = nombre;
      llanta1 = l1; llanta2 = l2; llanta3 = l3; llanta4 = l4;
      System.out.println("Construyendo un coche " + nombre +
      " con llantas asignadas.");
   }
   public String setNombre() {
      return nombre;
   }
}
```

```
//Programa 5.8c La clase Cliente, con sobrecarga en constructor.
public class Cliente {
   private String nombre;
   private int edad;
   public Cliente(String nombre) {
      this.nombre = nombre;
   edad = 0;
   System.out.println("Dando de alta un cliente " + nombre + ".");
```

```
   }
   public Cliente(String nombre, int edad) {
      this.nombre = nombre;
      this.edad = edad;
      System.out.println("Dando de alta un cliente " + nombre
      + " con edad.");
   }
   public String setNombre() {
      return nombre;
   }
   public int setEdad() {
      return edad;
   }
}
```

```
//Programa 5.8d La clase PosiblesClientes.
public class PosiblesClientes {
   private Coche[] listadoCoches;
   private Cliente[] listadoClientes;
   public PosiblesClientes(int n) {
      listadoCoches = new Coche[n];
      listadoClientes = new Cliente[n];
   }
   public void agregaCliente(int lugar, Coche co, Cliente cl ) {
      listadoCoches[lugar] = co;
      listadoClientes[lugar] = cl;
   }
   public void despliegaPosibleCliente(int lugar) {
      System.out.println(lugar + ") " +
      listadoClientes[lugar].setNombre() + " (" +
      listadoClientes[lugar].setEdad() + ") " +
      listadoCoches[lugar].setNombre());
   }
}
```

```
//Programa 5.8e Programa de prueba TestPosiblesClientes.
public class TestPosiblesClientes {
   public static void main (String [] args) {
      int numClientes = 3;
      PosiblesClientes reporte;
      Llanta l1 = new Llanta(165.0, 14.0, 15.0);
      Llanta l2 = new Llanta(165.0, 14.0, 15.0);
      Llanta l3 = new Llanta(165.0, 14.0, 15.0);
      Llanta l4 = new Llanta(165.0, 14.0, 15.0);
      reporte = new PosiblesClientes(3);
      reporte.agregaCliente(0, new Coche("matiz",
         l1,l2,l3, l4), new Cliente("Juan"));
      reporte.agregaCliente(1, new Coche("polo"), new Cliente("Juan"));
      reporte.agregaCliente(2, new Coche("matiz"),
         new Cliente("Pedro", 46));
      System.out.println("\nListado de clientes y "
         + "coches que les interesan:");
      for (int i=0; i<=2; i++) {
         reporte.despliegaPosibleCliente(i);
      }
   }
}
```

Diferencias entre C++ y Java en la refactorización del código

Un tema muy importante para el desarrollo de software y que por lo general se suele dejar de lado es la refactorización de código. ¿Cómo hacer cambios en un programa sin que esto afecte a las otras partes del sistema? Esta pregunta lleva a otra que no es tan sencilla de resolver: ¿quiénes usan la clase que se está modificando?

C + + liga las clases y las conjunta en un programa ejecutable. Si se modifican las clases originales no resulta afectado dicho programa ejecutable hasta que se cambie el programa principal. Esto puede resultar una gran ventaja porque no hay que tener cuidado en lo que sucede con el sistema al alterar una clase, pero puede ocasionar un gran conflicto a largo plazo, pues cuando se necesite variar el programa principal será necesario revisar si sus clases han cambiado o no.

Por otra parte, Java interpreta el código de bytes en tiempo real, por lo cual al cambiar una clase en forma automática será tomado el nuevo código de bytes por todas las clases que lo usen. Eso obliga a que cada modificación sea pensada para que no afecte a ninguna otra parte del sistema: más trabajo a corto plazo, pero una menor posibilidad de conflictos futuros.

La refactorización de código nunca será un trabajo sencillo, pero se puede disminuir su complejidad si se siguen varios consejos metodológicos y de programación:

- El ámbito de las variables locales, y los atributos y métodos privados, pueden modificarse con la seguridad de que no se afecta ninguna otra parte del sistema, a condición de que no varíen los atributos y métodos públicos.
- En ocasiones, pueden crearse métodos sobrecargados. De esta forma se tiene la seguridad de que no se modifica ningún método y, por tanto, todas las demás partes del sistema seguirán trabajando de manera adecuada.
- No es aconsejable utilizar clases de Java señaladas como obsoletas pues tienden a desaparecer, lo cual podría ocasionar a mediano plazo algún problema de compatibilidad o portabilidad de código.
- En principio, no debe haber ningún problema si agrega variables y métodos públicos. Una situación similar se da si se agregan campos o tablas a la base de datos.
- Si modifican las variables o métodos públicos, hay que revisar todas aquellas partes del sistema que los utilizan para readecuarlas. Una situación similar se da si se cambian nombres de campos o tablas.
- Explotar los aspectos metodológicos en forma adecuada: la necesidad de prototipos, el enfoque incremental y el trabajo por versiones (véase capítulo 7).
- En el diseño del sistema deben guardarse en una única fuente aquellos valores que alimentan a todo el sistema: tanto parámetros del usuario como la configuración externa e interna del sistema. Por ejemplo, nombre de la escuela, colores del sistema, conexión a la base de datos, etcétera.
- Incorporar en el sistema funciones de usuario que, aunque no son comunes, es muy probable que se den a mediano plazo, como incorporar un nuevo salón en una escuela.

REFLEXIÓN 5.3

La refactorización, un tema siempre pendiente

El mantenimiento al código ocupa gran parte de las tareas de las áreas de desarrollo de software. Sin embargo, suele ser un tema relegado en libros de ingeniería de software y los cursos de ingeniería en computación y carreras afines. Es necesario darle mayor peso al tema y diseñar ejercicios para que los estudiantes se enfrenten a la refactorización, un tema que no es sencillo de abordar.

EJERCICIO SUGERIDO

- Reproduzca los ejercicios descritos en su compilador para verificar que todo funcione en forma correcta. Después de eso realice los siguientes ajustes: a) incorpore otro atributo para la llanta de emergencia a la Clase coche, b) aumente un método para incorporar la llanta de emergencia a la Clase coche y c) modifique los programas de prueba para verificar que, en efecto, la nueva llanta forma parte del coche.

Nota: con este ejercicio se pretende reforzar los conceptos tratados a través de un proceso de refactorización de código.

5.3 Herencia, polimorfismo, clases abstractas e interfases

Las personas percibimos la realidad como abstracciones de un conjunto de objetos interrelacionados. La verdadera potencia de la programación orientada a objetos radica en su capacidad para reflejar la **abstracción**. Por eso trata de descomponer un problema en un conjunto de unidades (objetos) más pequeñas y que se subordinan al problema en cuestión. Después aíslan de manera conceptual los atributos y comportamiento (métodos) comunes a un determinado conjunto de objetos para almacenarlos como una clase.

La herencia permite crear una clase nueva (subclase o clase derivada) que tiene el mismo comportamiento que otra (superclase o clase base) y además extiende o adapta ese comportamiento a unas necesidades específicas, aunque en todo momento debe respetar las reglas impuestas por la clase padre. En otras palabras, una clase hija —también conocida como subclase— hereda de una clase padre todos sus atributos y sus métodos. Para ello se emplea en Java la palabra extends y en C++ se indica a través de dos puntos, como veremos en esta misma sección. Esto implica que la clase hija no puede acceder a los atributos privados de la clase padre; solo puede consultarlos o modificarlos a través de los métodos que brinda la propia clase padre. Tampoco puede dejar de usar a su constructor, ya sea que se invoque de manera automática como en C++ o Java cuando no hay llamada explícita, o se ejecute a través de la palabra reservada super en Java, que siempre será la primera instrucción del constructor de la clase hija: una llamada al constructor de la clase padre.

Una subclase puede añadir métodos y atributos. En todo momento se considerará que los atributos y métodos de la clase hija son los propios más los heredados.

Una subclase también puede sustituir alguno de sus métodos, lo cual se conoce como **sobreescritura**. La sobreescritura de métodos cancela al método de la clase padre y lo sustituye por el de la hija (a menos que el padre la haya declarado `final`).

Quiza el concepto más importante para entender a la herencia es que la clase hija es una clase especializada que heredó todos los atributos del padre. Un ejemplo muy ilustrativo se da en el caso de medicina. Un especialista en cataratas es un oftalmólogo que se especializó en ese campo. A su vez, un oftalmólogo es un médico general que se especializó. Por eso el especialista en cataratas tiene tres "carreras" y tres permisos para el ejercicio profesional, porque puede ejercer bajo tres figuras diferentes: es un especialista en cataratas, sigue siendo un oftalmólogo y sigue siendo un médico general (véase figura 5.3).

La herencia se representa con una flecha hacia arriba. Eso es relativamente fácil de notar. Pero hay algo más que por lo normal pasa desapercibido. Usemos de nuevo el ejemplo e imaginemos que deseamos saber si la oftalmóloga Ameyalli López llevó un curso a lo largo de su preparación profesional. Lo buscaríamos en sus conocimientos de cataratas y, si no lo encontramos, nos dirigiríamos a sus conocimientos de oftalmología en general. Si tampoco lo hallamos, nos dirigiríamos a sus conocimientos de medicina general. El sentido de la flecha indica el orden en que se buscarán los métodos y los atributos cuando existe herencia. Por eso es relativamente fácil para los compiladores de C++ y Java detectar qué método aplicar en caso de **sobreescritura**.

Figura 5.3 El diagrama de herencia representa la especialización de clases. Un especialista en cataratas sigue siendo un oftalmólogo, que a su vez es un médico general.

Aquí conviene hacer una aclaración: en Java todos los objetos de Java heredan de la clase **Object**. En otras palabras, todos los objetos de Java también son **Object** y, en consecuencia, podemos enviar cualquier objeto a un método que espere un **Object**. Esto se hace de manera automática, sin necesidad de utilizar la palabra extends.

Conviene aclarar que además de los modificadores privados y públicos existen el protegido y el "amigo" (default), conforme lo muestra la figura 5.3. Si no se indica el tipo de modificador, por omisión se asume que será de tipo **default**. En este instante es necesario hacer una acotación: el alcance privado, protegido y público son iguales en C++ y Java. Esto no sucede con el alcance "amigo" (`friend`). Si en Java no se pone ningún alcance, se asumirá por omisión que se está dando acceso al atributo a todas las clases del mismo paquete. En C++ se requiere hacer explícito el permiso a los atributos privados de la clase señalada.

"Para declarar una función o una clase como `friend` de otra clase [en C++], es necesario hacerlo en la declaración de la clase que debe autorizar el acceso a sus datos privados. Esto se puede hacer de modo indiferente en la zona de los datos o en la de los datos privados." Un ejemplo de declaración de una clase `friend` podría ser el que sigue:

Cuadro 5.5 Los modificadores de Java y su alcance

Modificador	Acceso por la propia clase	Acceso especial	Acceso por subclases	Acceso por cualquier clase
private (C++ y Java)	x			
default (Java)	x	clases del mismo paquete		
friend (C++)	x	clases permitidas explícitamente		
protected (C++ y Java)	x	x	x	
public (C++ y Java)	x	x	x	x

```
class Cualquiera {
   friend class Amiga;
   private:
   int secreto;
};
```

"Es muy importante tener en cuenta que esta relación funciona solo en una dirección; es decir, las funciones miembro de la clase Amiga pueden acceder a las variables privadas de la clase Cualquiera; por ejemplo a la variable entera secreto, pero esto no es cierto en sentido inverso: las funciones miembro de la clase

Cualquiera no pueden acceder a un dato privado de la clase Amiga… Si se quiere que varias clases tengan acceso mutuo a todas las variables y funciones miembro privadas, cada una debe declararse como `friend` en todas las demás, para conseguir una relación recíproca."[5]

Una tarjeta de débito y crédito en C++ y Java

Como ejemplo, presentaremos una tarjeta de débito y una tarjeta de crédito, aunque solo con algunos elementos. Pensemos un concepto que al principio nos resultará un poco extraño: una tarjeta de crédito es una clase especializada de una tarjeta de débito. Piénselo de la siguiente manera: si usted tuviera una tarjeta de crédito con una línea de crédito cero no podría gastar más allá de su saldo y no le cobrarían por disposición en efectivo ni por intereses, porque no está pidiendo prestado.

La figura 5.4 nos muestra el diagrama de clases, dibujado de nueva cuenta en la herramienta StarUML. Observe que la clase tarjeta de crédito hereda el atributo saldo, que por supuesto tendrá el derecho y la obligación de modificar (por eso es de tipo protegido); sobreescribe la disposición de efectivo, pues ahora puede disponer hasta la línea de crédito. No obstante, el método meter y consultar puede ser aprovechado sin ninguna modificación. El código se muestra en el programa 5.9,[6] con su respectivo programa de prueba.[7]

La palabra `protected` en la clase tarjetadebito indica que el atributo saldo podrá ser consultado y modificado por los hijos. La instrucción `class tarjetacredito: public tarjetadebito` marca que `tarjetadebito` heredará de la clase `tarjetacredito`. Las dos clases tienen un método disponer, lo cual significa que ha sido sobreescrito. Por tanto, se empleará el que no corresponda al tipo de objeto que invoca al método. La combinación de instrucciones:

```
tarjetadebito ejemplo1;ejemplo1.disponer(monto);
```

invocará el método disponer de tarjetadebito, mientras que la dupla de comandos:

```
tarjetacredito ejemplo2; ejemplo2.disponer(monto);
```

harán lo propio pero con el método disponer de tarjetacredito.

Sucede algo diferente con los métodos que no fueron sobreescritos. La instrucción `ejemplo2.meter(monto);` hará que el compilador busque de manera infructuosa el método meter en la clase `tarjetacredito`. Al no encontrarlo lo buscará en la clase padre `tarjetadebito`, el cual aplicará finalmente. Como podrá notarse, tanto el objeto `ejemplo1` como el `objeto2` terminarán por aplicar el mismo método.

```
/*Programa 5.9 El código en C++ de la clase tarjetadebito y la clase
heredada tarjetacredito, con su respectiva salida. Los datos son
proporcionados por el usuario.
#include "conio.h"
#include "stdio.h"
#include "iostream"
using namespace std;
```

[5] Bustamente, Paul; Aguinaga, Iker; Aybar; Miguel; Olaizola, Luis y Lazacano, Iñigo, *Aprenda C++ avanzado como si estuviera en primero*; Escuela Superior de Ingenieros, Campus Tecnológico de la Universidad de Navarra, San Sebastián, 2004, p. 24.

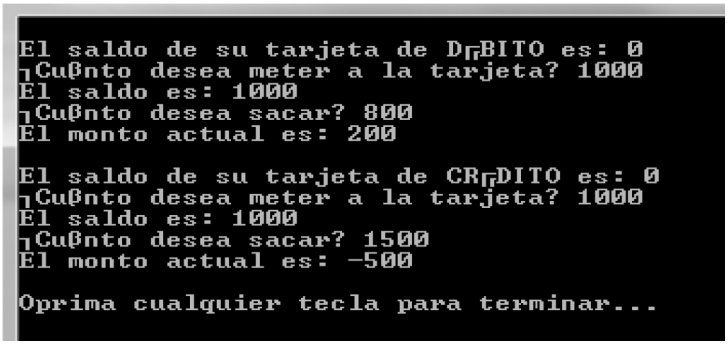
[6] Recordamos al estudiante y al docente que todos los programas de los capítulos 1 a 3 se probaron a detalle en C++ y Zinjai (en las primeras pruebas han resultados portables también con Code:: Blocks y C para Android). Los de los capítulos 4 y 5 han sido probados hasta el momento en Dev C++ de manera detallada.

[7] Lamentablemente, aún tenemos problemas de portabilidad en Dev C++ con los acentos y caracteres especiales. Hemos encontrado algunas soluciones, pero ninguna rápida y sencilla. Al aplicarlas a los cursos introductorios, notamos que los estudiantes concentran su atención en colocar de manera adecuada cada carácter y dejan en segundo término los conceptos medulares del tema. Los distintos caminos para la solución se publicarán en www.megasinapsis.com.mx

```
class tarjetadebito {
   protected:
      float saldo;
   public:
      tarjetadebito() { saldo = 0; }
      float consultar () { return saldo; }
      void meter(float monto) { saldo += monto; }
      void disponer(float monto) {
         if (monto <= saldo)
            saldo -= monto;
      }
};

class tarjetacredito: public tarjetadebito {
   private:
      float lineacredito;
   public:
      tarjetacredito() { lineacredito = 1000; }
      void cambiarlimite (float dato) { lineacredito = dato; }
      void disponer(float monto) {
         if (monto <= saldo + lineacredito)
            saldo -= monto;
      }
};

int main() {
   float monto;
   tarjetadebito ejemplo1;
   cout<< endl << "El saldo de su tarjeta de DÉBITO es: ";
   cout<< ejemplo1.consultar() << endl;
   cout<< "¿Cuánto desea meter a la tarjeta? ";
   cin>> monto;
   ejemplo1.meter(monto);
   cout<< "El saldo es: " << ejemplo1.consultar() << endl;
   cout<< "¿Cuánto desea sacar? ";
   cin>> monto;
   ejemplo1.disponer(monto);
   cout<< "El monto actual es: " << ejemplo1.consultar()<<endl;
   tarjetacredito ejemplo2;
   cout<< endl << "El saldo de su tarjeta de CRÉDITO es: ";
   cout<< ejemplo2.consultar() << endl;
   cout<< "¿Cuánto desea meter a la tarjeta? ";
   cin>> monto;
   ejemplo2.meter(monto);
   cout<< "El saldo es: " << ejemplo2.consultar() << endl;
   cout<< "¿Cuánto desea sacar? ";
   cin>> monto;
   ejemplo2.disponer(monto);
   cout<< "El monto actual es: " << ejemplo2.consultar()<<endl;
   printf("\nOprima cualquier tecla para terminar...\n");
   getch();
}
```



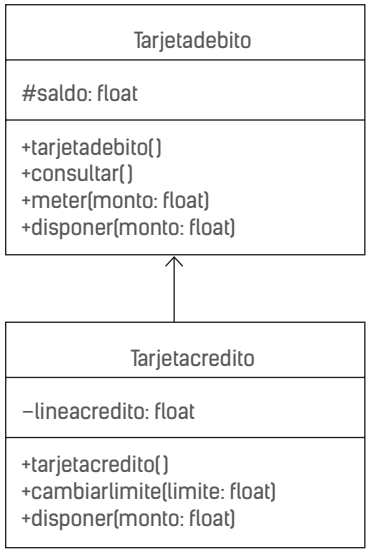


Figura 5.4 Una clase tarjetacredito que hereda de una tarjetadebito.

Java aplica los mismos principios generales, pero con una codificación diferente. Observe que fueron cambiados los nombres de las clases y los métodos para evitar confusiones entre el ejemplo de C++ y el de Java y se incorporó una sobrecarga en el constructor de TarjetaCredito2, posibilidad que también está presente en C++ (véase figura 5.5). Además, variamos un poco el diagrama de UML para incorporar varios elementos más, con la idea de repasar posibilidades de Java que no se encuentran en lenguaje C:

- Hay un atributo privado cuenta de tipo String. En Java existe una clase String que se incorpora de manera natural a los programas de Java, que simplifica ese tema con respecto al lenguaje C.
- Se emplea la notación de camello, así como setAtributo y getAtributo para los nombres de métodos.
- Se incorporaron atributos y métodos nuevos, simplemente para recalcar que la clase hija puede añadir las características que juzgue convenientes.
- Sacar Dinero devuelve un dato lógico (boolean), aprovechando que este tipo de dato se incorporó en Java.
- Por último, en Java se hace más obvio que cada Clase es un componente de software que debe ser probado a través de su programa de pruebas (nuevamente: el "foco" y el "probador de focos"). Sin embargo, si se cambiara el programa para probar la clase por una pantalla de usuario real ya se tendría una pequeña aplicación funcional sin necesidad de cambiar prácticamente nada de las clases (o tal vez con ajustes mínimos).

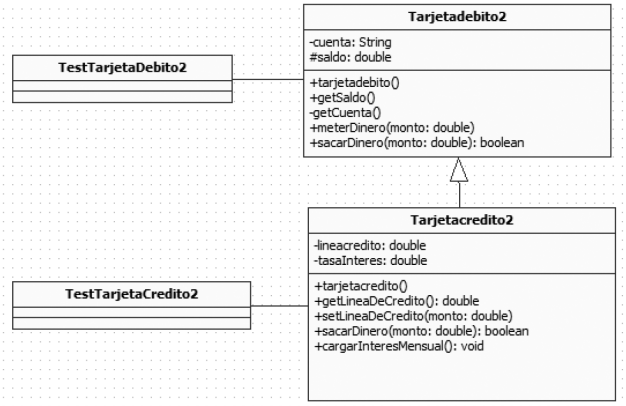


Figura 5.5 Una clase TarjetaCredito2 que hereda de una TarjetaDebito2.

En cuanto al código de la herencia, observe que la palabra `extends` marca la herencia en Java. También observe que la primera instrucción del constructor de la subclase es la llamada al constructor de la clase padre a través de la palabra `super`. Por lo demás, todos los principios siguen vigentes. Dicho de otra manera, existen diferencias significativas entre la codificación en C++ y en Java, pero en esencia utilizan los mismos principios, los que son comunes a la programación orientada a objetos.

REFLEXIÓN 5.4

¿Separar C++ y Java o enseñarlos juntos?

Este libro optó por una estructura poco común. Presenta el ejemplo y el código en C++ y en Java, tratando de respetar el estilo y las ventajas de ambos lenguajes, por lo cual puede haber ligeras variaciones. En algunos casos, por cuestiones de espacio, solo se encuentra el ejemplo en Java o en C++, sobre todo en características nuevas de Java con respecto al lenguaje C.

¿Qué tan pedagógico es? Tuvimos un caso similar cuando un grupo de profesores se inclinaba en su momento por Pascal y otro por lenguaje C. Se hizo incluso el experimento de mostrar el problema en ambos lenguajes. De manera preliminar podemos dar las siguientes sugerencias:

1. Lo más importante es llevar una secuencia coherente del curso y asentar los conceptos que servirán como base para conceptos nuevos.
2. No vemos problemas en presentar los ejemplos en ambos lenguajes. Sin embargo, proponemos que los estudiantes hagan sus ejercicios en uno solo, porque se suelen presentar confusiones en sintaxis, en equipamientos de laboratorios o problemas de portabilidad que en ocasiones quitan demasiado tiempo. También puede optarse porque hagan todos los ejercicios en un lenguaje y solo algunos en el otro.
3. Será de ayuda que queden bien asentadas las características de un lenguaje que no tiene el otro y los conceptos que son comunes a ambos. Por ejemplo, la herencia es tema común; el recolector de basura solo existe en Java.
4. Extremadamente importante: conceptualizar las clases como componentes de software que deben ser probados y no como un programa completo. La forma pedagógica de abordar la programación orientada a objetos es distinta a la programación estructurada.
5. Asentar bien la diagramación en UML y definir si en C++ se trabajará de manera tradicional (clase, herencia y prueba en un solo archivo) o cada clase tendrá su propio archivo como en Java. Recuerde la forma para crear los .h que se vio en el tema de subrutinas y que puede ser llevado más a fondo.

De cualquier forma, este es un tema abierto en el cual no existen conclusiones definitivas.

```
//Programa 5.10a Código en Java de la clase TarjetaDebito2.
public class TarjetaDebito2 {
   private String cuenta;
   protected double saldo;
   public TarjetaDebito2 (String cuenta, double saldo) {
      this.cuenta = cuenta;
      this.saldo = saldo;
   }
   public double getSaldo() {
      return saldo;
   }
   public String getCuenta() {
      return cuenta;
   }
   public void meterDinero (double monto) {
      saldo += monto;
   }
```

```
   public boolean sacarDinero (double monto) {
      if (monto <= saldo) {
         saldo -= monto;
         return true;
      }
   else
      return false;
   }
}
```

```
/*Programa 5.10b Código en Java del programa de prueba TestTarjetaDebito2,
   con su respectiva corrida.*/
public class TestTarjetaDebito2 {
   public static void main (String[] args) {
      TarjetaDebito2 tarjeta = new TarjetaDebito2("Cuenta 0001", 1000.0);
      System.out.println("El saldo de la tarjeta " + tarjeta.getCuenta()
         + " después de arrancar con $1000.00 es: "
         + tarjeta.getSaldo());
      tarjeta.meterDinero(500.0);
      System.out.println("El saldo después de meter $500.00 es: "
         + tarjeta.getSaldo());
      if (tarjeta.sacarDinero(2000.00) == true)
         System.out.println("El saldo después de sacar $2000.00 es: "
            + tarjeta.getSaldo());
      else
      System.out.println("No fue posible sacar $2000.00");
      if (tarjeta.sacarDinero(1000.00) == true)
         System.out.println("El saldo después de sacar $1000.00 es: "
            + tarjeta.getSaldo());
      else
         System.out.println("No fue posible sacar $1000.00");
 }
}
```

```
El saldo de la tarjeta Cuenta 0001 después de arrancar con $1000.00 es: 1000.0
El saldo después de meter $500.00 es: 1500.0
No fue posible sacar $2000.00
El saldo después de sacar $1000.00 es: 500.0

Process completed.
```

```
//Programa 5.10c Código en Java de la clase TarjetaCredito2.
public class TarjetaCredito2 extends TarjetaDebito2 {
   private double lineaDeCredito;
   private double tasaInteres = 0.03;
   public TarjetaCredito2 (String cuenta, double saldo) {
      super(cuenta, saldo);
      lineaDeCredito = 0;
   }
   public TarjetaCredito2 (String cuenta, double saldo,
      double lineaDeCredito) {
      super(cuenta, saldo);
   this.lineaDeCredito = lineaDeCredito;
   }
   public double getLineaDeCredito() {
      return lineaDeCredito;
   }
```

```
   public void setLineaDeCredito(double monto) {
      lineaDeCredito = monto;
   }
   public boolean sacarDinero (double monto) {
      if (monto <= saldo + lineaDeCredito) {
         saldo -= monto;
         return true;
   }
      else
         return false;
   }
   public void cargarInteresMensual() {
      if (getSaldo() < 0)
         saldo += getSaldo() * tasaInteres;
   }
}
```

```
Programa 5.10d Código en Java del programa de prueba TestTarjetaCredito2,
/*con su respectiva corrida.*/
public class TestTarjetaCredito2 {
   public static void main (String[] args) {
      TarjetaCredito2 tarjeta =
         new TarjetaCredito2("Cuenta 0001", 2000.0, 1000.0);
      System.out.println("El saldo de la tarjeta " + tarjeta.getCuenta()
         + " después de arrancar con $2000.00 es: " + tarjeta.getSaldo());
      System.out.println("La línea de crédito es: "
         + tarjeta.getLineaDeCredito());
      tarjeta.meterDinero(500.0);
      System.out.println("El saldo después de meter $500.00 es: "
         + tarjeta.getSaldo());
      if (tarjeta.sacarDinero(3000.00) == true) {
         System.out.println("El saldo después de sacar $3000.00 es: "
         + tarjeta.getSaldo());
      }
      else
         System.out.println("No fue posible sacar $3000.00");
      if (tarjeta.sacarDinero(1000.00) == true)
         System.out.println("El saldo después de sacar $1000.00 es: "
         + tarjeta.getSaldo());
      else
         System.out.println("No fue posible sacar $1000.00");
      tarjeta.cargarInteresMensual();
      System.out.println("El saldo después de cargar intereses es: "
         + tarjeta.getSaldo());
      tarjeta.meterDinero(1500.0);
      System.out.println("El saldo después de meter $1500.00 es: "
         + tarjeta.getSaldo());
      }
}
```

```
El saldo de la tarjeta Cuenta 0001 después de arrancar con $2000.00 es: 2000.0
La línea de crédito es: 1000.0
El saldo después de meter $500.00 es: 2500.0
El saldo después de sacar $3000.00 es: -500.0
No fue posible sacar $1000.00
El saldo después de cargar intereses es: -515.0
El saldo después de meter $1500.00 es: 985.0

Process completed.
```

Polimorfismo

Comencemos con una pregunta que descontrola al principio. Si se solicita como parámetro a un médico general, ¿podríamos mandar llamar a un oftalmólogo? El código sería similar al siguiente:

```
MedicoGeneral objetoEjemplo;
objetoEjemplo = new Oftalmologo("Juan");
```

La respuesta es sí, porque un oftalmólogo sigue siendo un médico general (su licenciatura es en medicina general y su posgrado es en oftalmología). Si le suena extraño no se preocupe. ¡El polimorfismo es extraño! Y resultará aún más si lo ve con el ejemplo de tarjetas de crédito.

¿Ve normal esta instrucción?

```
TarjetaDebito ejemploDeb = new TarjetaDebito (10000.0);
```

Suponemos que sí. ¿Y esta otra?

```
TarjetaDebito ejemploCre = new TarjetaCredito (10000.0, 1500.0);
```

La del médico suena relativamente obvia, pero el caso es similar. Ya se había comentado que una tarjeta de crédito también puede comportarse como tarjeta de débito. Si sintetiza, Java permite que una instancia de clase de tipo padre también pueda usarse para crear un objeto de tipo hijo. Este objeto de manera natural podrá acceder a los métodos que le son comunes a la clase y a la subclase. Cuando se explota esa posibilidad se está hablando de **polimorfismo**.

Ahora imagine que le solicitaron crear un método de seguridad para las tarjetas de débito desde una clase externa. Algo como lo siguiente:

```
void gestionarSeguridad (TarjetaDebito aux) {
 // aquí va el código de la rutina de seguridad

}
```

La instrucción `objetoDeSeguridad(ejemploDeb);` invocaría a ese método. Donde `objetoDeSeguridad` sería un objeto de la clase que contiene al método y `ejemploDeb` es un objeto tipo `TarjetaDebito` (idéntico al que acabamos de crear).

Un mes después surge la necesidad de aplicarlo también a tarjetas de crédito. ¡Eureka, no tiene que hacer nada! También puede invocarlo con objetos tipo tarjeta de crédito: `objetoDeSeguridad(ejemploCre);` puesto que las tarjetas de crédito también son tarjetas de débito. Pero existe una restricción: desde `ejemploCre` solo podrá invocar los métodos que son comunes a tarjetas de débito y de crédito. ¿El motivo? La rutina `objetoDeSeguridad` no sabe si recibió una tarjeta de débito o de crédito.

Eso me recordó un caso muy extraño que tuve en una sucursal bancaria. Solicitaba un retiro del saldo a favor de mi tarjeta bancaria (había pagado de más), pero no me había llegado la reposición de la tarjeta por parte del banco y no se sabía el tipo y estatus de la nueva. El cajero con toda la razón del mundo me decía que sin tarjeta vigente no podía hacerlo; yo, con toda la razón del mundo le contestaba que ese era problema del banco (primera lección: cuidado con las personas que "tenemos toda la razón del mundo"). Al final, la gerente permitió con sabiduría salomónica que hiciera la disposición pero únicamente por el saldo a favor, con lo cual quedé de acuerdo. Su razonamiento era sencillo: todos los estatus en que podía terminar la situación de mi tarjeta permitían retirar el saldo a favor, pero algunos impedirían usar el crédito. De manera parecida actuará `objetoDeSeguridad` hasta no averiguar si la tarjeta es de débito o de crédito.

Ahora compliquemos las cosas. Imagine que requiere averiguar cuál es la línea de crédito. Existe el método `getLineaDeCredito`, pero solo aplica para las tarjetas de crédito y si lo manda llamar para una tarjeta de débito el programa marcará un error. ¿Cómo averiguar de qué tipo es determinado objeto dentro de una herencia? Para eso existe el operador `instanceof`. Como en múltiples ocasiones no se sabe de antemano de qué tipo es el objeto —por ejemplo, cuando se recibe como parámetro— Java permite preguntarlo a través del operador `instanceof`. De esta forma nos podemos asegurar que en todo momento empleamos el tipo correcto de datos.

El paso siguiente es declarar una instancia de la subclase y, mediante un forzamiento de tipos, depositar ahí al objeto que fue declarado en la clase. De esta manera:

`if (ejemploCre instanceof TarjetaCredito)` devolvería verdadero
`if (ejemploCre instanceof TarjetaDebito)` devolvería verdadero
`if (ejemploDeb instanceof TarjetaDebito)` devolvería verdadero
`if (ejemploDeb instanceof TarjetaCredito)` devolvería falso

Si se considera ese enfoque, se podrá deducir con facilidadque si hay un objeto x, la sentencia `if (x instanceof Object)` devolverá verdadero sin importar qué tipo de objeto sea x, pues todas las clases heredan en forma automática de `Object`. A manera de ejemplo, el programa 5.11 presenta un arreglo de objetos en el cual se mezclan objetos de tipo TarjetaDebito y TarjetaCredito.

```
/*Programa 5.11 Ejemplo del uso de polimorfismo con tarjetas de débito y
crédito. Se incluye la salida del programa.*/
public class TestArregloDeTarjetas {
   public static void main (String[] args) {
      int limite = 2;
      TarjetaDebito2 arreglo[];
      TarjetaCredito2 auxCre;
      arreglo = new TarjetaDebito2[limite];
      arreglo[0] = new TarjetaDebito2("Cuenta #UNO", 2000.0);
      arreglo[1] = new TarjetaCredito2("Cuenta #DOS", 2000.0, 1000.0);
      for (int i = 0; i < limite; i++) {
         if (arreglo[i] instanceof TarjetaCredito2) {
            System.out.print(i + ") " + arreglo[i].getCuenta() +
               " Crédito con línea de crédito " );
            auxCre = (TarjetaCredito2) arreglo[i];
            System.out.println(auxCre.getLineaDeCredito() + ".");
         }
         else
            System.out.println(i + ") " + arreglo[i].getCuenta() +
               " Débito sin línea de crédito " );
      }
   }
}
```

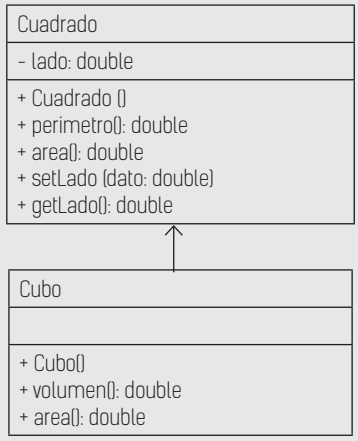
```
0) Cuenta #UNO Débito sin línea de crédito
1) Cuenta #DOS Crédito con línea de crédito 1000.0.

Process completed.
```

EJERCICIOS SUGERIDOS

1. Conteste las siguientes preguntas:
   - ¿Cuántos atributos tienen las clases TarjetaDebito2 y TarjetaCredito2 que hemos estado trabajando?
   - ¿Cuántos métodos tiene cada una de las clases?
   - ¿En qué parte se da sobreescritura?
   - ¿En qué parte se da sobrecarga?
2. (AVANZADO) Cree una clase **TarjetaDebito2** y una clase **TarjetaCredito2** con la misma funcionalidad que la mostrada, pero supone que el saldo fuera de tipo privado.
3. Cree una clase **Circulo** cuyo único atributo sea diámetro y tenga los métodos **area** y **perimetro**. Después cree una subclase **Cilindro** que añada el método altura y tenga codificados los métodos **area** y **volumen**.
4. A partir del siguiente diagrama, realice las clases **Cuadrado**, **Cubo** y **TestCubo**. Tenga especial cuidado en respetar las especificaciones de los diagramas de clase. La clase TestCubo debe desplegar lo siguiente:

```
Suponiendo que hay un cubo de lado 3.
El área es 54.
El volumen es 27.
```



5. ¿Qué desplegará el siguiente programa?

```
#include <stdlib.h>
#include <conio.h>
#include <iostream>
using namespace std;

class bienvenida {
   public:
      bienvenida() { cout << "constructor de bienvenida\n"; }
      void mensaje() { cout << "buenos días"; }
};

class bienvenida2 : public bienvenida {
   public:
      bienvenida2() { cout << "constructor de bienvenida 2\n"; }
      void mensaje() { cout << "a todos"; }
};
main() {
   bienvenida2 salida;
```

```
   salida.mensaje();
   cout<< endl<<endl<<"Oprima cualquier tecla para terminar...";
   getch();
}
```

6. ¿Qué desplegará el siguiente programa?

```
#include "conio.h"
#include "stdio.h"
#include "iostream"
using namespace std;
class bacteria {
   private:
      int individuos;
   public:
      long int consultar (void) { return individuos; }
      void aumentar (void) { individuos++; };
      void morir (void) { individuos--; };
      bacteria() { individuos = 10; }; // saldo inicial
};

class bacterias2: public bacteria {
   public:
      void duplicar (void) {
      int c = consultar(), i;
      for (i=0; i<c; i++)
         aumentar();
   }
   void aumentar (int dato=1) {
      int j;
      for (j=0; j<dato; j++)
         bacteria::aumentar();
   }
};

int main(void) {
      bacterias2 colonia1; int i;
      int dato;
      cout<< "Este programa simulará una población de bacterias\n";
      cout<< "El saldo inicial: " << colonia1.consultar() << endl;
      for (i = 1; i <= 300; i++)
      colonia1.aumentar();
   cout<< "Después de aumentar 300 existen: " <<
   colonia1.consultar() << endl;;
   for (i = 1; i <= 200; i++)
      colonia1.morir();
   cout<< "Fallecieron 200. Ahora existen: " <<
   colonia1.consultar() << endl;
   cout<< "¿Cuántas aumentaron por una nueva colonia? "<<endl;
   cin>> dato;
   colonia1.aumentar(dato);
   cout<< "Después existen: " << colonia1.consultar() << endl;
   printf("\nOprima cualquier tecla para terminar...\n");
   getch();
}
```

7. Suponga el siguiente caso:

```
public class Cuadrado
public double area() { return lado * lado }

public class Cubo extends Cuadrado
public double area() { return lado * lado * 6 }
public double volumen () { return lado * lado * lado }
```

a) ¿Qué desplegará?

```
Cuadrado c = new Cubo(5.0);
System.out.println(c.area());
```

Respuesta:

b) ¿Con qué instrucciones se podría desplegar el volumen del cubo?

8. Aplique las siguientes mejoras a las clases TarjetaDebito2 y TarjetaCredito2, con los cambios correspondientes en los programas de prueba.

a) Incorpore un método que permita consultar la tasa de interés en la clase TarjetaCredito2.

b) Añada otro método que permita modificar la línea de crédito.

c) Agregue el nombre del dueño de la tarjeta a la clase TarjetaDebito.

Clases abstractas

En ocasiones se tienen los atributos de una clase y algunos métodos, pero de otros se desconocen los detalles. No obstante, y a pesar de no tener toda la información, se puede definir la interfaz comunicación hacia el exterior (nombre, parámetros de entrada y tipo de datos de retorno). Con estos datos se establece un método abstracto, cuya implementación será establecida por la subclase.

Cuando uno o más métodos son abstractos se considera que la clase como tal también lo es. Una clase abstracta tiene constructor, pero no es posible crear objetos de estas clases porque están "incompletas". Las subclases están obligadas a instrumentar los métodos abstractos y añadir los atributos y métodos que crean convenientes. Por eso se consideran "clases completas" y todos los objetos deben ser de alguna de esas subclases.

A manera de ejemplo, hablaremos de la propuesta que planteó el Instituto Politécnico Nacional en 2006 para cambiar el sistema de calificaciones. En ambos sistemas, el anterior y el nuevo, se tienen tres evaluaciones parciales y un examen global. Al final, solo habría una calificación semestral. Como se verá, los atributos de entrada y salida se conocían a la perfección, no así los detalles, que todavía estaban en discusión en el H. Consejo General Consultivo.

De esta forma se tenía el reglamento actual y un reglamento desconocido hasta ese momento. Ambos compartirían los atributos y encabezados de los métodos para obtener la calificación final, pero se diferenciarían en los detalles. Se trata, sin duda, de un caso ideal para crear una clase abstracta y dos subclases.

El código de la clase abstracta establecería los atributos (que pueden ser privados, protegidos o públicos), el constructor y, en este caso, el método para cambiar las calificaciones, pero dejaría sin detalle la obtención del promedio, que forzosamente debe establecer cada una de las subclases, también conocidas como clases hijas. El programa 5.12 muestra la codificación. Observe que la clase tiene que llevar la pa-

labra `abstract`, al igual que cada uno de los métodos abstractos. Si existe un método abstracto toda la clase también lo será; si la clase es abstracta, al menos debe haber un método abstracto.

```
//Programa 5.12 Una clase abstracta para obtener promedios semestrales.
public abstract class Curso {
   protected int parcial1, parcial2, parcial3, global;
   public Curso (int c1, int c2, int c3, int eg) {
      parcial1 = c1; parcial2 = c2; parcial3 = c3; global = eg;
   }
   public void cambiarCalificaciones (int c1, int c2, int c3, int eg) {
      parcial1 = c1; parcial2 = c2; parcial3 = c3; global = eg;
   }
   public abstract int obtenerPromedio();
}
```

La forma de promediar vigente hasta 2006 establecía que el promedio semestral sería la calificación más alta entre el promedio de las tres evaluaciones parciales y el examen global. Otra aclaración importante: el promedio de las evaluaciones parciales se redondea al entero más cercano si se alcanza al menos un promedio de 6.0, pero si es menor los decimales simplemente no se tomarán en cuenta. El cuadro 5.6 muestra algunos ejemplos:

Cuadro 5.6

Evaluaciones parciales	Promedio de evaluaciones	Examen final	Calificación semestral
6 6 6	6	7	7
8 7 8	8	9	9
7 7 7	7	5	7

Como ya se conoce el detalle del procedimiento, puede programarse la primera subclase, a la cual llamaremos Curso2000. El constructor de la clase Curso2000 es, en realidad, una llamada al constructor padre sin agregar ningún otro elemento. Por otra parte, se hace el detalle del método obtenerPromedio conforme a las características descritas (véanse los programas 5.13 y 5.14).

```
/*Programa 5.13 Clase que implementa al método abstracto
   obtenerPromedio.*/
public class Curso2000 extends Curso {
   public Curso2000 (int c1, int c2, int c3, int eg) {
      super(c1, c2, c3, eg);
   }
   public int obtenerPromedio() {
      int sumaParcial = parcial1 + parcial2 + parcial3;
      int promedio;
      if (sumaParcial < 18)
         promedio = (int) (sumaParcial / 3.0);
      else
         promedio = (int) (sumaParcial / 3.0 + 0.5);
      if (global > promedio)
         return global;
      else
         return promedio;
   }
}
```

```
/*Programa 5.14 Test de prueba de la clase Curso2000, con su respectiva
corrida.*/
public class TestCurso2000 {
   public static void main (String[] args) {
      Curso2000 cursoEjemplo = new Curso2000 (7, 7, 6, 5);
      System.out.println("Parciales de 7, 7 y 8, y final de 5 es: "
         + cursoEjemplo.obtenerPromedio());
      cursoEjemplo.cambiarCalificaciones (8, 8, 8, 7);
      System.out.println("Parciales de 8, 8 y 8, y final de 7 es: "
         + cursoEjemplo.obtenerPromedio());
   }
}

Parciales de 7, 7 y 8, y final de 5 es: 7
Parciales de 8, 8 y 8, y final de 7 es: 8

Process completed
```

Cuando se aprobó la nueva forma de promediar se establecieron las siguientes reglas: se coloca la calificación más alta entre el promedio de las tres evaluaciones parciales y el examen global, pero solo si el promedio de las tres evaluaciones parciales era al menos de ocho. Si fuera menor, se colocaría directamente el resultado del examen global. El cuadro 5.7 muestra algunos ejemplos, remarcando el caso diferente a la clase Curso2006.

Cuadro 5.7

Evaluaciones parciales	Promedio de evaluaciones	Examen final	Calificación semestral
6 6 6	6	7	7
8 7 8	8	9	9
7 7 7	**7**	**5**	**5**

La codificación en realidad varía poco con respecto al ejemplo anterior; se dejará como ejercicio al estudiante.[8]

Interfases de Java[9]

Las interfases de Java son expresiones de diseño de clases ya conceptualizadas pero aún no implementadas; es decir, se conoce lo que debe hacerse, pero aún no se tiene la solución en todos los casos. Por ello, se declaran métodos abstractos y constantes: para que después puedan ser implementados según las necesidades del programa.

Visto desde otro enfoque, cuando una clase contiene uno o más métodos abstractos se dice que es una clase abstracta. Cuando todos los métodos son abstractos ya no se habla propiamente de una clase abstracta, sino de una interfase. Las interfases sirven para enunciar la estructura mínima que deben contener las clases hijas. De alguna forma heredan solamente "obligaciones".

Se utiliza la sentencia `interfase`, de la misma forma en que usa la sentencia `class`.

[8] Como anécdota, el nuevo Reglamento fue derogado unas semanas después de haberse publicado y años después volvió a modificarse con mayores adecuaciones, pero dejando de lado esta propuesta.

[9] El diccionario de la Real Academia Española reconoce la palabra interfaz o interfase; las definiciones que brinda se acercan al concepto manejado para computación, pero al mismo tiempo no son totalmente adecuadas. La palabra interface no existe como tal en el idioma español. A nivel código debe tenerse cuidado, pues interface es palabra reservada del lenguaje Java.

```
interface UnaInterfaz {
   int VALOR = 100;
   int metodoAbstracto( int parametro );
}
```

Observe que las variables adoptan la declaración en mayúsculas, pues en realidad actuarán como constantes `final`. Los métodos comienzan con su declaración y se omite el cuerpo, pues todos los métodos son abstractos o métodos sin implementación.

Debido a que las interfases carecen de funcionalidad, necesitan de algún mecanismo para dar cuerpo a sus métodos. Para lograr esto se utiliza la palabra reservada `implements` en la declaración de una clase, lo cual significa que la subclase implementará todos y cada uno de los métodos señalados en la interfase. Por ejemplo:

```
class I_Interfaz implements UnaInterfaz{
   int sumando = VALOR;
   int metodoAbstracto( int parametro ){
      return ( parametro + sumando );
   }
}
```

este ejemplo se observa que deben codificarse todos los métodos que determina la interfase *(metodoAbstracto())*. Para lograrlo puede usar las constantes que define la propia interfase durante la declaración de la clase.

Una interfase no puede implementar otra interfase, aunque sí puede extenderla al utilizar la palabra reservada `extends`. Una interfase puede heredar de más de una interfase antecesora.

```
interface InterfazMultiple extends Interfaz1,Interfaz2{ }
```

Una clase solo puede tener una clase antecesora, pero puede implementar más de una interfase, por eso suele haber una cierta ambigüedad acerca de si Java admite o no herencias múltiples. Aunque en sentido estricto no lo permite, en la práctica sí:

```
class UnaClase extends SuPadre implements Interfaz_1,Interfaz_2{ }
```

El ejemplo típico de herencia múltiple es el
que se presenta con la herencia en diamante (figura 5.6).

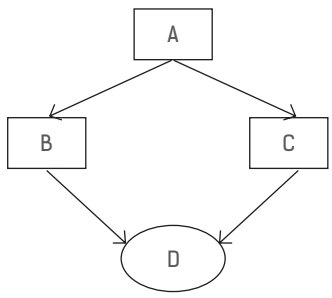


Figura 5.6 Herencia en diamante en Java.

En Java es necesario que las clases A, B y C sean interfases, y que la clase D sea una clase (que recibe la herencia múltiple):

```
interface A{ }
interface B extends A{ }
interface C extends A{ }
class D implements B,C{ }
```

A manera de ejemplo, suponga que desea hacer diversas clases de polígonos, y para cada uno debe haber —al menos— una rutina que calcule el área y otra que lo haga con el perímetro. Se puede definir una interfase **Poligono** para obligar a todas las subclases (triángulos, rectángulos, círculos, etc.) a que implementen estos métodos. A continuación se muestra el diagrama, junto con el detalle de la clase hija **Rectangulo** y la mención de la clase **Triangulo** (véase figura 5.7).

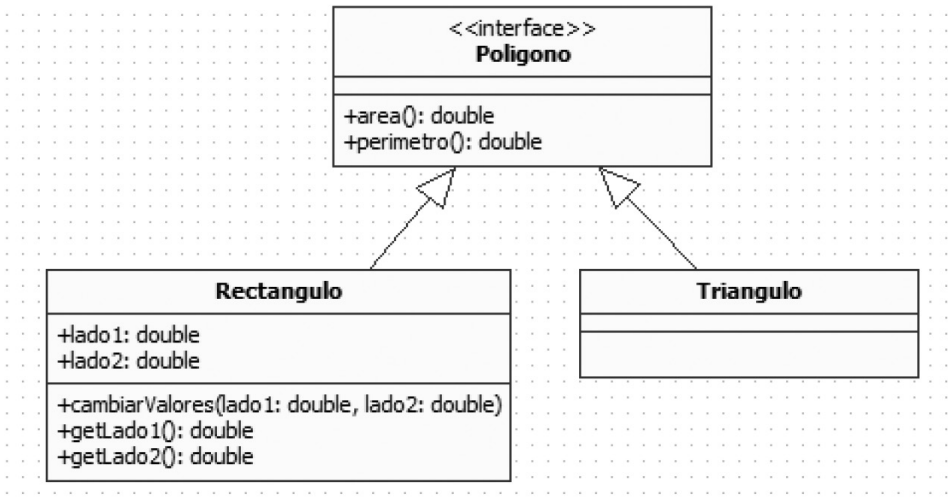


Figura 5.7 Diagrama de la interfase Poligono y la clase hija Rectangulo.

A continuación se muestra la interfase Poligono. Observe que en lugar de la palabra `class` se emplea interface y que el código es asombrosamente sencillo, pues solo expresa obligaciones. Como se trata de una interfase, no es necesario poner la palabra `abstract` a cada método.

```
public interface Poligono {
   public double area();
   public double perimetro();
}
```

En cuanto a la clase Rectangulo, es muy parecida a la forma en que trabaja la herencia normal, pero utiliza la palabra `implements` en lugar de `extends`. El programa de prueba se hace de la manera normal, como lo hemos venido trabajando durante todo el libro, y se le dejará al estudiante su elaboración (véase programa 5.15).

```
//Programa 5.15 Clase Rectangulo, implementación de la clase Poligono.
public class Rectangulo implements Poligono {
   private double lado1, lado2;
   public Rectangulo(double dato1, double dato2) {
      lado1 = dato1;
      lado2 = dato2;
   }
   public double area() {
      return lado1 * lado2;
   }
   public double perimetro() {
      return lado1 * 2 + lado2 * 2;
   }
   public void cambiarValores(double d1, double d2) {
      lado1 = d1;
      lado2 = d2;
   }
   public double getLado1() {
      return lado1;
   }
   public double getLado2() {
      return lado2;
   }
}
```

EJERCICIOS SUGERIDOS

De manera intencional se quedaron algunos puntos por completar en los temas de clases abstractas e interfases. De lo que se trata, justamente, es de terminarlos:

1. Realice la clase Curso2006 que quedó pendiente en el tema de clases abstractas, con su respectivo programa de prueba.
2. Realice el programa de prueba para la clase Rectangulo que ya quedó programada en el tema de interfases. Además, codifique la clase Triangulo, considerando que los métodos perimetro y area deben funcionar para cualquier tipo de triángulo.

5.4 Excepciones y lectura de datos

Hasta este momento hemos dejado varios temas pendientes. Algunos no podrán cubrirse por las limitaciones de espacio. Necesitaríamos multiplicar el número de páginas para abarcar todo lo deseado acerca de la programación estructurada, la programación orientada a objetos y los aspectos metodológicos básicos de la programación. Aun suponiendo que nuestro conocimiento fuera el suficiente (lo cual no es cierto), lo más probable es que la publicación fuera inaccesible para la gran mayoría de los estudiantes. Sin embargo, al menos hay una cuestión que no puede quedar fuera en un material básico que abarque el tema de Java: las excepciones.

Los programas que hicimos hasta este momento detienen su ejecución cuando el usuario no teclea el tipo de datos correcto. En los programas profesionales pueden darse otros casos: el archivo buscado no existe, se rompió la conexión de red, no hay acceso a la base de datos, etc. El control de flujo en un programa en Java tiene muchas opciones más que se pueden detectar mediante una técnica denominada **gestión de excepciones**. A través de esta técnica se crean métodos que "disparan" una excepción cuando sucede algo anormal y formas para que esta situación sea detectada por el método que invocó este método. Así se evita la repetición de código y se previenen posibles errores, informando cuando se detecte una condición anormal durante la ejecución del programa.

Algunos tipos de excepciones:

- **Error:** indican problemas graves, que suelen ser no recuperables.
- **Exception:** no definitivas, se detectan fuera del tiempo de ejecución.
- **RuntimeException:** se presentan durante la ejecución del programa.

Las excepciones tienen la clase base **Throwable** incluida en el paquete **java.lang**. Sus métodos son:

- **Trowable (String mensaje).** Constructor; la cadena es opcional.
- **Throwable fillInStackTrace().** Llena la pila de traza de ejecución.
- **String getLocalizedMessage().** Crea una descripción local de este objeto.
- **String getMessage().** Devuelve la cadena de error del objeto.
- **void printStackTrace.** Imprime este objeto y su traza en el flujo de los parámetros o en la salida estándar.
- **String toString.** Devuelve una breve descripción del objeto.

Es conveniente estructurar los programas en dos partes para poder tener un sistema funcional de gestión de excepciones. Primero debe definirse qué partes de los programas crean una excepción y en qué condiciones. Enseguida, comprobar en ciertas partes de los programas si se ha producido una excepción. Para realizar lo anterior, se utilizan las palabras reservadas **try**, **catch** y **finally**.

La importancia de esto es definir cómo controlar y qué hacer con la excepción que se creó. Aquí podemos hacer uso de la cláusula **catch**, y especificar en ella qué acción queremos realizar.

```
try {
 // códigoen caso normal
} catch(tipo_excepcion1 e) {
 // código de control para un tipo de excepción
} catch(tipo_excepcion2 e) {
 // Código de control para otro tipo de excepción
}
```

Como se observa, se pueden anidar sentencias `catch`. Debe tomarse en cuenta que existe una jerarquía de excepciones, pero todas ellas heredan de la clase `Exception`. Por ello es obligatorio indicarla en el último lugar de las excepciones. Así se permite que intercepte la excepción alguna situación más específica y solo si no está entre esas opciones, forzosamente la tomaría `Exception`. En cualquier caso, se ejecutará el bloque de código `catch` cuyo parámetro sea del tipo de la excepción ejecutada.

Existen casos en que se quiere insertar fragmentos de código que se ejecuten tras la gestión de las excepciones, sin importar si se ha tratado de una excepción (`catch`) o de un flujo normal (`try`). Casos típicos son cerrar un archivo ocupado o la conexión a la base de datos. Para estas situaciones se puede utilizar la sentencia **finally**, que se ejecutará tras el bloque **try** o **catch**:

```
try {
 // flujo normal
} catch( Exception e ) {
 // flujo de excepción
} finally {
// se ejecutara tras try o catch
}
```

Para comprobar si se cumple alguna condición de excepción, se utilizan las palabras reservadas **throw** y **throws**. La excepción se lanza mediante la sentencia `throw`:

```
if ( condicion_excepcion == true )
throw new excepcion();
```

Si se ha creado un objeto de la clase **excepcion**, este deber ser instanciado antes de ser utilizado, por ello los métodos que pueden lanzar excepciones deben saber en su declaración cuáles son esas excepciones. Se puede lanzar en el método más de una excepción, indicándolas en su declaración y separándolas por comas. Para ello se utiliza la sentencia `throws`:

```
tipo_devuelto metodo() throws excep1, excep2 {
 // Código para las excepciones
}
```

De seguro a estas alturas, quien lea estas líneas piensa que necesita un ejemplo concreto. Nosotros pensaríamos lo mismo si estuviéramos en su lugar. En consecuencia, aquí mostraremos uno de los más sencillos: la división entre cero, que nunca está permitida en operaciones aritméticas porque el resultado es indefinido.

La clase Division tiene un método estático llamado dividir que hace una división. A pesar de lo sencillo del cálculo, puede llegar a generar una excepción: una posible división entre cero. Desde el encabezado se avisa a través de la palabra `throws` que el método generará excepciones. Por eso, en el caso de que el dividendo sea cero se generará una excepción a través de la palabra `throw`.

El programa principal sigue la ejecución normal desde `try`. Pero si se generara una excepción —en este caso, una división entre cero— la pararía y haría lo que indica `catch`, lo cual incluye obtener el mensaje que arrojó la excepción con el método `getMessage`. Se haya seguido el flujo normal o no, se llevará a cabo lo que indica `finally`: en este caso, un texto meramente didáctico.

```
/*Programa 5.16 Clase Division, con una excepción para detectar una
posible división entre cero. Se muestra la ejecución correspondiente.*/
public class Division {
   public static double dividir (double x, double y) throws Exception {
      if (y != 0)
         return (x / y);
      else
         throw new Exception("Se generó una división entre cero.");
   }
   public static void main (String[] args) {
      double dato1 = 5.0, dato2 = 0.0, resultado = 0.0;
      try {
         resultado = Division.dividir(dato1, dato2);
         System.out.println(dato1 + " / " + dato2 + " = " + resultado);
      }
      catch (Exception e) {
         System.out.println(e.getMessage());
      }
      finally {
         System.out.println("Se intentó una división entre " + dato1
            + " y " + dato2 + ".");
      }
   }
}
```

```
Se generó una división entre cero.
Se intentó una división entre 5.0 y 0.0.

Process completed.
```

Por cierto, tuvimos la oportunidad de aplicar en otro lenguaje este enfoque de excepciones al diseñar y codificar una nómina. En todos los casos que se recibían datos externos para el cálculo se validaba que no se hicieran divisiones entre cero. Si esto sucedía, se grababa el error en un archivo, se omitía el cálculo específico del concepto en ese único empleado y se proseguía con los demás. ¡Ahorró muchos dolores de cabeza este enfoque y causó bastantes en las partes del código en las que no fuimos tan precavidos!

Las envolturas de datos simples (wrappers class)

La API de Java contiene un conjunto de clases especiales para modificar el comportamiento de los tipos de datos simples conocidos como envolturas de tipo simple. De esta forma se utilizan instancias de clase (objetos) en lugar de tipos de datos primitivos, ya sea que se dejen como objetos para utilizarlos en otras clases o se transformen de manera inmediata después de las variables correspondientes. Estas clases varían su nombre según el tipo de datos primitivos que manejan:

- **Double:** soporte al tipo double.
- **Float:** soporte al tipo float.
- **Integer:** soporte a los tipos int, short y byte.
- **Long:** soporte al tipo long.
- **Character:** envoltura del tipo char.
- **Boolean:** envoltorio al tipo boolean.

Una de sus aplicaciones principales es la lectura de datos del usuario. ¿Qué sucedería si este teclea letras cuando se le solicitó un número? Por lo regular, el programa dejaría de funcionar, como en los ejemplos que hemos mostrado a lo largo del libro, muy semejantes al programa 5.17.[10] Sin duda, una ventana emergente atractiva, pero sin robustez. Observe que lo tecleado por el usuario llega a una cadena, transformada a través del método estático `parseInt` de la clase `Integer` a un entero llamado **edad**, que por último se usa en forma normal. Sin embargo, no se aplicaron excepciones y por ello el programa detiene de manera abrupta su ejecución si el usuario se equivoca en lo que respecta al tipo de dato.

```
//Programa 5.17 Una lectura desde el teclado sin robustez.
import javax.swing.*;
public class PruebaEntrada1 {
   public static void main (String[] args) {
      String entrada =
         JOptionPane.showInputDialog("¿Cuántos años tienes?");
      int edad = Integer.parseInt(entrada);
      System.out.println("Dentro de un año tendrás: " +
         (edad + 1) + " años.");
      System.exit(0);
   }
}
```

Para que no suceda esto, basta con incorporar las excepciones que generan las envolturas de datos simples. En el programa 5.18 se muestra prácticamente el mismo código pero aplicando la excepción, primero a un posible error en el tipo de dato tecleado (`NumberFormatException`). Cabe aclarar que, aunque en este contexto es sumamente difícil que se genere cualquier otro tipo de excepción, se conserva la política de poner la excepción padre al final (`Exception`) para protegerse de cualquier flujo anormal no previsto.

[10] Los programas 5.17 y 5.18 fueron adaptados del libro de Horstmann y Cornell. *Java 2 Volumen I-Fundamentos*. Pearson, séptima edición, p.87.

```
/*Programa 5.18 Una lectura desde el teclado con robustez a través de
   excepciones.*/
import javax.swing.*;
public class PruebaEntrada2 {
   public static void main (String[] args) {
      String entrada =
         JOptionPane.showInputDialog("¿Cuántos años tienes?");
      try {
         int edad = Integer.parseInt(entrada);
         System.out.println("Dentro de un año tendrás: " +
            (edad + 1) + " años.");
      }
      catch (NumberFormatException e) {
         String cadena = e.getMessage();
         System.out.println(cadena);
         System.out.println("Error de formato en el número.");
      }
      catch(Exception e) {
         String cadena = e.getMessage();
         System.out.println(cadena);
         System.out.println("Se detectó un error en la ejecución.");
      }
      }
}
```

Existe otra forma un poco más laboriosa de leer, pero con mayores alcances (véase programa 5.19). Primero se abre un canal de entrada a través de la sentencia `InputStreamReader`. El canal de entrada del teclado es `System.in`, pero pueden utilizarse otros, entre ellos un archivo de texto. Este canal de entrada asignado a la variable `isr`, que en este caso asociamos al teclado, se relaciona a su vez con un búfer para almacenar de modo temporal los datos, identificado con la variable `br`. De ese búfer se capturarán los datos y aplicarán las excepciones de la forma en que ya se comentó.

```
/*Programa 5.19 Una lectura desde el teclado utilizando un canal y búfer
de entrada.*/
import java.io.*;
public class PruebaEntrada4 {
   public static void main (String[] args) {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String cadena;
      try {
         System.out.println("Escriba su nombre: ");
         cadena = br.readLine();
         System.out.println("Hola " +cadena+ " teclee un número entero: ");
         int entero = Integer.parseInt(br.readLine());
         System.out.println("El número introducido fue el " + entero);
         System.out.println("Introduzca ahora un número decimal: ");
         cadena = br.readLine();
         Double d = new Double(cadena);
         double real = d.doubleValue();
         System.out.println("El número es " + real);
      }
      catch(Exception e) {
         System.out.println("Error al leer los datos");
      }
   }
}
```

EJERCICIOS SUGERIDOS

1. Modifique el programa 5.18 para que, si el usuario no teclea un dato adecuado, se le brinden dos oportunidades más antes de dar por terminado el programa.
2. Existe otra forma parecida a las ya vistas para leer del teclado: a través de la clase Scanner. Busque un ejemplo en internet.

EJERCICIO INTEGRADOR

Desde el capítulo 4 se planteó el ejemplo de una pila en C y en C++. En el programa 5.20 se presenta la codificación en Java sin explicación alguna porque todos los elementos se han proporcionado a lo largo de este capítulo y el anterior. Repase en el libro o en internet cualquier duda que tuviera (Sugerencia: primero vea que el código trabaja sin problemas).

```
//Programa 5.20 Una pila en Java instrumentada a través de arreglos.
import java.io.*;
public class Pila {
   private int MAXIMO;
   private int ultimo;
   private char datos[];
   public Pila (int max) {
      datos = new char[max];
      MAXIMO = max;
      ultimo = -1;
   }
   public boolean vacia () {
      if (ultimo == -1) return true;
      else return false;
   }
   public boolean llena () {
      if (ultimo == MAXIMO-1) return true;
      else return false;
   }
   public void meter (char dato) {
      ultimo++;
      datos[ultimo] = dato;
   }
   public char sacar () {
      int aux = ultimo;
      ultimo--;
      return datos[aux];
   }

   public static void main (String [] args) {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String cadena; char aux;
      int longitud;
      try {
         System.out.println("Programa que despliega un mensajeal revés.");
         System.out.println("Ingresa la cadena para la pila. " +
            "Termina con ENTER..");
         cadena = br.readLine();
         longitud = cadena.length();
```

```
        Pila p;
        p = new Pila(longitud);
        System.out.print("La cadena invertida es ");
        for (int i = 1; i<= longitud; i++) {
           p.meter(cadena.charAt(i-1));
        }
        while (!p.vacia()) {
           aux = p.sacar();
           System.out.print(aux);
        }
     }
     catch(Exception e) {
        System.out.println("Error");
     }
  }
}
```

5.5 Introducción al Modelo-Vista-Controlador

Consideraciones básicas sobre Swing

La Java Foundation Classes (JFC) facilita herramientas que ayudan a los programadores de Java a tener una mejor construcción de sus *Graphics User Interface (GUI)*, las interfases gráficas con el usuario se realizaban mediante el Abstract Window Toolkit (**AWT**); hoy se ha generalizado el uso del paquete **Swing**, que sustituyó en parte a AWT, de manera que para todo componente AWT existe un componente `Swing` que lo reemplaza. Citemos un ejemplo: la clase `Button` de AWT es reemplazada por la clase `JButton de Swing`; se conservó el nombre `Button` y se le añadió la letra inicial J, de lo que resulta esa peculiar notación de camello. `Swing` es parte de JFC y casi en su totalidad hereda todo el manejo de eventos.

`Swing` abarca componentes como botones, tablas, marcos y otras facilidades que se incluyen en el paquete `javax.swing`. Los componentes `Swing` utilizan la infraestructura de AWT, reaccionan a eventos del teclado y el ratón, entre otros, por lo cual la mayoría de los programas `Swing` necesitan importar dos paquetes AWT: `java.awt.*` y `java.awt.event.*`.

Para poder realizar una aplicación con el uso de `Swing` se deben seguir las siguientes recomendaciones:

- Incluir al menos un contenedor de alto nivel (Top-Level Container-TLC), que facilita la aplicación de `Swing` para el dibujo y el manejo de eventos.
- Definir las componentes dependientes del contenedor de alto nivel. Estas pueden ser contenedores o componentes simples.

Las aplicaciones `Swing` tienen al menos un top-level container, que puede ser:

- `javax.swing.JFrame`: ventana independiente.
- `javax.swing.JApplet`: applet.
- `Diálogos`: ventanas de interacción con el usuario.
  - `java.swing.JOptionPane`: ventana de diálogo.
  - `java.swing.JFileChooser`: ventana para elegir un archivo.
  - `java.swing.JColorChooser`: elección de color.

`Swing` facilita la modificación del aspecto que presenta una ventana. Esto se puede lograr mediante la configuración de lo que conocemos como **JavaLook & Feel** de una aplicación.

A nivel código, primero se colocan las líneas que se encargan de importar los paquetes correspondientes:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

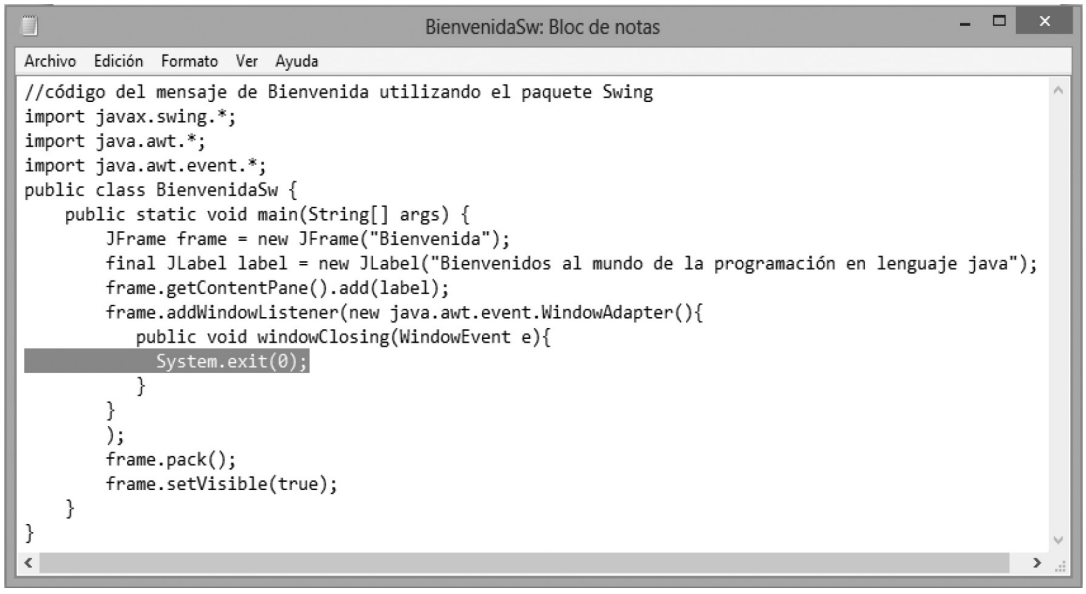
Las ventanas se codifican por lo común como un objeto de la clase JFrame y se asignan los atributos como bordes, título e íconos para minimizar, ampliar y cerrar la ventana. Las aplicaciones que utiliza GUI por lo regular acceden a un frame.

Después se declara una clase en el método **main** (en nuestro `casoBienvenidaSw`). Observe que se utilizó un `JFrame`, que es la clase para la creación de la ventana. `JLabel` permite definir un título para la ventana, la cual es agregada al frame a través de la instrucción:

```
frame.getContentPane().add(label);
```

El frame debe "empaquetarse" antes de mandar la instrucción de hacerlo visible (es invisible hasta que no se indique lo contrario). Para ello se emplean los métodos `pack` y `setVisible`, respectivamente.

Hasta aquí se ha tocado el tema de la presentación. Pero es necesario enlazar la parte visual (el `frame`) con los eventos que se generarán a través de la clase `addWindowListener`. Esta clase recibe como parámetro un objeto `WindowAdapter`, que entre otras funciones sirve para indicar lo que se hará cuando el usuario cierre la ventana. En nuestro caso —y en la mayoría de los casos— simplemente se da por terminada la aplicación mediante la instrucción `System.exit(0)`.



Programa 5.21 Creación de una ventana sencilla en Java con su ejecución.

Seamos honestos: para quienes hemos tenido la oportunidad de trabajar en ambientes integrados de desarrollo visual este método de trabajo resulta demasiado engorroso, además de que hay varias opciones y estilos de programación. En la práctica, se suele crear las ventanas a través de las ayudas de los ambientes de desarrollo o de herramientas automatizadas. Además, se está perdiendo campo de aplicación real en este tema ante la programación .NET para ambientes Windows y la tendencia hacia aplicaciones de internet, ya sea de páginas web tradicionales o de apps para celulares. El principal interés por el momento es que el estudiante tenga un primer acercamiento a las pantallas gráficas y, sobre todo, al enfoque Modelo-Vista-Controlador (MVC), retomado en gran parte por la programación profesional en Java.

Revisemos otro estilo de programar ventanas con los mismos componentes (véase el programa 5.22, adaptado de los autores Horstmann y Cornell).[11] Se han importado las mismas clases para el manejo visual, así como la utilización de una ventana a través de la clase `JFrame`, que también se hizo visible al utilizar el método `setVisible`. Sin embargo, en este caso se indicó de modo diferente el cierre de la ventana a través de la sentencia:

```
marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

donde marco es un objeto de la clase `JFramecon` valores en dos atributos: tiene como `Title` el valor "Marco sencillo" y como `Size` la dimensión máxima de la pantalla, obtenida a través de una cadena de instrucciones que involucraron tomar datos del sistema a través del método `getDefaultToolkit` de la clase `Toolkit`; aprovechar esos datos del sistema para guardar el número de pixeles con los cuales se está trabajando en la pantalla a través del método `getScreenSize()` de la clase Dimension y, por último, guardarlos en dos variables de tipo entero: **alturaPantalla** y **anchuraPantalla**, con los cuales se alimentó el atributo `size` de la ventana a través del método `setSize`. ¿Laborioso?

```
/*Programa 5.22 Segundo ejemplo de creación de una ventana sencilla en
Java.*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class PruebaMarcoSencillo {
   public static void main (String[] args) {
      MarcoSencillo marco = new MarcoSencillo();
      marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      marco.setVisible(true);
   }
}

class MarcoSencillo extends JFrame {
   public MarcoSencillo() {

      // se toma la resolución de la pantalla
      Toolkit kit = Toolkit.getDefaultToolkit();
      Dimension tamanoPantalla = kit.getScreenSize();
      int alturaPantalla = tamanoPantalla.height;
      int anchuraPantalla = tamanoPantalla.width;

      // se asignan los valores a la ventana
      setSize(anchuraPantalla,alturaPantalla);
      setTitle("Marco sencillo");
   }
}
```

[11] Horstmann y Cornell. *Java 2 Volumen I-Fundamentos*. Pearson, séptima edición, p. 362.

Por último, volveremos a adaptar un ejemplo de la misma fuente y los mismos autores,[12] de quienes reconocemos lo didáctico de sus materiales en este y otros temas. El programa crea tres botones: amarillo, azul y rojo. Al oprimir alguno de ellos, el color de la pantalla cambia. Primero observe que se crean los elementos gráficos en la forma en que hemos venido trabajando; de manera independiente se construye una clase que guarda las acciones a ejercer y después se ligan las acciones con los elementos gráficos que correspondan.

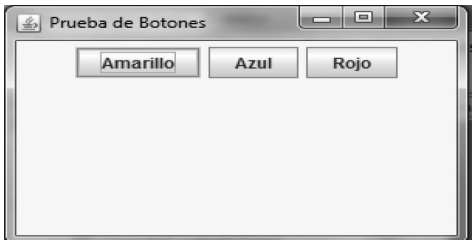
Ahora vayamos al detalle teniendo a la vista el programa 5.23:

- Las líneas 1 a 3 invocan las librerías necesarias de `AWT` y `Swing`.
- Las líneas 5 a 11 mandan llamar a la ventana indicando que al usuario cuando la cierre se dé por terminada la aplicación y, por supuesto, también la hace visible.
- Las líneas 13 a 23 dan a la ventana sus características de título y dimensiones. También le agregan una **lamina**, que es un objeto tipo `JPanel`.
- En las líneas 26 a 32 se crean tres botones con sus letreros respectivos y se incorporan al panel (que a su vez se incorpora a la ventana en la línea 20).
- Las líneas 43 a 51 implementan una acción a través de la cual se guarda el color de la ventana en el atributo **colorDeFondo** a través de su constructor. Como `ActionListener` es una interfase, es indispensable dar los detalles del método `actionPerformed`. En nuestro caso, cambiar el color de fondo de la pantalla. En las líneas 34 a 36 se crean tres acciones para los colores amarillo, rojo y azul.
- Ya tenemos tres botones y tres acciones. Lo que sigue es relacionarlos en las líneas 38 a 40. De esta forma, el botón amarillo queda ligado a `accionAmarillo` y cada vez que se oprima cambiará el color de la pantalla. Por supuesto, trabajarán bajo la misma lógica el botón rojo y el azul.

```
    Programa 5.23 Programa que puede cambiar el color de la pantalla
    utilizando el Modelo-Vista-Controlador, con su ejecución correspondiente.
1   import java.awt.*;
2   import java.awt.event.*;
3   import javax.swing.*;
4
5   public class PruebaBotones {
6      public static void main (String[] args) {
7         MarcoBotones marco = new MarcoBotones();
8         marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
9         marco.setVisible(true);
10     }
11  }
12
13  class MarcoBotones extends JFrame {
14     public static final int ANCHURA_PREFIJADA = 300;
15     public static final int ALTURA_PREFIJADA = 200;
16     public MarcoBotones() {
17        setTitle("Prueba de Botones");
18        setSize(ANCHURA_PREFIJADA, ALTURA_PREFIJADA);
19        LaminaBotones lamina = new LaminaBotones();
20        add(lamina);
21     }
22  }
23
```

[12] De hecho, recomendamos esta serie de libros de Java para quienes deseen tener una guía de referencia para el lenguaje: Horstmann y Cornell, *Java 2*, Pearson. Existen dos tomos que se venden de manera independiente (sugerimos comprar el segundo hasta haber aprovechado cabalmente el primero por tener un costo muy significativo). El tercero se refiere a Java Web, pero no fue publicado en español. La buena noticia es que puede obtenerse junto con otras obras de manera gratuita y legal del sitio de los autores: http://books.coreservlets.com/

```
24 class LaminaBotones extends JPanel {
25    public LaminaBotones() {
26       JButton botonAmarillo = new JButton("Amarillo");
27       JButton botonAzul = new JButton("Azul");
28       JButton botonRojo = new JButton("Rojo");
29
30       add(botonAmarillo);
31       add(botonAzul);
32       add(botonRojo);
33
34       AccionColor accionAmarillo = new AccionColor(Color.YELLOW);
35       AccionColor accionAzul = new AccionColor(Color.BLUE);
36       AccionColor accionRojo = new AccionColor(Color.RED);
37
38       botonAmarillo.addActionListener(accionAmarillo);
39       botonAzul.addActionListener(accionAzul);
40       botonRojo.addActionListener(accionRojo);
41       }
42
43    private class AccionColor implements ActionListener {
44       private Color colorDeFondo;
45       public AccionColor (Color c) {
46          colorDeFondo = c;
47       }
48       public void actionPerformed(ActionEvent evento) {
49          setBackground(colorDeFondo);
50       }
51    }
52 }
```



Ese es el enfoque del Modelo-Vista-Controlador: a) crear los elementos visuales, b) crear los elementos de acciones y proceso, y c) relacionarlos. Por eso pueden existir elementos gráficos que desembocan en la misma acción. Por otra parte, al hablar de una acción no nos referimos únicamente a algo visual. Las acciones pueden consistir en crear clases y procesar cálculos: todo lo que hemos trabajado a lo largo de estos dos capítulos (y aquellos temas que por razones de espacio tuvimos que dejar fuera).

EJERCICIOS SUGERIDOS

Es momento de tender el puente. Ya podemos leer datos del teclado y con un poco de paciencia también podemos construir ventanas. Ahora armemos clases que hagan la interacción con el usuario llamando a las ya construidas, pero respetando el enfoque: no queramos hacer todo en el mismo programa. De hecho, el primer ejercicio es realizar una clase que calcule el área y el perímetro de un triángulo a partir de sus tres lados (si ha realizado los ejercicios, ya programó esta clase en el tema de interfases). Diseñe una pantalla en modo texto que mande llamar a la clase Triangulo, aplicando robustez a través de excepciones. Diseñe otra pantalla en modo gráfico que mande llamar a la misma clase Triangulo.

¡Excelente! Dos entornos gráficos diferentes mandan llamar a la misma clase, que con anterioridad fue probada por TestTriangulo. Además, el código es portable y corre en cualquier plataforma.

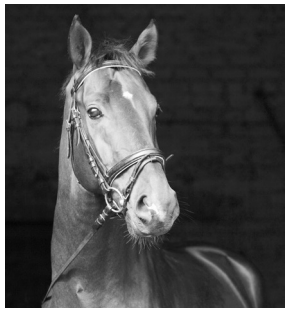
Mucho más laborioso que en lenguaje C, pero mucho más potente. Los lenguajes C, C++ y Java son tres opciones valiosas, igual que otras. ¿Cuál es mejor? Eso depende de nuestro contexto y necesidades.

La lección del oso y el dilema del caballo (ejercicio final del capítulo)

La maestra escribió en el pizarrón: "Ese oso hace eso".
Los estudiantes en su cuaderno escribieron: "Ese oso hace eso".
La maestra en el examen preguntó: ¿Qué hace ese oso?
Los alumnos (que estudiaron) respondieron: Eso hace ese oso.
Todos sacamos diez
y terminamos otro de nuestros cursos de educación superior.
Pero yo me quedé con dos dudas: ¿Qué hace ese oso? Y ¿cuál oso?

Roberto Durán

Compro un caballo en $600; lo vendo en $700; lo vuelvo a adquirir en $800 y lo vendo nuevamente en $900. ¿Cuánto gané?



El problema del caballo se planteó a un grupo de alumnos. Las respuestas iban desde que perdió $100 hasta que ganó $300. Se les pidió armar equipos de trabajo para argumentar y unificar respuestas y cada equipo respondió de manera diferente. La pregunta final —por demás lógica—suele ser: "Profesor, ¿cuál es la respuesta correcta?». Esperaríamos que el docente respondiera un monto, los exámenes se calificaran y se acabó el asunto, como diría mi abuela. Sin embargo, ¿el estudiante puede confiar en la respuesta del profesor?

De hecho, se planteó el mismo problema a un grupo de docentes de nivel superior y posgrado, y se obtuvieron prácticamente los mismos resultados. En definitiva, el docente no es infalible. Si aceptamos lo que dice sin mayor cuestionamiento y sin corroborar sus afirmaciones podríamos sacar 10 (como en la lección del oso) e incluso podríamos quedar bien con el jefe, pero nunca generaríamos conocimiento.

Tampoco los experimentos de los programadores son infalibles. Un estudiante tuvo la iniciativa de realizar el experimento con un grupo internacional de programadores y obtuvo el mismo resultado. ¡Qué elusivo resultó este caballo!

Muchas veces nos han dado a entender que la ciencia es un cúmulo de verdades que unos genios deducen en sus laboratorios y dejan plasmadas en libros incuestionables. ¡Cuán triste forma de visualizar la ciencia!

El pilar de la ciencia está en su forma de trabajo. Cuando se hace una afirmación, se publican las conclusiones y la forma en que se llegó a ellas. Otra persona puede reproducir el experimento, corroborar los datos, reinterpretar los resultados y rehacer la demostración, según sea el caso. Esa réplica también se hace pública. Al paso del tiempo los modelos que han podido ser confirmados quedan como un patrimonio colectivo, siempre expuestos a que en un futuro puedan refutarse.

Por eso nos permitimos dar un consejo final en este capítulo: ¡no crea nada de lo que le hemos dicho!

Mejor maticemos nuestra recomendación: tómelo por cierto en forma preliminar, pero verifíquelo con otros textos y, lo más importante, con su propia experiencia en el laboratorio: el equipo de cómputo. No es casual que le hayamos dado pistas acerca de lenguajes, compiladores y razonamientos. La ciencia es la búsqueda del conocimiento veraz y verificable acerca de la realidad.

Si descubre algún error, tendremos que admitirlo y corregirlo. El orgullo herido duele, pero en ningún docente-investigador puede pesar más que el amor a la verdad.

¡Por cierto! Se nos olvidaba la respuesta del caballo. Podemos decir: ganamos $x. Sin embargo, ¿por qué tendrían que creernos? Haga la siguiente prueba: comience con 10 billetes de $100 (reales o ficticios) y un caballo (de preferencia, ficticio). Reproduzca el problema y cuente al final su ganancia. Tendrá una respuesta corroborada por usted mismo... y eso es más útil que cualquier respuesta que quienes escribimos le podamos brindar.

Capítulo 6

Conexión a base de datos

6.1 Conceptos básicos de base de datos

Más allá del conocimiento técnico, el deber de un docente es motivar a la reflexión de sus estudiantes sobre el entorno en el cual se desenvuelve este conocimiento. Por ello, resulta indispensable aclarar que el problema del tratamiento de la información en una organización es un tema que está muy lejos de haber sido resuelto.

En primer lugar, ¿qué debe entenderse por información para una organización? El enfoque "tradicional" indica de manera implícita que se trabaja con respecto a las necesidades expresadas por un usuario con ayuda del ingeniero en computación sobre los aspectos técnicos: el experto en computación "traduce" los requerimientos del usuario a un sistema de información. No obstante, bajo este enfoque se pierde la oportunidad de que el experto en información ayude al usuario sobre los aspectos del manejo de la información; es decir, si hay otra posible información relevante que no se ha considerado u otros usuarios a quienes se deja de lado; además, si la presentación de resúmenes estadísticos es la más adecuada para los distintos usuarios. Prácticamente ya no hay escuelas que hagan hincapié en este enfoque, que diera pie a algunas carreras de informática en México y otros países. El aspecto de la automatización computacional (computación) ha dejado de lado al aspecto del tratamiento de la información (informática), siendo que deberían ser dos pilares del profesional de computación, sistemas y/o informática. ¡Es una lástima! En la medicina antes se le dejaba todo al médico, en la actualidad se reconoce que el paciente debe ser copartícipe activo en la búsqueda de su salud; en computación en algunas ocasiones se hace lo contrario: se le deja al usuario establecer los requerimientos, cuando el profesional en informática tendría que realizar junto al propio usuario un análisis crítico de las posibilidades sobre el tratamiento de la información.

En segundo lugar, existen al menos dos tipos de información. Una pertenece a información controlada, no redundante, con cortes de tiempo delimitados y de alcance fijo. Ese es el campo de los sistemas manejadores de bases de datos (SMBD), desde los pequeños como Access hasta los mejores y de alto costo como Oracle. La otra es información dinámica y prácticamente ilimitada, de alcances dispersos, con diferentes grados de confiabilidad, profundidad y exactitud, una gran parte disponible en la web. Se han dado diferentes enfoques para su organización y clasificación: buscadores como Google, bancos de información de creación colectiva y propósito general como Wikipedia, bancos de información de propósito específico y creación colectiva como Megasinapsis y redes sociales como Facebook, entre otras. Es el tema de Big Data, que apenas comienza su andar.

En tercer lugar, ¿quiénes explotarán la información y dónde estarán ubicados? ¿Se desea que un grupo restringido de personas tenga acceso o toda la humanidad? ¿Existirá información de acceso controlado? Ese tipo de preguntas da pie a establecer el hardware de almacenamiento: una computadora personal, una red local o una Intranet (también existe la posibilidad de almacenarlo «en la nube», si se emplea algún servicio gratuito o pagado proporcionado por un proveedor). La definición de requerimientos establece de manera implícita o explícita aspectos de seguridad y diferentes niveles de permisos de acceso a la información, así como alternativas de la estrategia de solución planteada. Por citar dos ejemplos: los archivos de texto o binario son útiles para sistemas monousuario y bajos volúmenes de datos (o una situación parecida), mientras que para pocos usuarios que desean acceder desde internet bastarían un SMBD como Access combinado con un lenguaje de programación como PHP o Java web en su versión más sencilla (JSP).

En este capítulo el estudiante podrá tener un primer acercamiento a SQL básico, el lenguaje de programación más empleado para SMBD relacionales (para información controlada). Se dejarán de lado por el momento aspectos de análisis de información, explotación avanzada de SMBD, la BIG DATA y seguridad. Después se dará un ejemplo de cómo establecer un "puente" entre la base de datos y los lenguajes de programación, en particular Java.

Si nuestro campo es tan delimitado, ¿por qué hemos dedicado estos párrafos a un panorama tan general? Justo por ello: para que el estudiante perciba que es solo el comienzo de un campo amplísimo, pues antes de hundirse en los detalles de los árboles, conviene dar un vistazo global al bosque entero.

Concepto de base de datos

Una base de datos es "una colección de información que existe durante un periodo largo… con la expresión base de datos se designa una colección de datos administrada por un sistema de gestión de bases de datos o sistema manejador de base de datos, que se abrevia DBMS o SMBD (Data Base Management System o Sistema de Administración de Bases de Datos).

Se espera que este sistema.

1. Permita a los usuarios crear otras bases de datos y especificar su esquema (estructura lógica de los datos) por medio de un lenguaje especializado denominado lenguaje de definición de datos.
2. Ofrezca a los usuarios la capacidad de consultar los datos (una consulta es un tecnicismo de base de datos que formula una pregunta sobre los datos) y modificarlos, para lo cual usará un lenguaje apropiado, llamado a menudo lenguaje de consulta o lenguaje de manipulación de datos.
3. Soporte el almacenamiento de cantidades muy voluminosas de datos durante un largo periodo, protegiéndolos contra accidentes o utilización no autorizada y permitir el acceso eficiente para hacer consultas y modificar la base de datos.
4. Controle el acceso simultáneo a los datos por parte de muchos usuarios, sin permitir que las acciones de uno de ellos afecte a los otros ni que los accesos simultáneos corrompan los datos por accidente."[1]

Concepto cliente/servidor

Comencemos con la definición:

"Client/server (cliente/servidor). Modelo de diseño para aplicaciones que corren en redes, donde la mayor parte del procesamiento en segundo plano (por ejemplo, realizar una búsqueda física en una base de datos) se lleva a cabo en un servidor. El procesamiento en primer plano, que implica comunicación con el usuario, lo manejan programas más pequeños que se encuentran distribuidos en las estaciones de trabajo clientes."[2]

Por lo regular, los componentes cliente y servidor de una aplicación se ejecutan en diferentes equipos que se comunican entre ellos mediante una red (aunque es posible trabajar el concepto en una sola PC). A la hora de conversar, los clientes y los servidores de una red deben emplear todos software de comunicaciones que les permita "hablar el mismo idioma".

En un sistema **cliente/servidor** de dos capas se da la siguiente distribución típica de tareas:

a) "El cliente es la presentación de la aplicación que se usará para realizar el trabajo… Suele encargarse de los siguientes tipos de operaciones:
   - Presentar una interfaz de usuario con la que pueda interactuar el usuario; por ejemplo, un formulario para introducir datos.
   - Validar la entrada de datos; por ejemplo, comprobar que el usuario introduce una fecha válida en un campo fecha.
   - Pedir información desde un servidor de base de datos, como los registros de cliente o las peticiones de ventas.
   - Procesar la información que devuelve un servidor de base de datos; por ejemplo, rellenar un formulario con datos, calcular campos totales en un informe o crear gráficos y diagramas".

[1] Ullman, Jeffrey D. y Widow, Jennifer. *Introducción a los sistemas de bases de datos*. Pearson, México, 1999, pp. 1 y 2.
[2] Pfaffenberger, Bryan. *Diccionario de términos de computación*. Pearson, México, 1999.

b) "El servidor es donde se ejecuta la aplicación. En segundo plano, un servidor de base de datos funciona para administrar una base de datos entre todos los usuarios y las aplicaciones que la usan para almacenar y recuperar datos. El servidor es el responsable de las siguientes operaciones:
   - Abrir una base de datos y hacerla accesible a las aplicaciones.
   - Evitar accesos no autorizados a la base de datos mediante estrictos controles de seguridad.
   - Evitar la interferencia destructiva entre transacciones concurrentes que acceden a los mismos conjuntos de datos.
   - Proteger una base de datos con características de copia de seguridad y recuperación indiscutida.
   - Mantener la integridad de los datos y la consistencia a medida que los usuarios realizan su trabajo."[3]

Los **sistemas cliente/servidor de tres capas** se presentan por lo común en aplicaciones web:

c) La primera capa está representada por el navegador, el cual interpreta el código HTML y el Java Script. Los navegadores más comunes son Internet Explorer, Chrome y Mozilla.

d) En la capa intermedia se encuentra el servidor de servicios de internet. Los más comunes son Internet Information Server (para ASP) y Apache/Tomcat (para PHP y JSP). Ellos interpretan el lenguaje de programación que recibe los datos de la base de datos y los programas manejan los elementos básicos de un lenguaje de programación. Con el aprovechamiento de estos elementos se solicitan datos a la base de datos a través de cadenas SQL y se reciben los datos de respuesta. Entre los más comunes: ASP, PHP y JSP.

e) Por último, la tercera capa es el sistema manejador de base de datos (SMBD), que ejecuta las sentencias SQL solicitadas. Algunos SMBD —entre ellos Oracle, MySQL y SQL Server— permiten la programación de rutinas dentro de ellos mismos si se utiliza un lenguaje de programación propio. Nos referimos a las vistas, transacciones, procedimientos almacenados y disparadores.

La figura 6.1 trata de sintetizar un sistema cliente/servidor de tres capas.

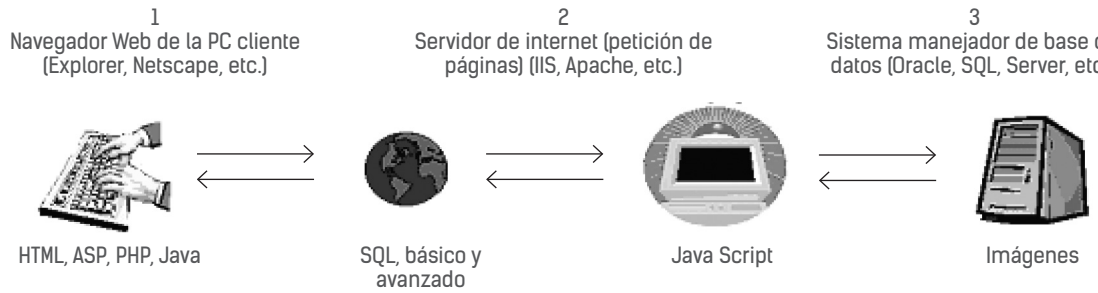


Figura 6.1 Sistema cliente-servidor de tres capas.

Para explotar de manera adecuada los SMBD y la programación para web es necesario entender el concepto cliente/servidor y ubicar los elementos que trabajan en cada capa, con sus potencialidades y limitaciones.

Si se repasan las definiciones básicas nos daremos cuenta de que una base de datos intenta facilitar el almacenamiento y la recuperación por múltiples usuarios de información controlada y no redundante, que se encuentra en la capa del servidor en un esquema cliente/servidor. La base de datos es controlada por un sistema manejador de base de datos (en este caso de tipo relacional: MySQL) que utilizará como una herramienta fundamental el lenguaje de programación SQL. En este caso, se empleará un esquema de dos capas (Java y la base de datos), lo cual implica aprender SQL básico y después hacer un "puente" entre Java y la base de datos.

[3] Bobrowski, Steve. *Oracle 8i para Windows NT, edición de aprendizaje.* McGraw-Hill Osborne, España, 2000, pp. 37-38.

Conviene aclarar que el SQL básico es un lenguaje independiente por completo de Java y puede establecerse un enlace también con otros lenguajes como Visual.NET o PHP. Por otra parte, los elementos básicos para establecer el "puente" son muy parecidos en los distintos casos, aunque cada uno emplea drivers y configuraciones específicas.

Acercamiento intuitivo a conceptos básicos sobre base de datos

Suponga que se requiere en forma cotidiana alguna información de los estudiantes: boleta, nombre, carrera y nivel de la carrera. Si los datos son pocos y es para uso personal, bastaría que se tuviera un listado en una hoja de cálculo, como lo muestra la figura 6.2. No obstante, suele haber un problema muy común en estos casos: si no se tiene mucho cuidado, una inconsistencia puede hacer que los filtros no funcionen de manera adecuada. Tal vez ya observó que en la descripción del nivel se empleó "Nivel superior" y "Superior" para referirse al mismo nivel de estudios: licenciatura. Por ello para la hoja de cálculo serán niveles distintos, cuando en realidad para el usuario es el mismo. Es posible que haya pasado desapercibido otro caso menos notorio: en una "LIC. EN INFORMATICA" se puso espacio después de "LIC.", mientras que en otra se omitió. De manera similar al caso anterior, para la computadora son dos carreras distintas.

ALUMNOS

clave	paterno	materno	nombre_de_pi	descripcion carrera	descripcion nive
0083147888	LOPEZ	LUNA	MIGUEL	LIC.EN INFORMATICA	Nivel superior
2004101234	LOPEZ	LOPEZ	AMEYALLI	INGENIERA EN SISTEMAS COMPUTACIONALES	Superior
2003569030	HUERTA	HERNANDEZ	CLAUDIA AIDEE	LIC. EN INFORMATICA	Nivel superior
2003688722	PEREZ	DE ALBA	LUIS ANTONIO	TECNICO EN PROGRAMACION	Nivel medio superior

Figura 6.2 Manejo cotidiano de información en hoja de cálculo.

Para evitar este tipo de inconsistencias se suelen crear catálogos. De esta forma se dan claves en lugar de los nombres completos y se localizan los datos con la función BUSCARV de la hoja de cálculo; cada catálogo se coloca en una hoja distinta. Los datos que deben estar catalogados serán aquellos que servirán para clasificar información y obtener estadísticas (en este caso, descripción de la carrera y nivel de estudios). El resultado es similar a la figura 6.3.

	A	B	C	D	E	F	G	H	I
1									
2		**ALUMNOS**							
3									
4		clave	patern	materno	nombre_de_pi	c_carrera	descripcion carrera	c_nivel	descripcion niv
5		0083147888	LOPEZ	LUNA	MIGUEL	001	LICENCIATURA EN INFORMATICA	01	Nivel superior
6		2004101234	LOPEZ	LOPEZ	AMEYALLI	002	INGENIERA EN SISTEMAS COMPUTACIONALES	01	Nivel superior
7		2003569030	HUERTA	HERNANDEZ	CLAUDIA AIDEE	001	LICENCIATURA EN INFORMATICA	01	Nivel superior
8		2003688722	PEREZ	DE ALBA	LUIS ANTONIO	003	TECNICO EN PROGRAMACION	02	Nivel medio superior
9									
10									

nivel | carreras | **alumnos** | alumnos 2

Figura 6.3 Manejo de información en hoja de cálculo con el uso de catálogos.

Con estos breves elementos ya es posible acercarse a algunos conceptos de las bases de datos: las filas o renglones de Excel son los registros de la base de datos, que deben ser identificados por un dato de manera unívoca; a este dato se le conoce como llave primaria o clave primaria y es irrepetible (como el número de placas de un coche); puede darse el caso que una llave primaria esté compuesta por más de un campo, en cuyo caso la combinación de estos será irrepetible. Las columnas son los campos y cada hoja corresponde a una tabla diferente. Quizá ya observó que la hoja de cálculo marca un error cuando se da una clave inexistente, con lo que se evita una inconsistencia en la información; a esta propiedad se le conoce como integridad referencial. Al archivo completo se le da el nombre de base de datos.

Es importante aclarar que los registros y campos deben estar diseñados de manera que eviten inconsistencias; es decir, no es permitido que una información aparezca doble vez en el sistema y con ello se dé pie a que existan datos distintos (por ejemplo: que el nombre de la persona sea diferente según la tabla que se consulte). Para lograr esta meta se realiza un proceso de normalización de la base de datos. Lamentablemente, por cuestiones de espacio, el proceso de normalización queda fuera del alcance de este libro.

REFLEXIÓN 6.1

Cuidemos las labores de análisis al diseñar la base de datos

Hemos observado muchos trabajos finales semestrales, tesis en proceso de revisión final e incluso sistemas a punto de entregarse con errores delicados en el diseño de la base de datos: la longitud de los campos no corresponde a la información real que manejará el usuario; no existe consistencia entre los requerimientos del usuario expresados en los casos de uso o alguna técnica equivalente y los campos de la base de datos; o hay redundancia en la información de las tablas.

El proceso de diseño de bases de datos queda fuera de los límites de este libro, pero juzgamos necesario resaltar este tema.

Si el mantenimiento de la información se diera por una sola persona y hubiera muchas que desearan consultarlo al cierre del día y de la semana, bastaría con poner el archivo con formato de solo lectura en un directorio común. Sin embargo, ¿qué pasaría si varios usuarios desean modificar la información al mismo tiempo, se trabaja con cientos de miles de registros y se requiere la información actualizada al instante (en línea)? Ese es el campo de los sistemas manejadores de bases de datos.

6.2 Manejo de SQL básico

Concepto de SQL

SQL (Structured Query Language) es el lenguaje que se usa en los sistemas manejadores de bases de datos relacionales. En la actualidad lo manejan todos los SMBD con presencia significativa en el mercado. Su estructura de comandos es relativamente simple. Sus finalidades son:

a) Recuperar, introducir, actualizar y eliminar los datos de la base de datos.
b) Crear, alterar y colocar objetos de la base de datos.
c) Restringir el acceso a los datos de la base de datos y a las operaciones del sistema.

En muchos casos, la única forma de que una aplicación pueda interactuar con un servidor de base de datos es al ejecutar un comando SQL, como en Oracle. Existen otras formas de modificar datos sin pasar por sentencias SQL. Hay componentes de programación visual que vinculan un formulario con tablas de una base de datos (Visual Basic, Delphi, etc.). El ahorro en tiempo de programación es notable, pero hay que tener cuidado especial para módulos que requieren mucha flexibilidad, eficiencia o seguridad. En algunos SMBD, como SQL Server, muchas de las tareas de SQL también pueden realizarse mediante interfases gráficas proporcionadas por el propio SMBD. Son más rápidas de manejar y más intuitivas. La desventaja es que pertenecen a un único SMBD.

En resumen, las sentencias SQL son la forma más general y flexible de acceder a una base de datos relacional.

Las cuatro categorías principales de comandos SQL son:

a) Los comandos de manipulación de datos o lenguaje de modificación de datos (DML, *data modification language*) recuperan, insertan, actualizan y eliminan las filas de una tabla en una base de datos. Los cuatro comandos DML básicos son SELECT, INSERT, UPDATE y DELETE.
b) Las aplicaciones que usan el lenguaje SQL y las bases de datos relacionales realizan el trabajo si se usan transacciones. Una transacción de base de datos es una unidad de trabajo que realiza una o más sentencias SQL relacionadas entre sí. Para preservar la integridad de la información en una base de datos, las bases de datos relacionales aseguran que todo el trabajo dentro de cada transacción se confirme o restaure. Una aplicación usa los comandos SQL de control de transacción COMMIT y ROLLBACK para controlar el resultado de una transacción de base de datos.
c) Los comandos del lenguaje de definición de datos (DDL, *data definition language*) crean, alteran y eliminan objetos de la base de datos. La mayoría de los objetos de bases de datos tienen los comandos CREATE, ALTER y DROP.
d) Una aplicación administrativa usa comandos del lenguaje de control de datos (DCL, *data control language*) para controlar el acceso de usuario a una base de datos. Los tres comandos DCL que más se usan son GRANT, REVOKE y SET ROLE.

Hay convenciones estándar de SQL, como el ANSI/ISO y el SQL3. Si una aplicación solo usa comandos estándar SQL, se dice que es portable. En otras palabras, si decide sustituir la base de datos por otra que soporte el mismo estándar, la aplicación aún funciona sin modificaciones. En caso contrario, será necesario modificar y volver a compilar la aplicación antes de poder trabajar con otros sistemas de bases de datos. En los casos en que se permitan accesos por formas nativas y por SQL, en general es preferible SQL para dar mayor portabilidad. Los tiempos de programación en general son similares; la eficiencia de SQL en la mayoría de los casos es mejor.

En la ejecución de las sentencias SQL no influye la alineación, el número de líneas ni el uso de mayúsculas y minúsculas. Sin embargo, se recomienda fijar reglas para hacer más fácil la lectura del código.

Instalación de MySQL

Existen varios sistemas manejadores de base de datos (SMBD) en el mercado, desde pequeños como Access hasta grandes como Oracle, pasando por sistemas medianos como MySQL y SQL Server. Algunos de ellos tienen versiones gratuitas limitadas —como el propio MySQL—, aunque suficientes para aprender los fundamentos de SQL.

La versión 5.6.16 de Community Server de MySQL puede descargarse de manera gratuita del sitio http://dev.mysql.com/downloads/file.php?id=450946,[4] con lo cual se obtiene un archivo de más 250 Mb en el caso de la versión para 32 bits para Windows. Se mantuvieron las opciones por omisión y no se tuvo ninguna dificultad en la instalación (véase figura 6.4). La única precaución fue conservar la clave de acceso del administrador, de preferencia al menos de nivel medio de seguridad (en nuestro caso: eslmqceRD2304).[5] Ya realizada la instalación, esa clave o la de algún otro usuario que se haya agregado debe proporcionarse cuando se ejecute el programa de MySQL (véase figura 6.5).

[4] Información actualizada a febrero de 2014.
[5] Para crear esta clave de acceso se recurrió a un truco nemotécnico: la canción "estas son las mañanitas que cantaba el Rey David"…, combinada con el 23 de abril: día del libro. Este tipo de "trucos" pueden ayudar a crear claves de seguridad media o alta fáciles de recordar.

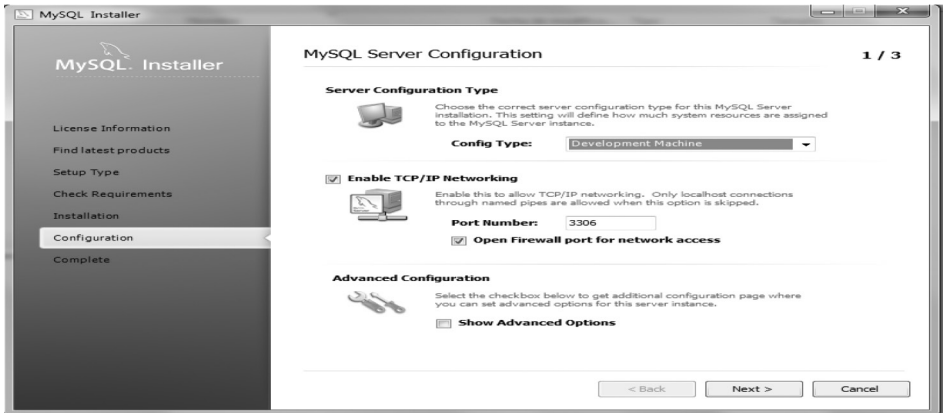


Figura 6.4 Primera pantalla de instalación de MySQL.

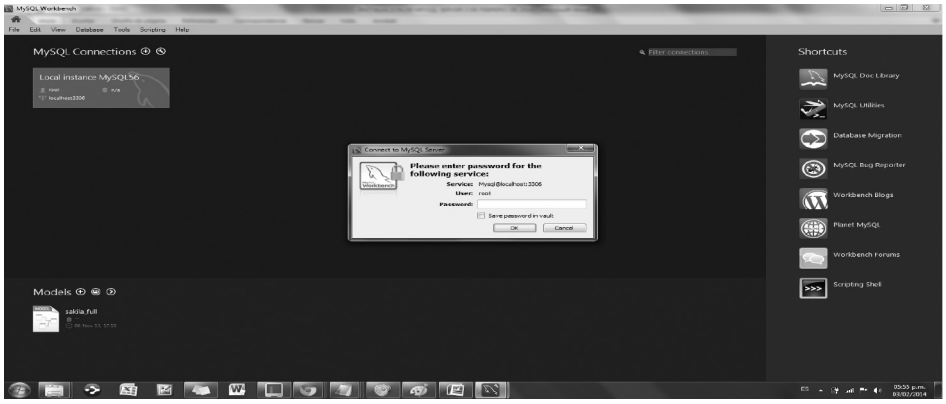


Figura 6.5 Pantalla de acceso cada vez que se abre MySQL.

A partir de este momento ya es posible trabajar con las diversas sentencias SQL que se necesiten. Lo primero, por supuesto, es crear la base datos (nosotros ocuparemos la base de datos llamada test que ya viene predefinida en el sistema). En segundo lugar, es necesario crear las tablas.

Antes de continuar creemos conveniente aclarar que los códigos aquí citados fueron probados en **SQL Server 2000**, **Access2002**, **Oracle 8iLite** y **MySQL Workbench 6.0 CE**, con una portabilidad cercana a 100%. Se indica en forma expresa cuando algún código tiene problemas con alguna o algunas de las plataformas mencionadas.

Para elaborar los ejemplos se usarán las siguientes tablas, con su contenido respectivo (la A indica que se trata de un dato alfanumérico).

ALUMNOS

Clave	A 10	llave principal
Paterno	A 30	
Materno	A 30	
Nombre_de_pila	A 30	
C_carrera	A 3	

CARRERAS

Clave	A 3	llave principal
Descripcion	A 40	
C_nivel	A 2	

NIVEL

Clave	A 2	llave principal
Descripcion	A 40	

ALUMNOS

Clave	Paterno	Materno	Nombre_de_pila	C_carrera
0083147888	LOPEZ	LUNA	MIGUEL	001
2004101234	LOPEZ	LOPEZ	AMEYALLI	002
2003569030	HUERTA	HERNANDEZ	CLAUDIA AIDEE	001
2003688722	PEREZ	DE ALBA	LUIS ANTONIO	003

CARRERAS

Clave	Descripción	C_nivel
001	LICENCIATURA EN INFORMATICA	01
002	INGENIERIA EN SISTEMAS COMPUTACIONALES	01
003	TECNICO EN PROGRAMACION	02

NIVEL

Clave	Descripción
01	Nivel superior
02	Nivel medio superior

Creación de tablas

Para crear tablas se utiliza la instrucción CREATE TABLE. Esta instrucción tiene la siguiente estructura (los corchetes indican información opcional).

```
CREATE TABLE nombre_de_la_tabla (
nombre_del_campo tipo_de_campo [otras características del campo],
nombre_del_campo tipo_de_campo [otras características del campo],
...
[características generales de la tabla]
)
```

Las características de los campos se colocan de manera inmediata después de que el campo fue declarado, mientras que las características a nivel tabla se señalan al final de la lista de campos. A nivel campo se puede declarar que el campo no puede llevar valores nulos mediante la cláusula NOT NULL. O bien, que los valores no pueden repetirse a través de UNIQUE. También es posible señalar que ese dato requiere forzosamente estar en una tabla externa para permitir que el registro se inserte (en otras palabras: que debe haber integridad referencial); para ello se emplea la palabra REFERENCES. Para señalar que ese campo será la llave primaria se utiliza PRIMARY KEY. A nivel tabla se puede indicar que uno o varios campos forman la llave principal mediante la instrucción,

```
PRIMARY KEY ( lista de campos )
```

o bien, que un campo o varios campos tienen integridad referencial. Para ello se usa:

```
FOREIGN KEY (lista de campos) REFERENCES tabla externa (campo externo)
```

Las tablas que sirven como ejemplo se crearon con los siguientes comandos. El orden de creación y llenado debe seguirse al pie de la letra por la integridad referencial.

```
CREATE TABLE nivel (
   clave varchar (2) NOT NULL,
   descripcion varchar(40) NOT NULL,
   PRIMARY KEY (clave)
)

CREATE TABLE carreras (
   clave varchar (3) NOT NULL,
   descripcion varchar(40) NOT NULL,
   c_nivel varchar(2) NOT NULL,
   PRIMARY KEY (clave),
   FOREIGN KEY (c_nivel) REFERENCES nivel (clave)
)

CREATE TABLE alumnos (
   clave varchar (10) NOT NULL ,
   paterno varchar (30) ,
   materno varchar (30) ,
   nombre_de_pila varchar (30) NOT NULL ,
   c_carrera varchar (3) NOT NULL ,
   PRIMARY KEY (clave),
   FOREIGN KEY (c_carrera) REFERENCES carreras (clave)
)
```

Para añadir datos a una tabla se emplea `ALTER TABLE`. Para modificar, `ALTER TABLE <nombre de la tabla> ALTER COLUMN <nombre del campo>` o `MODIFY <nombre del campo>`.[6] Para eliminar, `DROP`. Las siguientes instrucciones añaden un campo Sexo a la tabla Alumnos, modifican su tamaño y después lo eliminan (esto no tiene mucho sentido, pero nos da una idea clara de la sintaxis de los comandos).

```
ALTER TABLE alumnos ADD sexo varchar(1)
ALTER TABLE alumnos MODIFY sexo varchar(10)
ALTER TABLE alumnos DROP COLUMN sexo
```

Inserción de datos de la tabla

Para insertar se utiliza el comando `INSERT`. Este comando puede tener cualquiera de dos vertientes: `INSERT INTO tabla (campos) VALUES (valores)`, o bien, `INSERT INTO tabla (campos) SELECT campos FROM tabla WHERE condición`. Para ambos casos los valores que se indican deben coincidir en número y orden con los campos señalados. Aunque se puede omitir la mención de los campos cuando los valores indicados corresponden a todos los campos de la tabla y se proporcionan en orden. Sin embargo, no es aconsejable; cualquier cambio en la tabla haría que esta correspondencia exacta ya no se cumpliera.

[6] Para el caso del SMBD Oracle y MySQL se debe usar MODIFY, mientras que Access y SQL Server se ejecutan con ALTER COLUMN.

Es responsabilidad del programador validar que no se intente duplicar la llave principal. Si se intenta insertar un registro con llave principal duplicada, el sistema manejador de base de datos detectará la anomalía, mandará un mensaje de error y detendrá la ejecución.

En nuestro caso se establecieron integridades referenciales y por eso es necesario llenar la tabla `NIVEL`, para después agregar los datos a la tabla `CARRERAS` y terminar con la tabla `ALUMNOS`. Para el llenado de la tabla `NIVEL` presentada como ejemplo, la sentencia `INSERT INTO` correspondiente sería la siguiente:

Consultas con una sola tabla

Una consulta es una sentencia SQL que usa el comando `SELECT` para recuperar información de una base de datos. El conjunto resultado de una consulta es el conjunto de filas y columnas que la consulta pidió al servidor de base de datos (el asterisco indica que se recuperen todos los campos). La cláusula `SELECT` tiene la estructura siguiente:

```
SELECT lista_de_campos
FROM lista_de_tablas
WHERE lista_de_condiciones
```

La sentencia `SELECT * FROM` alumnos tendrá el siguiente resultado:

Clave	Paterno	Materno	Nombre_de_pila	C_carrera
0083147888	LOPEZ	LUNA	MIGUEL	001
2004101234	LOPEZ	LOPEZ	AMEYALLI	002
2003569030	HUERTA	HERNANDEZ	CLAUDIA AIDEE	001
2003688722	PEREZ	DE ALBA	LUIS ANTONIO	003

Mientras que `SELECT clave, paterno, materno, nombre_de_pila FROM alumnos` dará la siguiente tabla:

Clave	Paterno	Materno	Nombre_de_pila
0083147888	LOPEZ	LUNA	MIGUEL
2004101234	LOPEZ	LOPEZ	AMEYALLI
2003569030	HUERTA	HERNANDEZ	CLAUDIA AIDEE
2003688722	PEREZ	DE ALBA	LUIS ANTONIO

También se pueden utilizar operadores diversos, como concatenar cadenas -||- y el alias —AS—, que permite poner sobrenombres a campos y tablas. Por ejemplo:

```
SELECT clave, concat(paterno,' ',materno)
nombreCompleto
FROM alumnos
ORDER BY paterno, materno, nombre_de_pila
```

dará como resultado:[7]

[7] Esta instrucción en Access y SQL Server se coloca SELECT paterno + ' ' + materno + ' ' + nombre_de_pila AS nombre FROM alumnos; mientras que en Oracle el + se sustituye con ||. La sintaxis mencionada directamente en el libro corresponde a MySQL.

Nombre
LOPEZ LUNA MIGUEL
LOPEZ LOPEZ AMEYALLI
HUERTA HERNANDEZ CLAUDIA AIDEE
PEREZ DE ALBA LUIS ANTONIO

Otros operadores son: + (más), – (menos), * (multiplicación), / (división), UPPER (conversión a mayúsculas), LOWER (conversión a minúsculas), SQRT (raíz cuadrada) y LN (logaritmo natural). Una función grupo (o agregada) devuelve un valor agregado para todas las filas que forman parte del conjunto resultado de una consulta.

```
SELECT count(Clave) AS no_candidatos FROM alumnos
```

dará como resultado:

No_candidatos
4

Otras funciones agregadas son: COUNT para cuenta; MAX para máximo; MIN para mínimo y AVG para promedio.

Es recomendable utilizar la palabra DISTINCT en los casos que no deba repetirse el valor de una columna. Supóngase que se desea saber las claves de las carreras que tienen alumnos inscritos. La siguiente instrucción cumple ese propósito:

```
SELECT DISTINCT(c_carrera) FROM alumnos
```

Para poner condiciones se utiliza la palabra WHERE.

```
SELECT * FROM alumnos
WHERE c_carrera = '001'
```

Clave	Paterno	Materno	Nombre_de_pila	C_carrera
0083147888	LOPEZ	LUNA	MIGUEL	001
2003569030	HUERTA	HERNANDEZ	CLAUDIA AIDEE	001

Pueden utilizarse operadores relacionales: x = y, x < > y, x > y, x < y, x IN, x NOT IN, x BETWEEN y AND z, EXIST, x LIKE y, x IS NULL, x IS NOT NULL. Para el caso de LIKE existe el operador %, que puede colocarse antes del valor, después o en ambos lugares. Por ejemplo: nombre_de_pila like '%MIGUEL%' seleccionará a todos los valores coincidentes con MIGUEL en el nombre de pila, independientemente que existan caracteres antes o después de esa coincidencia (Vg. "LUIS MIGUEL" y "MIGUEL ANGEL" cumplen el criterio). El comando quedaría de la siguiente manera: `SELECT * FROM alumnos WHERE nombre_de_pila LIKE '%MIGUEL%'`.

Muchas consultas necesitan contestar cuestiones basadas en agregados o resúmenes de información, en vez de proporcionar los detalles de registros individuales de una tabla. Para agrupar datos en el resultado de una consulta esta debe incluir una cláusula GROUP BY. Una cláusula GROUP BY puede listar cualquier nombre de columna de las tablas que aparecen en la cláusula FROM de la consulta, con excepción de los alias definidos. Cuando se genera la cláusula SELECT de una consulta que incluya GROUP BY se

puede incluir una expresión que use una función de grupo. Por ejemplo, AVG, COUNT, MAX, MIN, entre otras. Para eliminar los grupos seleccionados del resultado de una consulta se puede agregar una cláusula HAVING a la cláusula GROUP BY de la consulta, prácticamente igual que la condición de cláusula WHERE.

```
SELECT c_carrera AS carrera, COUNT(c_carrera) AS frecuencia
FROM alumnos
GROUP BY (c_carrera)
HAVING COUNT(c_carrera) >= 2
```

dará como resultado

Carrera	Frecuencia
001	2

Para ordenar el resultado se puede emplear ORDER BY, en orden ascendente (ASC) o descendente (DESC). Es necesario indicarlo cuando se requiera, pues los SMBD no tienen un orden definido para regresar los registros de las tablas; ni el orden de captura ni la llave principal. Siempre que se desee un determinado orden se deberá emplear la expresión ORDER BY. Por ejemplo:

```
SELECT clave, paterno, materno, nombre_de_pila
FROM alumnos
ORDER BY paterno, materno, nombre_de_pila ASC
```

Consultas con más de una tabla

Para unir dos tablas es necesario que exista un campo común que sea del mismo tipo y se refiera al mismo dato; el nombre puede ser diferente (técnicamente pueden referirse a campos con significado distinto, pero la consulta carecería de sentido). SQL devolverá un conjunto resultado que es el producto de las dos tablas al descartar las filas que no cumplan con las condiciones indicadas. Esas condiciones se señalan con WHERE campo1 = campo2, donde campo1 y campo2 son los campos que ligan a las tablas (puede existir más de una condición). Si no se colocara ninguna condición, se devolverán todas las filas del producto; en casos especiales, puede ser que en verdad deseemos esta combinación, pero en la mayoría de ocasiones suele ser un error. Por ejemplo, a partir de los siguientes datos:

CARRERAS		
Clave	Descripción	C_nivel
001	LICENCIATURA EN INFORMATICA	01
002	INGENIERIA EN SISTEMAS COMPUTACIONALES	01
003	TECNICO EN PROGRAMACION	02

NIVEL	
Clave	Descripción
01	nivel medio superior
02	nivel superior

la consulta:

```
SELECT carreras.clave, carreras.descripcion, c_nivel, nivel.descripcion as
nivelFROM carreras, nivelWHERE c_nivel = nivel.clave
```

obtendrá:

CARRERAS			
Clave	Descripción	C_nivel	Nivel
001	LICENCIATURA EN INFORMATICA	01	nivel superior
002	INGENIERIA EN SISTEMAS COMPUTACIONALES	01	nivel superior
003	TECNICO EN PROGRAMACION	02	nivel medio superior

Pero si no se hubiera señalado ninguna condición se obtendría:

CARRERAS			
Clave	Descripción	C_nivel	Nivel
001	LICENCIATURA EN INFORMATICA	01	nivel superior
001	LICENCIATURA EN INFORMATICA	01	nivel medio superior
002	INGENIERIA EN SISTEMAS COMPUTACIONALES	01	nivel superior
002	INGENIERIA EN SISTEMAS COMPUTACIONALES	01	nivel medio superior
003	TECNICO EN PROGRAMACION	02	nivel superior
003	TECNICO EN PROGRAMACION	02	nivel medio superior

Como se habrá notado, los campos pueden llevar un sobrenombre, el cual aparecerá en el resultado. También las tablas pueden llevar un sobrenombre. Cuando existen campos que se llaman igual, como sucedió con la descripción en el comando anterior, puede colocarse el nombre de la tabla y el campo para evitar la ambigüedad. Otra opción es poner alias a las tablas y usarlos como si fueran el nombre de las tablas. La instrucción anterior es equivalente a la siguiente:[8]

```
SELECT car.clave, car.descripcion, c_nivel, niv.descripcion AS nivel
FROM carreras car, nivel niv
WHERE c_nivel = niv.clave
```

SQL permite recursividad y consultas anidadas. Supóngase que se requiere obtener la clave de las carreras que no tienen alumnos inscritos. La siguiente instrucción cubrirá ese requisito:

```
SELECT clave
FROM carreras
WHERE clave NOT IN (SELECT DISTINCT(c_carrera) FROM alumnos)
```

Con los datos manejados, el resultado de esta consulta es vacío. Un resultado vacío no implica forzosamente un error. Quiere decir que no existe ningún renglón que cumpla los requerimientos señalados. En nuestro caso, es correcto.

Ligar tablas con WHERE tiene algunas limitaciones. Supone que el campo común no está vacío y existe en ambas tablas. Si no es el caso, utilice la instrucción JOIN. Existen cuatro modalidades:

"Combinaciones internas. Una *combinación interna* (*Inner join*) es el tipo de combinación predeterminado; especifica que solo se han de incluir en el resultado filas de la tabla que satisface la condición ON. Este comando produce el mismo resultado que la liga con WHERE.

[8] Según el sistema manejador de base de datos que se utilice, se debe poner AS para las tablas y/o para los campos. La sintaxis más portable es omitir el AS para tablas y emplearlo para campos: las instrucciones construidas de esta forma trabajan en Access, Oracle, SQL Server y MySQL.

"Combinaciones externas completas. Una *combinación externa completa* (*full outer join*) especifica que se deberían incluir en el resultado las filas no coincidentes (filas que no cumplen la condición ON) así como las filas que coincidan (filas que cumplen la condición ON). Para filas que no coincidan, aparecerá NULL en la columna de no coincidencia.

"Combinaciones externas por la izquierda. Una *combinación externa por la izquierda* (*left outer join*) devuelve las filas coincidentes más todas las filas de la tabla que se especifican a la izquierda de la palabra clave JOIN.

"Combinaciones externas por la derecha. Una *combinación externa por la derecha* (*right outer join*) es lo contrario a una combinación externa por la izquierda: devuelve las filas coincidentes más todas las filas de la tabla especificada a la derecha de la palabra clave JOIN."[9]

Como ejemplo, obsérvese el comando que crea el analizador sintáctico de SQL a partir de la sentencia anterior:

```
SELECT car.clave, car.descripcion, car.c_nivel, niv.descripcion AS nivel
FROM carreras car INNER JOIN
nivel niv ON car.c_nivel = niv.clave
```

A continuación se ofrecerá un ejemplo de los cuatro diferentes tipos de combinaciones. Para ello suponga que se crearon dos tablas con las siguientes instrucciones:

```
CREATE TABLEnombres1(
   clavevarchar(5)NOT NULL PRIMARY KEY,
   nombrevarchar(20)NOTNULL
)

CREATE TABLE nombres2 (
   clavevarchar(5)NOTNULLPRIMARY KEY,
   nombre varchar(20)NOTNULL
)
```

Suponga también que su contenido es el siguiente:

NOMBRES 2	
01	Ameyalli
02	Eva
03	Aidée
04	Adán
07	Adriana
08	María
09	Silvia
10	Eunice
11	Lourdes

[9] Frohock García, *et al. Running SQL Server 2000*. McGraw Hill, España, 2001, pp. 267-269.

Se listan las instrucciones con sus respectivos resultados:

```
SELECT * FROM nombres1, nombres2
WHERE nombres1.nombre = nombres2.nombre
```

04	Eva	02	Eva
03	Adán	04	Adán

```
SELECT * FROM nombres1
INNER JOIN nombres2 ON nombres1.nombre = nombres2.nombre
```

04	Eva	02	Eva
03	Adán	04	Adán

,

```
SELECT * FROM nombres1
LEFT JOIN nombres2 ON nombres1.nombre = nombres2.nombre
```

01	Miguel	NULL	NULL
02	Antonio	NULL	NULL
03	Adán	04	Adán
04	Eva	02	Eva
05	Luis	NULL	NULL
21	Laura	NULL	NULL
22	Gabriela	NULL	NULL

```
SELECT * FROM nombres1
RIGHT JOIN nombres2 ON nombres1.nombre = nombres2.nombre
```

NULL	NULL	01	Ameyalli
04	Eva	02	Eva
NULL	NULL	03	Aidée
03	Adán	04	Adán
NULL	NULL	07	Adriana
NULL	NULL	08	María
NULL	NULL	09	Silvia
NULL	NULL	10	Eunice
21	Laura	11	Lourdes

```
SELECT *
FROM nombres1 OUTER JOIN
nombres2 ON nombres1.nombre = nombres2.nombre
```

NULL	NULL	01	Ameyalli
04	Eva	02	Eva
NULL	NULL	03	Aidée
03	Adán	04	Adán
NULL	NULL	07	Adriana
NULL	NULL	08	María
NULL	NULL	09	Silvia
NULL	NULL	10	Eunice
NULL	NULL	11	Lourdes
21	Laura	NULL	NULL
05	Luis	NULL	NULL
01	Miguel	NULL	NULL
02	Antonio	NULL	NULL
22	Gabriela	NULL	NULL

Como ya se mencionó, la cláusula INSERT puede combinarse con SELECT. Para ejemplificarlo, supóngase que se desea crear una tabla con la información combinada de las tablas de ejemplo. La tabla tendría la estructura indicada por el siguiente comando:

```
CREATE TABLE alumnos2 (
   clave varchar (10),
   paterno varchar (30) ,
   materno varchar (30) ,
   nombre_de_pila varchar (30),
   c_carrera varchar (3),
   nombre_carrera varchar(40),
   c_nivel varchar(2),
   nombre_nivel varchar(30)
)
```

La forma de llenarla sería de la siguiente manera:

```
INSERT INTO alumnos2 (clave, paterno, materno, nombre_de_pila,
   c_carrera, nombre_carrera, c_nivel, nombre_nivel)
SELECT alu.clave, paterno, materno, nombre_de_pila, c_carrera,
   car.descripcion AS carrera, niv.clave AS c_nivel,
   niv.descripcion AS nivel
FROM carreras car, alumnos alu, nivel niv
WHERE car.c_nivel = niv.clave AND alu.c_carrera = car.clave
```

Actualización y borrado de datos en las tablas

Para borrar datos de una tabla se utiliza el comando

```
DELETE nombre de la tabla WHERE condiciones
```

Es de suma importancia colocar las condiciones, pues de lo contrario se borrarán todos los registros, aunque no podrá borrar registros de una tabla si otras tablas dependen de esta por integridad referencial. Antes de ejecutar el borrado debe estar seguro que la condición identifica con exactitud los registros que desea borrar (ni uno más ni uno menos). La cláusula WHERE puede combinarse con un SELECT hacia otra tabla para dar mayor versatilidad. La forma más común es WHERE campo NOT IN (SELECT campos FROM tabla).

Hagamos un ejemplo sencillo.

```
DELETE FROM alumnos
WHERE nombre_de_pila LIKE '%MIGUEL%'
```

La instrucción borrará a todas las personas con nombre Miguel (en nuestro caso es correcto, pues solo hay una con esas características). Sin embargo, si se deja un código así en un programa puede resultar riesgoso pues no sabemos si cambiarán las circunstancias en un futuro. Por eso, se sugiere utilizar la llave primaria para identificar un registro. La instrucción sería la siguiente:

```
DELETE FROM alumnos
WHERE clave LIKE '0083147888'
```

Para borrar una tabla completa se utiliza el comando `DROP TABLE nombre_de_la_tabla`.

Para actualizar valores en una o más filas se usa el comando

```
UPDATE nombre_de_la_tabla
SET campo = valor
WHERE condiciones
```

Pueden colocarse varios campos (con su respectivo valor), separados por comas. El nuevo valor del campo puede obtenerse si se combinan valores y campos de la tabla, incluso del propio campo a actualizar. Es de suma importancia colocar las condiciones. De lo contrario se actualizarán todos los registros. Por ejemplo:

```
UPDATE nivel
SET descripcion = 'superior'
WHERE clave = '01'
```

Debe tomarse en cuenta que la cláusula WHERE puede combinarse con un SELECT hacia otra tabla para dar mayor versatilidad. La forma más común es WHERE campo NOT IN (SELECT campos FROM tabla). Por ejemplo, supóngase que se tiene un listado de empleados y deben cambiarse las claves de puesto. El listado de empleados es el siguiente:

EMPLEADOS	
Clave	Puesto
000001	00001
000002	00002
000003	00001

Deben asignarse las nuevas claves de puesto como sigue:

PUESTOS	
Anterior	Nuevo
00001	00101
00002	00102

Puede hacerse una sentencia de actualización que realice estas modificaciones:

```
UPDATE empleados
SET puesto = (SELECT nuevo FROM puestos WHERE anterior = empleados.puesto)
```

Ejercicios sugeridos para el tema de SQL

Suponga que se tiene programada una serie de conferencias para alumnos y egresados del IPN. Se desea enviar una invitación a los interesados, pero solo para quienes no han asistido a la conferencia que se promueve. Las tablas manejadas son las siguientes:

CURSOS	
Clave	Descripción
NOR	normalización de base de datos
SQL	SQL básico
CSE	concepto de cliente-servidor

CONFERENCIAS				
Número	Día	Mes	Año	cve_curso
1		23	4	NOR
2		20	5	CSE

INTERESADOS			
Clave	Paterno	Materno	Nombre_de_pila
0083147888	LOPEZ	LUNA	MIGUEL
2004101234	LOPEZ	LOPEZ	AMEYALLI
2003569030	HUERTA	HERNANDEZ	CLAUDIA AIDÉE
2003688722	PEREZ	DE ALBA	LUIS ANTONIO

ASISTENTES	
cve_conferencia	cve_interesado
1	0083147888
1	2004101234
2	2004101234
2	2003569030

- Cree las tablas y respete la integridad referencial.
- Llene las tablas por medio de sentencias.
- Obtenga un listado de los interesados en orden alfabético.
- Obtenga un listado de las conferencias impartidas, que incluyan la descripción del tema.
- Obtenga el número de participantes por conferencia.
- Obtenga un listado de los alumnos que no han participado en ninguna conferencia.
- Obtenga un listado de interesados que no han participado en la conferencia "concepto de cliente-servidor".
- Dé de alta una nueva conferencia sobre "normalización", con fecha 22 de mayo de 2004.
- Obtenga el listado de las conferencias de mayo de 2004.
- Suponga que desea cambiar las claves de las conferencias.
  NOR NBD
  CSE CCS
  SQL SQB
  Diseñe las instrucciones UPDATE para este propósito.

6.3 Conectando una base de datos en MySQL con Java

Quien haya leído este capítulo de seguro se preguntará por qué nos hemos alejado de la programación orientada a objetos. De alguna forma, no mencionar Java durante varias páginas tiene una razón pedagógica: hacer hincapié en que el manejo de la información corresponde a la base de datos. La base de datos debe tener un diseño que permita su explotación adecuada y evite la redundancia; el sistema manejador de base de datos (SMBD) es el encargado de procesar las instrucciones para crear tablas e insertar, borrar y actualizar registros; el SMBD también valida que no se repitan llaves primarias ni se transgreda la integridad referencial. Y todo ello no tiene que ver con el lenguaje de programación utilizado; de hecho, el lenguaje de programación no puede violar ni corregir en forma directa las características que se hayan especificado en la base de datos (a menos que se permita el acceso con una clave de administrador con permisos totales y se modifique la propia estructura de la base de datos, lo cual no es muy recomendable).

REFLEXIÓN 6.2

¿Cómo ligar los cursos de programación con la base de datos?

Hemos observado en varios libros, temarios y en la práctica cotidiana de la docencia, que los temas de programación, base de datos e ingeniería de software son independientes por completo. Consideramos que, aunque esta medida es sana en lo general, sí debe cuidarse que los temas comunes sean enlazados en forma adecuada. En programación debieran brindarse algunos elementos básicos de SQL y no solo las instrucciones de conexión, con el propósito de que el estudiante perciba que crear una base de datos conlleva un gran trabajo de análisis y normalización que verá en otros cursos; por otra parte, quien imparta base de datos debiera tener una idea general del uso de cadenas de conexión desde los lenguajes de programación para ayudar a tender el "puente" entre estas dos áreas de conocimiento.

El profesor del curso de lenguajes de programación no debiera permitir que en los trabajos finales de su materia existan errores delicados en el diseño de la base de datos, aunque el tema "no pertenezca a su clase".

Es momento de "tender un puente" entre la base de datos y el lenguaje de programación a través de un driver que sirva de enlace entre ambos productos dentro del contexto del sistema operativo manejado. Por desgracia, no existe un driver universal y por tanto hay que buscar al que corresponda según el sistema manejador de base de datos y el lenguaje de programación. Incluso, pueden ser diferentes según las versiones del SMBD, el lenguaje de programación y el propio sistema operativo.

En nuestro caso, se trata de conectar Java. De manera más concreta, el jdk1.7.0_51. Del otro lado, MySQL Workbench 6.0 CE Community. Nuestro contexto es el sistema operativo Windows 7.0 Professional. Se cuidó que las versiones de Java, MySQL y la librería citada estuvieran disponibles a inicios de 2014. Para eso se recurrió a la librería mysql-connector-java-5.1.22-bin (con extensión .jar), que se descargó del sitio correspondiente de MySQL. Esa librería se agregó a las librerías de JCreator después de usar la opción \Configure \Options; en la pantalla se eligió JDK Profiles, se seleccionó la versión activa del compilador (en nuestro caso JDK versión 1.7.0_51) y se le dio Edit. Ahí se agrega mediante el botón Add la librería citada, que por lo regular, se suele colocar en el directorio correspondiente a las librerías del compilador (C:\Program Files\Java\jdk1.7.0_51\lib) (véase figura 6.6).

Para crear la conexión es necesario seguir los siguientes pasos (véase programa 6.1):[10]

- Cargar el controlador.
- Conectar con el SMBD.
- Crear y ejecutar una instrucción SQL.
- Procesar los datos que devuelve el SMBD.
- Terminar la conexión con el SMBD.

[10] Sintetizado y adaptado de Keogh, Jim. *J2EE Manual de Referencia*. McGraw-Hill, España, 2003, pp. 126-134.

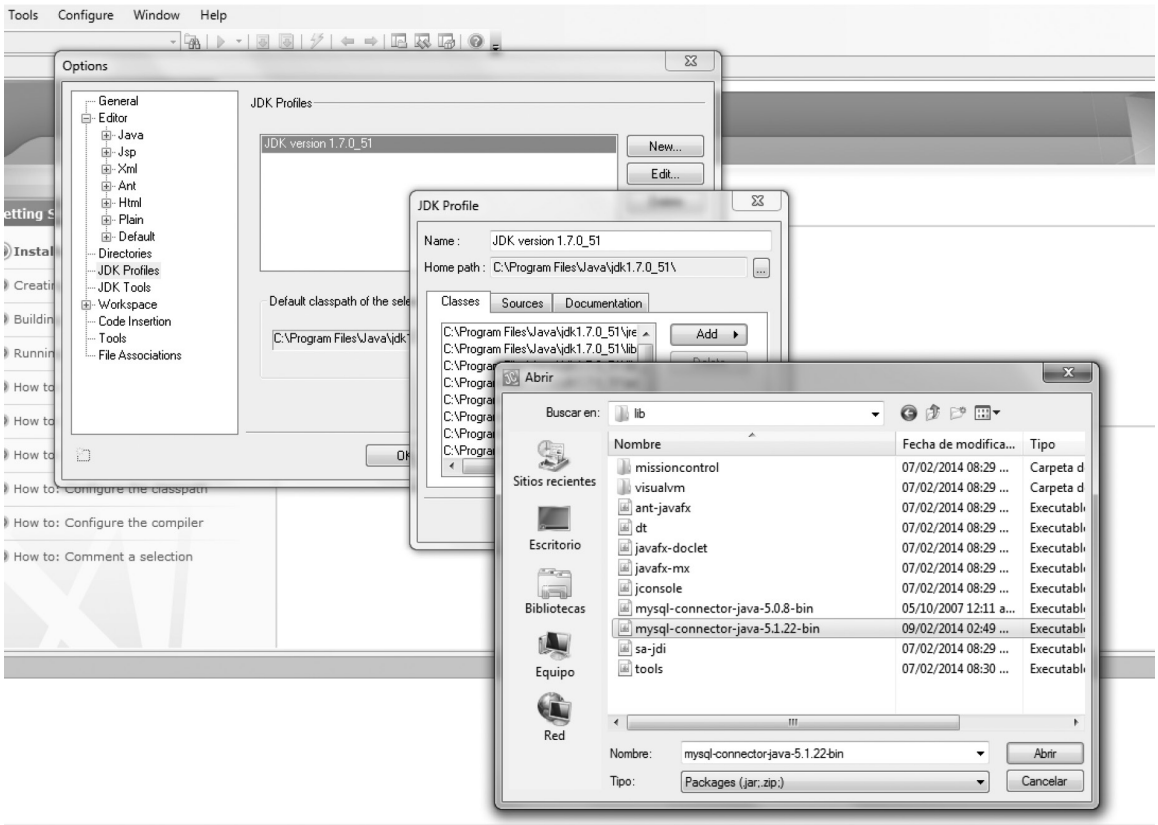


Figura 6.6 Librería de conexión de MySQL agregada al compilador.

```
1   package megasinapsisJava; // no olvide poner su propio paquete
2   importjava.sql.*;
3
4   public class ClasesBase {
5      public static void main (String[] args)
6                throwsClassNotFoundException, SQLException {
7         String url = "jdbc:mysql://localhost:3306/test";
8         String usuario = "root";
9         String password = "eslmqceRD2304";
10        Connection c = null;
11        Statement s = null;
12        ResultSetrs = null;
13        String cadenaPrueba;
14        try {
15           Class.forName("com.mysql.jdbc.Driver");
16           c = DriverManager.getConnection(url,usuario,password);
17           s = c.createStatement();
18           rs = s.executeQuery("SELECT * FROM nivel");
19           System.out.println("CLAVE DESCRIPCION \n");
20           while (rs.next()) {
21              // desplegando a través de una variable String
22              cadenaPrueba = rs.getString("clave") + " "
23                + rs.getString("descripcion");
24              System.out.println(cadenaPrueba + "\n");
```

```
25          }
26          c.close();
27          rs.close();
28          s.close();
29       }
30       finally {
31          if (rs != null) {
32             try {
33             rs.close();
34             } catch (SQLException e) { }
35          }
36          if (s != null) {
37             try {
38                s.close();
39             } catch (SQLException e) { }
40          }
41          if (c != null) {
42             try {
43             c.close();
44             } catch (SQLException e) { }
45          }
46       } // finally
47    } // main
48 } // clase
```

Programa 6.1 Conexión con Java a una base de datos de MySQL.

Es necesario cargar el controlador JDBC para que el componente pueda conectarse al SMBD; para eso se emplea el método `Class.forName()` y se utiliza como parámetro el nombre del controlador que corresponda al sistema manejador de base de datos en la versión en que se trabaja. En nuestro caso la instrucción está en la línea 15:

```
Class.forName("com.mysql.jdbc.Driver");
```

Una vez cargado el controlador, se debe conectar mediante el método `DriverManager.getConnection()` si se emplea el URL de la base de datos, el usuario y la clave de acceso (línea 16); el método `getConnection` devuelve un objeto que implementa la interfaz `Connection` (`conexión`), la cual se utiliza a lo largo del proceso para hacer referencia a la base de datos. La clase `java.sql.DriverManager` (gestor de controladores) es la clase más alta dentro de la jerarquía de JDBC y es la responsable de gestionar la información de los controladores. La interfaz `java.sql. Connection` es otro miembro del paquete java.sql, el cual gestiona la comunicación entre la base de datos, el controlador JDBC y el componente J2EE (Java 2 Enterprise Edition); esta interfaz envía las instrucciones al SMBD para su procesamiento.

El siguiente paso es enviar una consulta SQL para su ejecución. El método `Connection.createStatement()` se emplea para crear un objeto de tipo `Statement` (línea 17). Con base en este objeto se envía una instrucción SQL que se ejecuta al invocar el método `executeQuery`, el cual devuelve la respuesta del SMBD en un `objeto ResultSet` (por lo general, un conjunto de registros, línea 18). Es usual que los resultados se asignen a un objeto de tipo `String` para su posterior manipulación, la cual se hace con las instrucciones del lenguaje Java.

Por último, una recomendación muy importante: cierre todas las conexiones que ha abierto y considera una terminación normal (líneas 26-28), tanto como una terminación inesperada (líneas 30-45). Cabe aclarar que en este caso bastaría con emplear el cierre en finally, pues incluye ambas situaciones. No obstante, es redundante para que el estudiante perciba la forma de considerar ambas situaciones: la normal y aquella preparada para casos inesperados y que logra dar robustez a las aplicaciones.

El "puente" nos trajo la información desde la base de datos a nuestro programa en objetos de tipo String (véase figura 6.7). ¡Ya estamos en el campo del lenguaje Java!

```
package megasinapsisJava;     // no olvide poner su propio paquete
import java.sql.*;

public class ClasesBase {
  public static void main (String[] args) throws ClassNotFoundException, SQLException {
    String url = "jdbc:mysql://localhost:3306/test";
    String usuario = "root";
    String password = "eslmqceRD2304";
    Connection c = null;
    Statement s = null;
    ResultSet rs = null;
    String cadenaPrueba;
    try {
        Class.forName("com.mysql.jdbc.Driver");
        c = DriverManager.getConnection(url,usuario,password);
        s = c.createStatement();
        rs = s.executeQuery("SELECT * FROM nivel");
        System.out.println("CLAVE    DESCRIPCION \n");
        while (rs.next()) {
            // desplegando a través de una variable String
            cadenaPrueba = rs.getString("clave") + " " + rs.getString("descripcion");
            System.out.println(cadenaPrueba + "\n");
        }
        c.close();
        rs.close();
        s.close();
    }
```

```
eneral Output
----------------------Configur
CLAVE    DESCRIPCION

01 Nivel superior

02 Nivel medio superior

03 Nivel maestría


Process completed.
```

Figura 6.7 Resultado de la ejecución del programa ejemplo de conexión a base de datos.

EJERCICIOS SUGERIDOS

- Logre la conexión citada en este subcapítulo al seguir uno a uno los pasos que se señalan.
- Logre una conexión a una base de datos que contenga una tabla con la misma estructura, pero montada en otro sistema manejador de base de datos (SQL Server, Access, Postgress, etcétera).

Capítulo 7

Aspectos metodológicos básicos para la programación

7.1 El concepto BANO (Bienes Adquiridos Nunca Ocupados)[1]

Un BANO (Bien Adquirido Nunca Ocupado) es aquel bien o servicio que nunca fue utilizado, por lo cual la inversión que se hizo en él fue infructuosa por completo. Los BANOS son un error desde todos los puntos de vista, incluyendo —por supuesto— el financiero y el ecológico.

Mencionamos este concepto porque desde hace décadas se citan estudios que señalan que alrededor de la mitad de los esfuerzos del área de sistemas son BANOS. Una encuesta del grupo Standish realizada en 2000 con base en el análisis de 280 000 proyectos arrojó que 23% fueron cancelados antes de terminarse; en otra encuesta, realizada en 2002 por el Cutre Consortium se cita que en 45% de los casos las fallas de los sistemas eran tan graves que el producto de software no podía utilizarse (véase figura 7.1).[2]

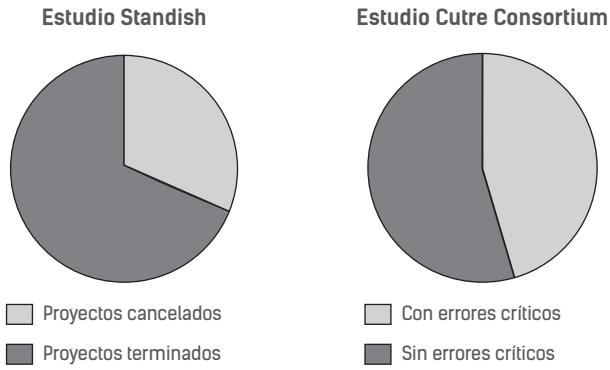


Figura 7.1 Dos estudios que ejemplifican la llamada "crisis del software".

Los BANOS incluyen:

- Sistemas completos de los que ninguna de sus partes fue llevada a producción o nunca lograron estabilizarse como sistemas. Por ejemplo, el caso del Registro Nacional de Usuarios de Telefonía Celular en México, nacido por ley en 2010 y dado de baja también por ley en 2012 y cuyas bases de datos se vendieron de manera ilegal.[3]
- Módulos de un sistema que fueron desarrollados y nunca fueron utilizados.
- Funcionalidades que se habían terminado y tuvieron que ser modificadas antes de liberar el sistema.
- Adquisiciones de software que nunca fueron ocupadas.

No pueden considerarse BANOS aquellas adquisiciones que cubren contingencias probables, se den o no. Por ejemplo: respaldos, arreglos RAID en servidores, seguros, etc. Tampoco incluyen las actividades realizadas para la evaluación de alternativas o *benchmarking*, independientemente de qué opción se haya elegido. El diseño de prototipos que exploran nuevas posibilidades técnicas tampoco puede considerarse un BANO ni las actividades para adquirir nuevos conocimientos, siempre y cuando exista la posibilidad de que estos puedan traer un beneficio futuro para la organización (en consecuencia, el apoyo a la investigación científica nunca es un BANO para un país).

---

[1] El concepto de BANO fue dado a conocer en López Goytia, José Luis y César Cruz, Ana Karen. "*Los BANOS: un problema no reconocido*", NotiUpiicsa, año 4, No. 6, julio-septiembre de 2013, de donde fue adaptada esta sección.

[2] Schach, Stephen. *Ingeniería de software clásica y orientada a objetos*. McGraw-Hill, sexta edición, México, 2006, p. 6.

[3] http://www.eluniversal.com.mx/notas/673768.html

REFLEXIÓN 7.1

Una compra infructuosa de casi un millón de dólares[4]

Una universidad compró un software a una empresa muy reconocida con un valor aproximado de 10 millones de pesos para controlar la publicación de sus páginas web. Los módulos que compró no incluían las partes que permitían realizar esta labor, por lo cual fue sustituido por otro software de la misma empresa cuyo costo fue de 3 millones de pesos. Se trató de utilizar el primer software para el control de flujos de información, lo cual parecía viable, pues la universidad genera alrededor de dos millones de documentos anuales. No obstante, no fue posible lograr que se autorizara el gasto en capacitación, compra de un módulo adicional requerido y soporte (alrededor de cuatro millones de pesos anuales). Al final, nunca se llegó a utilizar.

Debemos dejar en claro que el problema en este caso no fue la herramienta (por cierto, con una buena imagen en el mercado), sino la falta de un análisis de las necesidades y la coherencia entre estas necesidades y la adquisición del software.

Sistema		**Horas**	**¿Se usó?**
Control de fondo de ahorro		480	Sí
Examen psicométrico especial		120	Sí
Procesamiento de encuesta para cliente Digito Todo		80	No
Subtotal de BANOS en sistemas completos o inconclusos:		**80**	
Control de fondo de ahorro		**Horas**	**¿Se usó?**
Captura de préstamos de fondo y caja de ahorro		120	Sí
Reporte individual de fondo y caja de ahorro		100	Sí
Reporte concentrado de fondo y caja de ahorro		140	Sí
Cálculo de intereses de fondo de ahorro		120	Sí
Control de examen psicométrico		**Horas**	**¿Se usó?**
Captura del examen		30	Sí
Procesamiento de calificación		30	Sí
Resultado individualizado		20	Sí
Reporte comparativo de candidatos		40	No
	Total:	120	
Total de BANOS en funcionalidades:		**40**	
Cálculo de intereses de fondo de ahorro (versión 1):		100	
+ Adecuación al requerimiento real (que era más simple)		20	
= Total de esfuerzo dedicado		120	
– Tiempo necesario para producir el requerimiento		40	
Total de BANOS por cambios de requerimientos:		**80**	
Total de BANOS en sistemas completos:	80		
Total de BANOS en funcionalidades:	40		
Total de BANOS por cambios en requerimientos:	80		
Esfuerzo total del área de desarrollo:	680		
Total de BANOS del semestre (en horas)	**200 (29.4% del esfuerzo total)**		

Cuadro 7.1 Un cuadro ilustrativo de BANOS en desarrollo de software.[4]

[4] El caso es real, pero por aspectos de confidencialidad se omitió citar la organización y la empresa proveedora.

Para medir los BANOS es necesario considerar tres conceptos y aplicarlos a un periodo específico:

a) Realizar un listado completo de todos los sistemas de la empresa y el esfuerzo dedicado a cada uno de ellos. Aquellos que nunca se llegaron a ocupar constituyen BANOS.
b) Tener una relación de las funcionalidades de cada sistema que sí fue ocupado y el esfuerzo dedicado a cada uno de ellos. Aquellas funcionalidades que nunca se utilizaron constituyen BANOS.
c) Listar funcionalidades que ya estaban terminadas y fueron reprogramadas. En este caso, el BANO es el tiempo desperdiciado por tomar este "camino equivocado". Solo se considerarán funcionalidades que sí fueron ocupadas.

Para obtener el esfuerzo en BANOS hay que sumar los tres conceptos anteriores. El porcentaje en BANOS es el esfuerzo en BANOS dividido entre el esfuerzo total dedicado al desarrollo de sistemas. Un ejemplo se da en el cuadro 7.1 basada en una experiencia real, en donde la empresa Tecnología Avanzada —por supuesto, el nombre es ficticio— desarrolló tres sistemas pequeños durante el primer semestre de 2008.

La mayoría de los BANOS se producen por una mala gestión en el desarrollo de sistemas, sobre todo en los aspectos metodológicos. También llegan a generarse por problemáticas tecnológicas, aunque con menor frecuencia. Algunas posibles causas:

a) El área de sistemas o algún área usuaria promueve desarrollos ajenos a los intereses de la organización. La causa es la falta de análisis de los objetivos de la empresa, en muchos casos combinados con problemas de corrupción.
b) Los directivos solicitan sistemas "contra reloj" en un tiempo imposible de conseguir. Estos casos, en general, son causados por directivos que actúan sin un plan a mediano plazo y necesitan dar un resultado visible muy rápido. Pasada la urgencia, el sistema suele quedar casi olvidado, aun cuando este se siga desarrollando.
c) El desarrollo se basó en un único usuario, sin hacer un análisis de las necesidades de los distintos usuarios finales del sistema. Cuando se intenta liberar, se detectan un sinnúmero de carencias.
d) No hubo una planificación del proyecto. Los tiempos requeridos eran mucho mayores a los estimados *a priori*. De manera paulatina o abrupta, los sistemas se paralizan ante la falta de los recursos adicionales necesarios.
e) En la etapa de pruebas se detectó que la arquitectura no soportaba el volumen de información requerido, con lo cual es necesario reiniciar el proyecto.
f) Los directivos toman decisiones "de buena fe" sin analizar las implicaciones. Por ejemplo: en México se ordenó que durante todo 2013 cualquier documento oficial en dependencias gubernamentales debía llevar el escudo nacional como fondo de agua. Sin duda, fue una instrucción con hermosa intención, pero que nunca analizó los costos de la medida en millones de impresiones y fotocopias.

Los BANOS siempre van a existir. Hay que desconfiar cuando un gerente de sistemas o un líder de proyecto dicen tener 0% de BANOS; casi es seguro que falseen la información o se expresan con honestidad pero tienen una imagen irreal de su área. Como en muchas enfermedades, reconocer el problema y tratar de medirlo es la mitad del camino. La primera recomendación es, entonces, realizar una medición de los BANOS que han existido en un área en un periodo cercano (por ejemplo: el semestre recién terminado).

La medición de los BANOS siempre será un asunto problemático. Primero, porque pocos tienen una estimación del esfuerzo dedicado a cada sistema. En segundo lugar, porque es difícil que alguien reconozca que existen BANOS. Un tercer punto: es difícil identificar un BANO cuando se acaba de terminar un sistema, pues muchas veces no se sabe si en realidad se utilizará o no.

A pesar de su dificultad, medir los BANOS puede resultar una tarea muy atractiva pues detecta un sinnúmero de esfuerzos inútiles y ayuda a identificar las causas que los producen. Si se ataca el origen de los BANOS se puede aumentar la eficiencia del área de sistemas de manera muy significativa.

Desde un punto de vista técnico, la vertiente principal para la reducción de BANOS es la aplicación de normas metodológicas básicas, en particular con un proceso unificado.

Habría que destacar:

a) La incorporación de los distintos tipos de usuario desde la etapa de requerimientos.
b) Una etapa inicial de diagnóstico general del sistema antes de aprobarse el proyecto. Esta etapa debe detectar los requerimientos generales, así como alternativas de solución, además de establecer los costos y tiempos aproximados del proyecto.
c) Probar desde un inicio que el desarrollo es viable bajo el diseño arquitectónico elegido cuando no existe experiencia en sistemas de igual complejidad técnica. Es muy recomendable auxiliarse con expertos en la materia.

Hablar de BANOS puede representar un choque cultural para muchas personas del área de sistemas. Aunque al mismo tiempo puede ser un aspecto motivador para los desarrolladores. (¡Por fin haremos algo que sí se va a usar!) Tanto los directivos como los equipos de desarrollo deben beneficiarse de alguna forma con la reducción de BANOS y el aumento de eficiencia (logro de objetivos, horario más flexible, más capacitación, incentivos, mejor ambiente de trabajo, etc.). Querer hacerlo con "medidas de capataz" suele resultar contraproducente.

EJERCICIO SUGERIDO

El área de sistemas tuvo los siguientes desarrollos en 2009:

El prototipo de control de gestión para mandar oficios vía electrónica y ahorrar papel se realizó con éxito y demostró su viabilidad. Consumió 400 horas de trabajo. Al hacer el plan de trabajo para el desarrollo definitivo, se concluyó que casi no se podía aprovechar ninguna parte del código.

El sistema de seguimiento de contratos del abogado general se tuvo que reprocesar, con un consumo de 960 horas, de las cuales se puede considerar que 200 se perdieron en el reproceso.

La mejora de seguridad del sistema de control escolar se finalizó en tiempo y forma, con un consumo de 500 horas.

La mejora al sistema administrativo de contratación nunca se usó. Se invirtieron en él 500 horas.

El manejo de becas sí se ocupó, con excepción del módulo de becas por excelencia, que nunca fue utilizado. El consumo total fue de 600 horas; el módulo de becas por excelencia consumió 120 horas.

¿Cuál fue el porcentaje anual de BANOS?

REFLEXIÓN 6.2

La necesidad de reconocer los BANOS en los cursos de programación

Es necesario incorporar el concepto de BANO en los cursos de programación. Por una parte, se recomienda que el software se lleve ante un usuario real (situación bastante plausible en muchos programas concretos y juegos); por otra parte, que se reprocese hasta quedar dentro de los requerimientos. Sin pruebas de usuario real y reprocesos es muy difícil que el estudiante capte lo triste de elaborar un software que al final se convierte en un BANO.

7.2 ¿Qué es un sistema de información?

Un sistema de información (SI) es un sistema cuyos componentes —automatizados o no— transforman un conjunto de datos de entrada en una información establecida y delimitada con claridad, que tiene un sig-

nificado primordial para la organización que lo utiliza. Un SI conlleva el uso de programas de aplicación y una o más bases de datos, así como hardware e infraestructura de comunicaciones sobre los cuales se instala, y una serie de procesos administrativos que alimentan al software y explotan su información. Está inmerso dentro de una organización: su estructura orgánica, su cultura, su normatividad y los recursos financieros, técnicos y humanos de los cuales dispone (véase figura 7.2). Por eso un software que funcione de manera adecuada en una empresa o en un contexto determinado puede resultar un fracaso en otras situaciones o quedar obsoleto por completo al paso del tiempo.

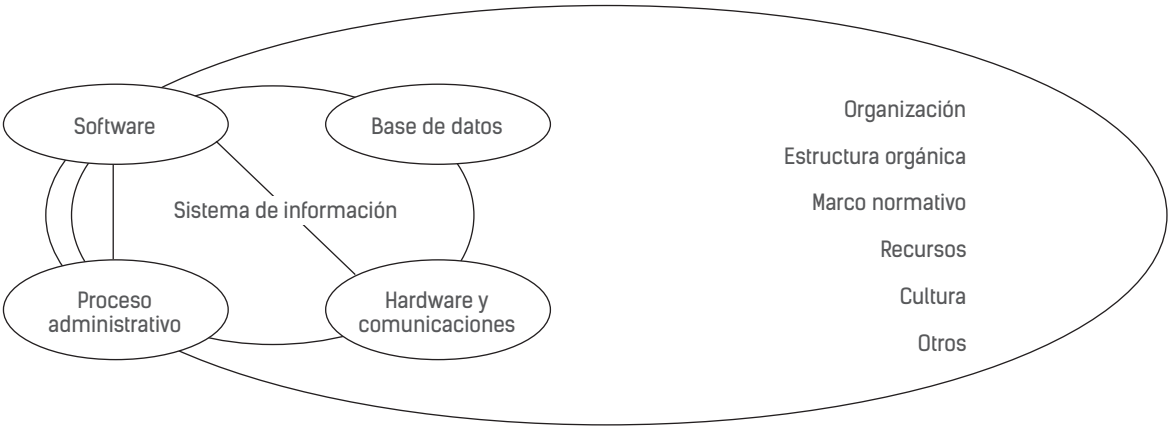


Figura 7.2 El concepto de sistema de información.

Un componente de software —por ejemplo: una subrutina, una librería, una clase o un paquete— es una pieza de propósito específico que realiza una labor determinada y tiene entradas y salidas definidas con claridad, con la finalidad de que dicho componente sea parte integrante de un sistema de información.

Para que un sistema de información funcione de manera adecuada se necesita que no existan fallas significativas en ninguno de sus elementos y que se relacione de manera consistente con la organización (véase casos de estudio).

REFLEXIÓN 7.3

Un sistema de exámenes con tecnología "obsoleta", pero que trabaja adecuadamente.

Una dependencia gubernamental tenía en 2008 un Centro de Evaluación con 720 terminales tontas, las cuales eran utilizadas para hacer exámenes para promoción a distintos aspirantes. Para ello se alimentaba la aplicación con los concursos a realizar, cada uno de los cuales consideraba diferentes materias. Los reactivos eran realizados por diversos instructores. Para evitar errores se diseñaron protectores de cartón para que el usuario no pudiera oprimir teclas que no aplicaran al caso y otros para impedir que vieran las respuestas del concursante vecino. Además, las preguntas se presentaban en orden aleatorio para cada aspirante.

A falta de un sistema que manejara la base de datos, se grababa todo en archivos binarios. El sistema estaba realizado en lenguaje C y solo manejaba texto por las limitaciones de hardware y comunicaciones.

El sistema presentaba limitaciones, pero llevaba años sin haberse "caído" y trabajando en forma correcta, aunque con las restricciones derivadas del presupuesto bajo.

REFLEXIÓN 7.4

Una aplicación administrativa que basa su adecuado funcionamiento en tecnología que ya no existe

El formato único de personal (FUP) de una universidad pública era el documento oficial a través del cual se mandaban todas las notificaciones de movimientos de personal (altas, bajas, permisos, cambios de adscripción, etc.). Este formato estaba en 2009 en hojas que excedían el tamaño carta a lo largo y a lo ancho (originalmente se diseñó para impresoras de matriz). En 2007 se realizó una automatización de su llenado. Sin embargo, para los usuarios este fue un avance muy limitado, pues debido al tamaño del papel tenían que alimentar el sistema y después llenar el formato en máquina de escribir electrónica, con un fuerte consumo en tiempo y altas probabilidades de error.

¿Por qué no se pasaba este formato a tamaño carta? Según explicación del usuario, era el aprobado de manera oficial por la Secretaría de Educación Pública y no podía hacer nada al respecto sin su autorización. No obstante, el analista nunca encontró ningún reglamento que indicara que ese formato no podría cambiarse a tamaño carta.

EJERCICIO SUGERIDO

Una escuela tiene un sistema de control escolar llamado SICOES, el cual se ejecuta en internet de manera aceptable. Sin embargo, hay fuertes problemas de seguridad y en ocasiones el procesamiento se vuelve lento por problemas de las instalaciones de red.

Por otra parte, el reglamento exige que los alumnos que reprobaron alguna materia tengan una entrevista con un órgano colegiado para analizar su situación. Esa estrategia funcionó cuando eran pocos alumnos, pero se volvió ineficiente cuando el número de estudiantes pasó de 200 a 4 000.

Con los elementos anteriores, identifique elementos a mejorar en:

a) El software.

b) Las tecnologías de información que rodean al software.

c) Los aspectos administrativos del sistema de información.

REFLEXIÓN 7.5

Nunca dejar de analizar el entorno cuando se habla de un sistema

En muchas ocasiones el estudiante realiza trabajos para un usuario final. En estos casos es primordial que se acostumbre, en primer lugar, a que exista en realidad un usuario final; en segundo lugar, analizar los aspectos del entorno (reglamentos, tipo de usuario, equipo en el que trabajará, etc.); por último, que el usuario final realice efectivamente una prueba. Cuando esto no se hace resultan programas bien realizados desde el punto de vista técnico pero inaplicables en el entorno al que se dirigen. Por ejemplo: juegos para niños con instrucciones redactadas en lenguaje universitario.

7.3 Orientación hacia el usuario final

El software por sí mismo no tiene valor; es una herramienta que forma parte de un sistema de información. Dicho sistema de información debe apoyar una o más actividades primordiales de una organización, tanto en su aspecto operativo como en la posibilidad de extraer información valiosa para tomar decisiones.

En otras palabras, los sistemas de información deben proporcionar a los usuarios encargados de tomar decisiones (**usuarios finales**) la información que estos requieran a fin de realizar una correcta evaluación de las alternativas existentes. Los sistemas de información también sirven para hacer más eficientes los procesos a través de la sistematización y automatización de los procesos.

Conviene hacer una distinción entre los **usuarios finales** y los **usuarios intermedios**. Mientras el **usuario final** usa la información para tomar una decisión o realizar una operación que le interesa en forma

directa, el **usuario intermedio** solicita información por la obligación de entregar datos a otra área, sin que exista un interés directo en esa información, salvo el cumplimiento de su tarea cotidiana.

¿Pueden eliminarse los requerimientos de los "usuarios finales" sin afectar a la organización en su conjunto? No, pues quedarían sin elementos de información valiosos que pueden afectar a un proceso clave dentro de la organización.

¿Pueden eliminarse los requerimientos de los "usuarios intermedios"? Sí, siempre y cuando no afecten a procesos clave de los "usuarios finales".

La orientación al usuario final significa que se tratará de satisfacer los requerimientos de información de los usuarios finales, reacomodando en caso necesario la forma en que se realizan los procesos y los requerimientos de información de los "usuarios intermedios".

Un desarrollador de software que se guía por el enfoque orientado al usuario final no puede aceptar una automatización pasiva de los procesos actuales. Siempre debe analizar si estos responden a necesidades de usuarios intermedios o finales y, en su caso, proponer mejoras que eliminen pasos innecesarios y optimicen los resultados esperados por los usuarios finales (véase caso de estudio).

REFLEXIÓN 7.6

Rediseñar un proceso de impuestos

Diferenciar a los usuarios finales de los intermedios no es sencillo en la práctica.

Aprovechando el cambio de sistema de nómina, el responsable de nómina solicitó a la empresa proveedora que el cálculo de impuestos se realizará con determinadas especificaciones. La empresa proveedora observó que ese cálculo dificultaría enormemente el cierre anual y ofreció una mejor alternativa. Se lo hizo saber al usuario, quien contestó que estaba de acuerdo en la mejora, pero que él no podía hacer nada, pues así se lo había pedido el área de contabilidad.

Antes de implementarlo, se platicó con el área de contabilidad, quien se asombró de que hubiera otras posibilidades, pues el sistema anterior solo manejaba la opción que se ejecutaba en la actualidad. El responsable de contabilidad revisó a conciencia las alternativas que brindaba la reglamentación fiscal, apoyado por un asesor fiscal. Al final, resolvió que convenía modificar el cálculo bajo unos lineamientos que eran, incluso, mejores a los ofrecidos por la empresa proveedora del nuevo software de nómina. La opción comentada no le representaba cargas adicionales de trabajo a la empresa proveedora y le beneficiaba a las áreas de contabilidad y nómina. El cambio de proceso se formalizó y con base en esta documentación fue implementado.

En este caso, nómina actuó como un usuario intermedio, mientras contabilidad en realidad hizo las veces de usuario final. Otro usuario final implícito fue la Secretaría de Hacienda, pues el cálculo fue realizado con base en las especificaciones que dictaban sus reglamentos.

Para el caso de la compra-venta de componentes de software específico esta definición se aplica desde diferentes perspectivas. Por ejemplo: suponga que una empresa elabora un paquete de clases que pueden identificar en un segundo una huella digital entre 40 000 y que permite ser combinado con varios lenguajes de programación. Desde el punto de vista del vendedor, la creación de componentes se justifica cuando estos se promocionan y se venden. Desde el punto de vista del usuario, se justifica cuando el componente adquirido se interrelaciona con otras piezas de software para conformar un sistema de información para aspectos de seguridad, control de entradas y salidas de personal u otro campo de aplicación.

¿Tendría sentido comprar un software que no se usa? La respuesta es obvia: no, habríamos hecho un BANO. Y eso aplica para cualquier producto. El software más caro es aquel que no se usa. Un software que no está ligado a un sistema que lo explote de manera adecuada es un "software muerto". Entonces, ¿por qué muchas experiencias que hemos tenido y algunos estudios localizados inducen a pensar que cerca de 50% del software nunca se llega a utilizar?

REFLEXIÓN 7.7

Una evaluación de desempeño que solo consideró a un usuario final

Después de un proceso de acreditación, una carrera recibió una observación: que no había evaluaciones docentes cotidianas. Por ello, el subdirector académico encargó a un jefe de departamento su implementación. El jefe de departamento elaboró el instrumento y coordinó a un grupo de alumnos para el desarrollo del software. Cada estudiante tenía que llenar la evaluación docente semestre tras semestre.

No obstante, solo se imprimieron los reportes individuales y se les hicieron llegar a los docentes. Efectivamente, se cubrió la necesidad planteada por los organismos acreditadores (que pueden considerarse un usuario final). Sin embargo, a los usuarios finales más importantes (estudiantes y coordinadores académicos) sigue sin llegarles la información.

Además, los requerimientos de los organismos acreditadores fueron modificados. Como nunca se mantuvo una comunicación con ellos, la información que se obtenía no fue la adecuada.

Como podrá observarse, el sistema solo respondió a las necesidades de un usuario final único, con el cual nunca hubo retroalimentación.

Una de las causas más comunes es levantar los requisitos de un sistema basándose en el punto de vista de uno de los usuarios finales, al hacer caso omiso de los demás. Incluso puede darse un caso más lamentable: que solo se consulte a usuarios intermedios. La orientación al usuario final es primordial para los sistemas web, pues por su naturaleza siempre tienen varios usuarios finales.

REFLEXIÓN 7.8

Cuando se plantea la realización de un software en clase, no olvidar a los distintos usuarios

En muchas escuelas, los trabajos de programación de semestres avanzados se basan en un único tipo de usuario final, cuando resulta notorio que existen varios tipos de usuario finales, en particular tratándose de sistemas administrativos. Los cursos de programación deben buscar que se respeten aspectos básicos de ingeniería de software.

EJERCICIO SUGERIDO

En la escuela que dirige se ha generado un problema con los sobrecupos. Por una parte, algunos profesores piden conservar el derecho de admitir o no admitir a más alumnos, pues argumentan que el tamaño de los grupos es excesivo. Los estudiantes no tienen conflicto con ello. Como solución, la coordinación académica de la escuela elaboró un formato en el cual el profesor firmará de conformidad. No obstante, la reglamentación exige que todo documento académico debe ir avalado por el jefe de división y el subdirector académico. Al final, el estudiante tiene que recopilar las firmas del profesor, el jefe de división y el subdirector académico para poderse inscribir con sobrecupo, cuya consecuencia es la pérdida de alrededor de una semana de clases en esa materia.

¿Quiénes son los usuarios intermedios y finales en este proceso? ¿Considera que existe alguna otra solución con la tecnología actual para simplificar el proceso?

7.4 Análisis costo-beneficio

El **costo inicial** de un sistema es la suma de todos los costos que deberán realizarse para que el sistema llegue a un funcionamiento normal. El **costo de mantenimiento** es la suma de todos los costos que deberán realizarse para que el sistema se mantenga en funcionamiento a un nivel de funcionalidad equivalente al momento inicial del sistema.

El **análisis costo-beneficio** es un comparativo de los costos de un sistema propuesto (costo inicial y de mantenimiento) contra el costo del sistema vigente. En general se parte del hecho de que ambos sistemas brindan una funcionalidad similar. En la práctica, existen diferencias en este sentido, por lo regular

a favor del sistema propuesto, por lo cual es conveniente documentar los beneficios para tener argumentos adicionales para la propuesta. De manera gráfica, el costo-beneficio se presenta como dos funciones, referentes al viejo y al nuevo sistema, aunque nada impide que se presenten varias propuestas en forma simultánea (véase figura 7.3).

El **tiempo de retorno de la inversión** se obtiene al dividir el costo inicial entre la diferencia de costo de los sistemas. Solo aplica cuando el costo de mantenimiento del nuevo sistema es menor al del sistema actual.

La **depreciación** es la pérdida del valor de un bien por el paso del tiempo. La depreciación anual se puede medir como el valor del bien dividido entre el número de años que esté en uso activo.

Para efectos del análisis costo-beneficio la depreciación no debe incluirse en el costo de mantenimiento, pues el costo inicial ya incluye todo el valor del bien y no puede duplicarse dicho costo. Otra opción igual de válida sería incorporar la depreciación en cada año (hasta alcanzar el total), pero al asumir el costo inicial como cero.

$$\text{tiempo de retorno de la inversión} = \frac{\text{c.i.n.s}}{\text{c.m.n.s.} - \text{c.m.s.a.}}$$

c.i.n.s. = costo inicial del nuevo sistema
c.m.n.s. = costo de mantenimiento del nuevo sistema
c.m.s.a. = costo de mantenimiento del sistema anterior

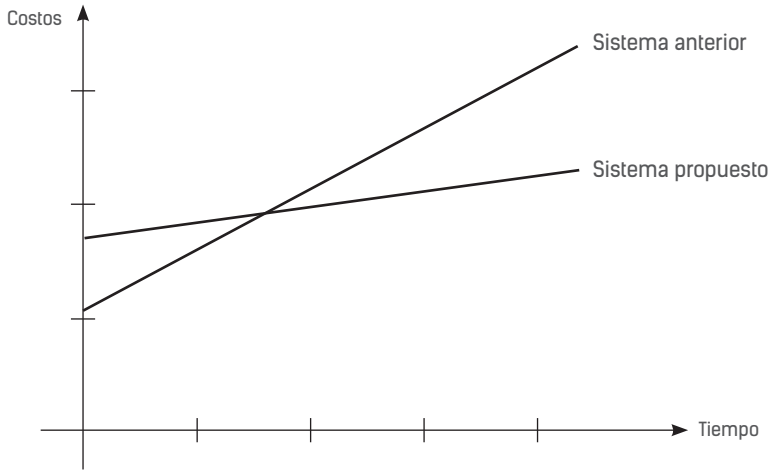


Figura 7.3 El modelo básico costo-beneficio.

Este concepto es adecuado como punto de partida. No obstante, tiene al menos dos limitaciones que deben tomarse en cuenta:

a) **El costo de mantenimiento se considera fijo a lo largo del tiempo.** En la realidad, el costo de mantenimiento suele incrementarse año con año y pudiera darse el caso de que la variación sea un poco "extraña". Por ejemplo, el precio de la póliza de mantenimiento de una impresora Dell 5310n es el siguiente[5] (cada año habría que sumar la cifra indicada): año 1 = 0 (incluido en el costo inicial); año 2 = $2 257; año 3 = $882; año 4 = $1 262.

[5] Precios al público según www.dell.com al 22 de julio de 2008.

b) **Solo se toma en cuenta el punto de vista del inversionista.** Aunque el modelo es relativamente sencillo, hay un factor "escondido" que de manera tradicional no se toca cuando se hace la descripción del costo-beneficio. ¿Con respecto a qué cliente se debería realizar este análisis? La respuesta en el caso de desarrollo de software debería ser: con respecto a todos los usuarios.

Ejemplifiquemos con el caso de un sistema de recepción de impuestos. Existen al menos dos clientes del sistema: el contribuyente y la tesorería (véase figura 7.4). Además, el área de sistemas es un cliente interno que desarrolla el producto y define los requisitos no funcionales (técnicos). Para cada uno de ellos hay un costo-beneficio (véase cuadro 7.2).

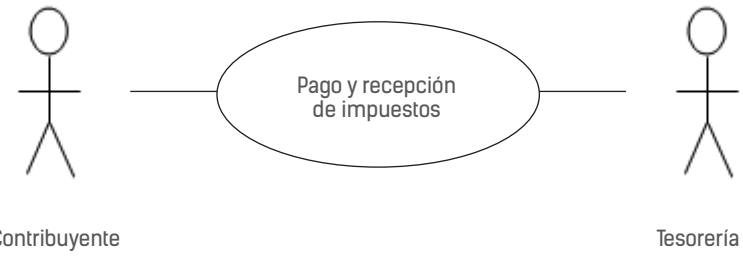


Figura 7.4

Cuadro 7.2

Cliente	Costo de mantenimiento del sistema anterior	Costo inicial del nuevo sistema	Costo de mantenimiento del nuevo sistema
Contribuyente	Tiempo y dinero dedicado para el cálculo y pago de sus impuestos en el sistema anterior.	Tiempo y dinero dedicado para usar el nuevo sistema. Incluye trámites necesarios, capacitación y curva de aprendizaje.	Tiempo y dinero dedicado para el cálculo y pago de sus impuestos en el nuevo sistema.
Tesorería	Todos aquellos costos involucrados en recibir y procesar el pago de impuestos en el sistema anterior.	Todos aquellos costos invertidos en dar a conocer el nuevo sistema: difusión, capacitación interna, capacitación al contribuyente, soporte, ajustes al hardware y software, etcétera.	Todos aquellos costos involucrados en recibir y procesar el pago de impuestos en el nuevo sistema.
Sistemas (cliente interno)	Costo para el mantenimiento del software, hardware y área de soporte para el sistema anterior.	Costos de desarrollo del nuevo sistema, considerando aspectos de desarrollo de software e instalaciones de hardware.	Costo para el mantenimiento del software, hardware y área de soporte para el nuevo sistema.

¿Tendría sentido crear un sistema en el cual el costo de mantenimiento del nuevo sistema es mayor al costo de mantenimiento del sistema anterior? Puede ser, siempre y cuando esto no suceda para todos los clientes. Por ejemplo, el costo de mantenimiento puede aumentar para el área de sistemas, pero bajar para la tesorería y el contribuyente. El desarrollo tendría sentido.

Hagamos hincapié en que el enfoque tradicional solo mide los costos para el área de sistemas. Esa limitada forma de pensar puede traer desde efectos menores hasta una férrea oposición por parte de los otros usuarios del sistema.

¿Qué incluye el costo inicial y de mantenimiento?

No es fácil enumerar los aspectos que definen los costos para los usuarios finales, pues estos dependen mucho del sistema en particular y del contexto en que este se encuentra. Los del área de sistemas en general involucran para el costo inicial:

- Tiempo invertido en el proceso de búsqueda de alternativas, evaluación de las mismas y adquisición.
- Adquisición del software.
- Asesoría.
- Capacitación inicial (tanto del área de desarrollo como del área usuaria).
- Horas de aprendizaje tecnológico de herramientas específicas para el proyecto (curva de aprendizaje).
- Hardware requerido para uso específico del proyecto.
- Horas dedicadas al desarrollo o adaptaciones.
- Horas dedicadas a la configuración y carga inicial.
- Pruebas de funcionamiento.
- Instalación.
- Costo de recuperación del sistema anterior (se le resta a los gastos involucrados).

Erogaciones involucrados en el costo de mantenimiento:

- Póliza de mantenimiento.
- Asesoría y soporte.
- Horas dedicadas por cambios de requerimientos del usuario (por cambios en la organización o requerimientos legales).
- Cambios al sistema por modificación a la plataforma tecnológica (capacitación y migración).
- Actualización tecnológica de los componentes del sistema (hardware y comunicaciones).

EJEMPLO

Ejemplo real de análisis costo-beneficio

Se trabaja en la actualidad con una impresora láser Dell 1110, cuyo costo fue de $1 391.00. El volumen anual de impresión es de 3 000 hojas. El costo del tóner para 2 000 hojas es de $908.00 y el precio de la garantía anual es de $126.00.

Se espera que las condiciones cambien a un volumen de impresión de 48 000 hojas al año. Se tienen consideradas dos opciones de compra:

a) Una impresora Dell 1720 LA, con un costo de $2 816.00, cuyo tóner para 6 000 hojas es de $1 727.00 y la garantía por año es de $220.85.

b) Una impresora Dell 5310n con un costo de $10 677.00, cuyo tóner para 30 000 hojas es de $4 249.00 y la garantía por año es de $1 100.00

¿Cuál es la mejor opción? Para facilitar el cálculo, suponga un costo de papel fijo de $0.07 por hoja y que la impresora no tendrá costo de recuperación.[6]

Concepto \ Impresora	Dell 1110	Dell 1720 LA	Dell 5310n
Costo inicial	No aplica	$2 816.00	$10 677
Costo por página (tóner más papel)	0.52	0.36	0.21
Garantía anual	$126.00	$220.85	$1 100.00
Costo de mantenimiento (costo de impresión más garantía anual)	$25 086	$17 500.85	$11 180.00

a) El tiempo de inversión para la alternativa Dell 1720 LA es:

$2 816 / ($25 086 – $17 500.86) = 0.37 (cuatro meses y medio).

[6] Se emplearon datos del sitioDell a julio de 2008 para tener información realista. El hecho de utilizarlos no constituye una recomendación ni un demerito para la marca. El usuario tendrá que realizar la comparaciòn contra otras alternativas.

b) El tiempo de inversión para la alternativa Dell 5310nes:
$10 677 / ($25 086 – $11 180) = 0.76 (poco más de nueve meses).

Aunque la alternativa A tiene el mejor costo de recuperación, a mediano y largo plazos conviene más la alternativa B. El costo de mantenimiento es más bajo que la alternativa A y el tiempo de recuperación es muy bueno.

Tiempo acumulado \ Opción	Dell 1100	Dell 1720 LA	Dell 5310n
1er. año costo inicial + costo de mantenimiento (el costo inicial para el sistema actual es cero)	$25 086	$20 317	$21 857
2do. año (costo de primer año + costo de mantenimiento)	$50 172	$37 818	$33 037
3er. año	$75 258	$55 319	$44 217

Conclusión: conviene realizar el cambio por la alternativa B. El tiempo de recuperación es alrededor de nueve meses y representa un ahorro de casi 14 000 pesos anuales a partir de ese momento con respecto al sistema actual.

EJERCICIO SUGERIDO

Se trabaja en la actualidad con una impresora, cuyo costo fue de $2 000.00. El volumen anual de impresión es de 6 000 hojas. El costo del tóner para 2 000 hojas es de $908.00, el del papel es de $0.08 por hoja y el precio de la garantía anual es de $126.00. Se espera que las condiciones cambien a un volumen de impresión de 24 000 hojas al año. Se tienen consideradas dos opciones de compra, ninguna de las cuales tendría costo de recuperación:

Una impresora, con un costo de $3 000.00, cuyo tóner para 6 000 hojas es de $2 000.00 y la garantía por año es de $500.00.

Una impresora con un costo de $10 000.00, cuyo tóner para 30 000 hojas es de $4 500.00 y la garantía por año es de $1 500.00.

Con base en los datos anteriores, calcule el costo de mantenimiento del sistema actual, así como el costo inicial y de mantenimiento de las dos opciones mencionadas. Indique su recomendación de compra y los argumentos que la sustentan.

7.5 Factores de calidad del software[7]

Un software debe reunir ciertas características para que su aprovechamiento y funcionamiento sean adecuados. Existen factores **externos** visibles para el usuario final (funcionales). Otras cualidades, como la legibilidad, son perceptibles solo para quienes leen el código fuente. Esos factores son **internos (no funcionales)**. Es importante recalcar que "usuario final" no se refiere solo a quien opera en forma directa el sistema; abarca a las personas que compran el software o adquieren el desarrollo.

Por otra parte, las características del software deben soportar procesos esenciales para la organización; un software puede ser excelente, pero pudiera no corresponder a las necesidades del usuario final.

Cualidades que rodean al software

Estas cualidades no pertenecen al software en sí. Se refieren a la relación del software con la organización que lo utiliza.

- **Coherencia entre el software y los objetivos de la empresa.** El software debe ser prioritario para el área que lo utiliza y contribuir a los objetivos y procesos que esta realiza. Si el software tiene una contribución marginal, por lo común se dejará de usar.

[7] Las partes citadas textualmente fueron tomadas de: Meyer, Bertrand. *Construcción de software Orientado a Objetos*. Prentice-Hall, segunda edición, Madrid, 1999, pp. 3-13.

- **Costo de compra-venta razonable.** El software se debe ofrecer a un precio que sea atractivo para compradores potenciales en razón de la problemática que soluciona y de las ofertas que realiza la competencia a la cual se enfrenta, tanto en su adquisición como en su costo de mantenimiento.

Pensando en la compra-venta:

- Antes de adquirir un producto, determinar con claridad las características indispensables que debe cubrir; desglosar todas las actividades que implican tener a punto el sistema (compra-venta del software, licencias adicionales, asesoría, capacitación) e incluirlas en el presupuesto de arranque; analizar las posibilidades de crecimiento del software, así como todos los costos involucrados con el mantenimiento, y realizar un análisis comparativo entre las opciones que cubran las funcionalidades indispensables.
- Siempre considerar opciones de software libre y propietario. La elección dependerá de un análisis costo-beneficio de las diversas alternativas.

Garantía de soporte y actualización. El proveedor del software debe garantizar el soporte oportuno durante todo su tiempo de vida, así como sus adecuaciones por cambios tecnológicos, de ley o por nuevos requerimientos de la empresa. Muchas de estas actualizaciones se pueden hacer vía internet.

Pensando en la garantía de soporte y actualización:

- En caso de proveedores externos, medir el costo de mantenimiento, asesoría y soporte antes de la compra del producto.
- En caso de desarrollos internos, estimar el costo de mantenimiento, con base en el tiempo dedicado a los cambios del producto.
- Garantizar que la realización del sistema sea en tecnologías que no estén a punto de salir del mercado (independientemente de si se trata de una compra o de software propio).
- Buscar mecanismos (visitas a las instalaciones, cartera de clientes, etc.) para medir si los proveedores tienen cierta garantía de permanencia en el mercado y la experiencia real de los usuarios, así como esquemas que eviten la dependencia con proveedores.

Cualidades del software como producto

"Funcionalidad es el conjunto de posibilidades que proporciona un sistema."

Para lanzar la versión de un producto se debe cubrir un área lo bastante amplia del conjunto completo de características para atraer a los consumidores previstos en lugar de alejarlos. De seguro se dejarán de lado algunas características, pero las que se decida incluir deben cubrir todas las características de calidad. "Hay que rechazar pasar a considerar nuevas propiedades hasta que no se esté satisfecho con las que se tiene."

Para solucionar el problema de mantener la calidad y avanzar lo más posible en ofrecer más facilidades, debe trabajarse "una y otra vez sobre la consistencia del producto global, tratando de que todo encaje en un molde general. Un buen producto de software se basa en un número pequeño de ideas potentes: incluso si tiene muchas propiedades especializadas, estas deberían explicarse como consecuencia de los conceptos básicos". El "gran plan" debe estar visible y todo debería ocupar su sitio dentro de él.

Para verificar la funcionalidad:

- Como proceso de verificación, contrastar el producto obtenido contra el listado de la funcionalidad requerida por el sistema, según las especificaciones establecidas.

"**Corrección** es la capacidad de los productos de software para realizar con exactitud sus tareas [sin errores], tal y como se definen en las especificaciones.

"La corrección es la cualidad principal. Si un sistema no hace lo que se supone que debe hacer, poco importan el resto de las consideraciones que hagamos sobre él —si es rápido, si tiene una bonita interfaz de usuario…"

Para llegar a la corrección se requiere como primer paso especificar los requisitos del sistema en una forma precisa, lo cual no es sencillo. Este paso requiere de diversas técnicas de ingeniería de software. En esta parte el usuario final está involucrado de manera directa.

En una segunda etapa, se trata de construir el software que cumpla con dichas especificaciones. Lo cual corresponde más a temas de lenguajes de programación y bases de datos. Son rubros en donde prácticamente no interviene el usuario final.

Hay que hacer hincapié en que se construye un software bajo el supuesto de que los niveles inferiores son correctos. Por ejemplo: quien está elaborando un programa sobre desviación estándar parte del hecho de que las funciones de raíz cuadrada realizadas por el compilador son correctas. Por supuesto, nunca hay una garantía absoluta en ello, pero es la única forma realista de trabajar.

Pensando en la corrección:

- Tener un documento que identifique las funcionalidades que debe tener el sistema en la versión actual y aquellas que se dejarán para versiones futuras.
- Establecer desde etapas tempranas del proyecto los requerimientos no funcionales (estándares de nombres de archivos, campos y colores; forma de trabajar la resolución; forma de codificar el acceso a base de datos; normas de robustez; normas de seguridad; etcétera).
- Tener un plan de implantación de funcionalidades pendientes.
- Detallar las necesidades específicas del módulo (reporte, cálculos, etc.) con casos típicos y especiales antes de proceder a su codificación.
- Probar un sistema con casos representativos y con casos especiales que cubran todas las funcionalidades del sistema.
- Revisar los distintos parámetros de configuración del sistema.

"**Robustez** es la capacidad de los sistemas de software de reaccionar en forma apropiada ante condiciones excepcionales.

"La robustez complementa la corrección. La corrección tiene que ver con el comportamiento de un sistema en los casos previstos por su especificación; la robustez caracteriza lo que sucede fuera de tal especificación." Es decir, lo que sucedería en casos excepcionales, entendiendo por "excepcional" un caso no previsto por la especificación y por "normal" algo "planificado durante el diseño del software".

Una forma práctica de abordar la robustez es garantizar que el sistema no termine su ejecución en forma abrupta en todos aquellos casos anómalos que de antemano se sabe pudieran suceder. Por ejemplo, que se intente grabar cuando no hay espacio suficiente en el disco; que se intente copiar a una unidad de disco inexistente; que se dañe una tabla en la base de datos; que se intente realizar una división entre cero; que el saldo neto de una quincena de un trabajador resultara en negativo al aplicar todos los descuentos autorizados, etc. En muchos lenguajes de programación ya existen estructuras que manejan en forma automática estas excepciones.

En procesos interactivos (por ejemplo, pantallas de captura) se debe retroalimentar al usuario de manera inmediata; en procesos batch (por ejemplo, carga de datos, cálculo de nómina) deberá existir un reporte final que especifique si hubo casos que no se procesaron por problemas de información y la acción que se tomó.

En procesos que pudieran ser interrumpidos —instalaciones, transferencias a tablas históricas— se debe procurar que el proceso se reinicie en el punto en que se quedó o, al menos, que pueda repetirse sin generar problemas de duplicación o carencia de información.

Pensando en la robustez:

- Las pantallas de captura deben detectar que hayan sido llenados todos los campos necesarios y los datos inconsistentes (por ejemplo, datos con letras cuando se espera un dato numérico, fechas ilógicas, etcétera).
- Al nivel de la programación, poner validaciones en situaciones que pueden provocar errores (por ejemplo, leer datos con valores nulos y divisiones entre cero).
- Alimentar el sistema con información excepcional previsible. Por ejemplo, que el saldo neto de una quincena de un trabajador resulte negativo después de aplicar todo los descuentos.
- Llevar el sistema a condiciones anormales previsibles. Por ejemplo, que la red no trabaje de manera adecuada.
- Interrumpir el sistema en procesos que pudieran quedar inconclusos por situaciones diversas (por ejemplo, procesos de instalación) y verificar que se pueden ejecutar de nuevo sin ningún problema.
- Verificar que los procesos batch del sistema (carga, cálculos, alimentación de tablas históricas, etc.) reportan de manera adecuada casos anómalos y tienen formas de corregir el problema.
- Establecer procedimientos que impidan crear inconsistencias a los sistemas, tanto de índole técnica (por ejemplo, replicaciones automáticas, rutinas de cambios "en cascada" a claves usadas en el sistema) como administrativas (por ejemplo, que el departamento de base de datos valide cualquier diseño para verificar que no existe información redundante entre los distintos sistemas).

"**Facilidad de uso** es la posibilidad de que personas con diferentes formaciones y aptitudes puedan aprender a usar los productos de software y aplicarlos a la resolución de problemas. También cubre la facilidad de instalación, de operación y de supervisión."

Uno de los problemas de la facilidad de uso es: "¿cómo proporcionar explicaciones y guías detalladas a los usuarios novatos sin fastidiar a los usuarios expertos que quieren ir directo al grano?".

Debe construirse un buen diseño, de acuerdo con una estructura clara y pensada en función del usuario final. Debe considerarse que un usuario final puede ser alguien diestro en el manejo del sistema y en el área que aborda el software, pero también alguien que toma el software por primera vez y no tenga casi ningún conocimiento del tema. Por ejemplo, puede ser un experto en nómina que lleva manejando este software tres años, o un capturista que lo comenzó a utilizar hace un par de horas. Un sistema sencillo e intuitivo no está peleado con un sistema que se puede manejar con rapidez y que tiene opciones avanzadas.

Lo anterior, implica tener ayuda documental clara, completa y concisa, tanto en línea como remota. Los ejemplos mostrados deben tener datos representativos del sistema. En la ayuda nunca deben usarse datos que solo cubran el aspecto de sintaxis. Por ejemplo, RFC = AAAA000000. El texto del manual de usuario debe estar en un lenguaje que sea comprensible para el usuario y, de preferencia, realizado por un analista cercano a la problemática de este y no por el desarrollador.

Pensando en la facilidad de uso:

- Hacer pruebas de funcionamiento a pantallas prototipo antes de generalizar su uso.
- Dar las instrucciones de uso necesarias en las pantallas considerando el tipo de usuario que las manejará.
- Realizar pruebas de funcionamiento con un usuario representativo e independiente de los desarrolladores.
- Verificar la ayuda en línea del sistema y la existencia de un manual de usuario, los cuales deben ser llenados con datos que simulen ejemplos reales de funcionamiento del sistema.
- Los reportes y consultas deben poder limitarse al menos por los parámetros más usuales que necesita el usuario (fechas, sucursales, etcétera).
- Uniformar el formato de las pantallas (colores, tipos de menú, etcétera).

El software, además de ser funcional y fácil de usar, debe ser **visualmente atractivo**, pues de esta manera contribuirá a que el usuario se sienta motivado a usarlo, con independencia de que lo haga por iniciativa propia o por necesidad laboral.

Pensando en lo atractivo del producto:

- Respetar la imagen institucional en los productos de software.
- Apoyarse en diseñadores gráficos que brinden mejoras de imagen desde la creación de pantallas prototipo.

"**Compatibilidad** es la facilidad de combinar unos elementos de software con otros."
Al menos hay dos enfoques para garantizar la compatibilidad:

- Una prueba directa en un ambiente de pruebas con el software que va a convivir en forma simultánea. Si apenas se desarrollará el software se puede hacer una prueba preliminar con algunas rutinas del nuevo sistema.
- Un análisis de los diferentes estándares y protocolos manejados por los sistemas para observar que existe compatibilidad (formatos de archivo, estructuras de datos, interfases de usuarios, etcétera).

Debe destacarse que muchos problemas de compatibilidad se resuelven si se cambian diversos parámetros de configuración en uno o varios de los sistemas que se ejecutan en forma simultánea.

También puede darse el caso de que no pueda solucionarse el problema de compatibilidad, pero no se requiere que ambos sistemas trabajen al mismo tiempo. Dejar esta situación así hasta que el software sea sustituido al paso del tiempo es una solución práctica en muchísimos casos.

Pensando en la compatibilidad:

- Analizar los requisitos de instalación y configuración de los sistemas que convivirán en una misma instalación para prever posibles problemas, antes de comprar un sistema o iniciar un desarrollo.
- Poner a ejecutar el sistema en combinación con todos los otros sistemas con los cuales convivirá.

"**Portabilidad (transportabilidad)** es la facilidad de transferir los productos de software a diferentes entornos de hardware y software.

"La portabilidad tiene que ver con las variaciones no solo del hardware físico sino de modo más general de la **máquina hardware-software**, la que en realidad programamos y que incluye el sistema operativo, el sistema de ventanas (si se emplea) y otras herramientas fundamentales."

En este rubro debe tomarse en cuenta desde un principio si el software que se ofrece se ejecutará en una computadora personal, una red LAN, en internet o en cualquier otro tipo de plataforma.

Un enfoque preventivo a problemas de portabilidad implicaría probar el sistema en los diferentes entornos a los cuales se enfrentará y establecer estándares que se ejecuten en todos los entornos esperados. Pueden destacarse: que los diseños trabajen en cualquier resolución de monitor; que el estándar de mayúsculas y minúsculas de la codificación trabaje en los diferentes sistemas operativos y sistemas manejadores de bases de datos esperados; que las sintaxis de los lenguajes se ejecuten en las diferentes versiones de compiladores que se usarán, etcétera.

Pensando en la portabilidad:

- Establecer, desde el inicio del desarrollo, los entornos en que se ejecutará el sistema (sistemas operativos, sistemas manejadores de bases de datos, navegadores, resoluciones de monitores), tanto en lo referente a marcas como versiones.
- Ejecutar el sistema en los diferentes entornos de sistemas operativos en que debe trabajar (UNIX, Linux, Windows).
- Ejecutar el sistema en las diferentes resoluciones de monitor en las que deberá trabajar.
- Probar los estándares de programación en las diferentes herramientas (versiones, entornos integrados, etc.) en que se espera que se ejecute el sistema.

"**Seguridad (integridad)** es la capacidad de los sistemas de software para proteger sus diversos componentes (programas, datos, etc.) contra modificaciones y accesos no autorizados. Esta capacidad se basa —entre otras formas— en políticas de claves de acceso y algoritmos de encriptamiento.

Pensando en la seguridad (integridad):

- Desde el inicio del sistema, establecer las políticas de seguridad, tanto administrativas como técnicas, a seguir para los desarrollos del sistema.
- Tener definidas con claridad las matrices de acceso por perfil.
- En el caso del navegador web, no mostrar información que facilite accesos no autorizados (direcciones IP, directorios reales, claves de acceso, etcétera).
- Los datos críticos de las bases de datos deben estar encriptados.
- Verificar que no se pueda acceder en forma directa a las páginas web; solo después de teclear el usuario y contraseña, o algún otro mecanismo equivalente.

"**Eficiencia** es la capacidad de un sistema de software para exigir la menor cantidad posible de recursos de hardware, como tiempo del procesador, espacio ocupado de memoria interna y externa o ancho de banda utilizado en los dispositivos de comunicación."

Con respecto a la eficiencia se puede caer en dos actitudes opuestas (y ambas equivocadas): restarle importancia, porque cada día las computadoras son más rápidas; o tener una obsesión por las cuestiones de rendimiento, y por ello tratar de optimizarlo al máximo.

Una de las formas de mejorar la eficiencia es incrementar la ventaja de los buenos algoritmos sobre los malos. Por ejemplo, supongamos que para sacar un reporte se leen uno por uno medio millón de registros de un archivo histórico. Otro desarrollador hace una pequeña rutina que combina las condiciones dadas por el usuario para traer –en promedio–unos 2000 registros. En este caso, el tiempo de procesamiento disminuye mucho más por usar un buen algoritmo que por emplear una computadora con mayores recursos.

Ahora bien, optimizar un proceso muchas veces implica un mayor esfuerzo de programación y a veces impacta de manera negativa en la claridad y en la reutilización. La mayoría de los procesos no se ejecutan en forma cotidiana y por ese motivo —o por otros motivos igual de válidos— no requieren ese esfuerzo extra. Por el contrario, hay algunas rutinas que por su uso frecuente ameritan ser optimizadas al máximo. El desarrollador debe hacer un balance entre la eficiencia y el tiempo de desarrollo.

También vale la pena recalcar que el uso adecuado de instrucciones (de consulta a bases de datos y diseño y elección de las subrutinas a utilizar) puede incrementar la eficiencia sin hacer ningún esfuerzo extra e, incluso, al disminuir la complejidad y el tiempo de desarrollo.

Pensando en la eficiencia:

- Ejecutar el sistema con diferentes volúmenes de datos (10, 100, 1 000, 10 000) para tratar de establecer su comportamiento (función de trabajo) y anticipar su funcionamiento con altos volúmenes de información.
- Probar el sistema con el nivel máximo de información al cual se espera llegar a mediano plazo (pruebas de estrés o carga máxima), de ser posible con datos reales o, al menos, con datos simulados.
- Probar el sistema en las condiciones mínimas de recursos en los cuales trabajará para verificar que funciona en forma aceptable (conexión a internet a 56 kbps, memoria RAM mínima recomendada).
- Realizar programas de simulación que brinden condiciones lo más cercanas posible al sistema real trabajando, para verificar que el sistema trabaja de manera adecuada.
- Establecer recomendaciones y procesos que impidan que se carguen imágenes o archivos fuera de los tamaños recomendados.

Características del software como proceso de desarrollo

"**Oportunidad** es la capacidad de un sistema de software de ser lanzado cuando los usuarios lo desean, o antes."

La oportunidad es una de las mayores frustraciones de nuestra industria. Un gran producto de software que aparece demasiado tarde puede no alcanzar su objetivo. Esto es cierto también en otras industrias, pero pocas evolucionan tan rápido como el software… la oportunidad todavía es para proyectos grandes [y pequeños], un fenómeno poco común.

Pensando en la oportunidad:

- No realizar compromisos de tiempo sin tener un análisis del tiempo real de desarrollo.
- Basar las fechas de compromiso en las horas que los desarrolladores dedicarán en realidad al proyecto.
- Establecer un seguimiento de la planificación y la forma en que el avance se reporte con bases medibles y verificables, en particular de la ruta crítica del proyecto.

"**Economía**, junto con la oportunidad, es la capacidad que tiene un sistema de completarse con el presupuesto asignado o por debajo del mismo."

Pensando en la economía:

- Medir los tiempos de desarrollo con base en estimaciones realistas (desglose de módulos y experiencia histórica).
- No subestimar la "curva de aprendizaje" por productos nuevos o personal nuevo.
- Medir el desarrollo del proyecto con bases objetivas y verificables.
- Establecer mecanismos para el cobro de funcionalidades adicionales a las establecidas.

Características no funcionales del software

La **facilidad de mantenimiento** implica que el software realizado pueda adaptarse con relativa facilidad a nuevos requerimientos. Para cumplir esta cualidad deben cumplirse varios aspectos, tanto a nivel comercial como de diseño del software y codificación:

- Que el software sea **configurable**. Todos los cambios con alta posibilidad de cambiar a lo largo de la vida del sistema deben ser modificables por un usuario autorizado (cambios de claves de acceso, altas de nuevos usuarios, cambios de tablas de impuestos, etcétera).
- Que el software sea **modular** a nivel código, con alta cohesión y poco acoplamiento; es decir, construido con elementos (clases, paquetes, drivers, etc.) que realicen tareas completas de modo lo más independiente posible. Un software modular facilita el desarrollo, mantenimiento y reutilización del código.
- Que el software esté **documentado**. Esto abarca los aspectos de instalación y operación, así como toda aquella documentación que facilite la modificación del código. La documentación debe estar encaminada a cubrir los siguientes rubros:
  - Su instalación en los diferentes entornos en que pueda trabajar (manual de instalación).
  - Su operación cotidiana y en casos especiales (manual de usuario).
  - En caso de tenerse el código fuente, todos aquellos elementos para que este pueda modificarse según requerimientos nuevos del usuario o entornos nuevos de hardware y software (análisis de requerimientos; documentos de diseño; diseño de bases de datos; código fuente actualizado; etcétera).
- Con **garantías de mantenimiento**, cuando todo el código o parte de él no pertenecen a la organización.
- "**Reparabilidad** es la capacidad para facilitar la reparación de los defectos."

"Verificabilidad es la facilidad para preparar procedimientos de aceptación, en especial datos de prueba y procedimientos para detectar fallos y localizar errores durante las fases de validación y operación."

Pensando en la facilidad de mantenimiento:

- Incorporar la generación de la documentación a los procesos de desarrollo.
- Si existen datos con alta probabilidad de modificarse a mediano plazo, deben hacerse con facilidad por el usuario o, al menos, por el equipo de soporte.
- No debe existir redundancia de la información. Un dato debe tener una única fuente.
- Crear rutinas genéricas de validación de datos y manejo de permisos.
- Generar la documentación necesaria para garantizar el mantenimiento de los sistemas. La documentación técnica debe abarcar, al menos:
  - Contrato inicial para el desarrollo del sistema o equivalente.
  - Listado de las funcionalidades generales del sistema, tanto de aquellas en desarrollo, como las que están en producción o se dejaron para versiones posteriores.
  - Diagramas generales del sistema para entender su funcionamiento global e información detallada que permita su mantenimiento (listado de programas, diccionario de campos, explicación de rutinas genéricas, etcétera).
  - Un documento que especifique la forma de instalación, tanto en máquinas cliente como en el servidor.
  - Manual de usuario que explique la operación y configuración del sistema a nivel usuario.

Pensando en la verificabilidad:

- Establecer pistas de auditoría que brinden información clara y detallada sobre los orígenes de posibles errores.
- Establecer una comunicación automática con el área de soporte cuando se genere un problema o cuando se dé una situación fuera de los parámetros aceptables.
- Establecer rutinas automáticas (por ejemplo, disparadores en la base de datos) que brinden pistas de auditoría respecto a operaciones delicadas referentes a la configuración o a los datos del sistema.
- Establecer mecanismos administrativos y técnicos que permitan documentar cambios a los sistemas desarrollados.

Pensando en la reparabilidad:

- Tener un proceso de respaldo establecido y validado.
- Tener procesos redundantes para procesos críticos (doble fuente de poder, arreglos de discos duros, servidores espejo, etcétera).
- Establecer procesos de replicación que eviten inconsistencias de información en casos de caída de red.

7.6 Importancia de la metodología para el desarrollo de software

A esta altura, esperamos no haber asustado demasiado a quien lea este libro. El desarrollo de software se trata de producir software de calidad (véase sección 7.5) orientado hacia los distintos tipos de usuarios finales (véanse sección 7.3), para quienes debe haber un adecuado costo-beneficio (véase sección 7.4). Dicho software debe enmarcarse de manera adecuada dentro de los diferentes elementos de un sistema de información para cumplir su objetivo (véase sección 7.2) y, sobre todo, para que no se convierta en un BANO (véase sección 7.1). Para lograr este propósito debe recurrirse a conocimientos de programación estructurada (véanse capítulos 1 a 3) y programación orientada a objetos (véanse capítulos 4 a 6), entre otros paradigmas.

Para quien se sienta abrumado, un comentario más: este es un libro introductorio al campo de la programación. Estamos lejos siquiera de haber llegado a la mitad del camino tanto en aspectos de programación como metodológicos. ¿Conclusión? El desarrollo de software es una rama interdisciplinaria del conocimiento que debe tomarse con respeto y que no debería permitir improvisaciones ni trabajo desordenado.

¡Y claro que solemos cometer errores básicos! Aún más, ni estudiantes ni maestros estamos conscientes de ello. La siguiente es una lista de algunos síntomas típicos que se presentan cuando no se ha seguido el ci-

clo de desarrollo de sistemas, basado en el método de cascada. En diversos cursos los alumnos (estudiantes, profesores y egresados de escuelas privadas y públicas) han identificado entre 38 y 43% de estas carencias en los sistemas con los que han tenido contacto en el último año. De hecho, todas las etapas se acercaron al promedio y todas las opciones fueron marcadas, aunque algunas con menor frecuencia que otras.[8]

Autorización

( ) No se tiene un contrato o plan de trabajo por escrito al arrancar el proyecto.

( ) Se habían realizado varias pláticas para promover el proyecto y se descubrió que la persona no tenía un interés real o no tenía una influencia significativa en la decisión.

( ) Ya se tenían avances significativos en el proyecto y no lo autorizaron.

( ) No se hizo una estimación de tiempos con bases sólidas.

( ) No se hizo una estimación de costos con bases sólidas.

( ) El contrato no indica los costos y alcances del mantenimiento.

( ) No se consideró la "curva de aprendizaje" en los tiempos y costos.

( ) No se consideró la compatibilidad entre versiones.

Análisis

( ) No hay documentos que expliquen los procesos de cálculo.

( ) Se presentan ambigüedades o puntos encontrados acerca del significado de varios términos en etapas posteriores.

( ) En etapas posteriores se agregan muchos reportes que no estaban considerados.

( ) En etapas posteriores se agregan módulos completos que no estaban considerados.

( ) Cuando se le presenta el proyecto al usuario dice que no era lo que él había pensado.

( ) Los reportes y consultas no se pueden limitar por los parámetros más usuales que necesita el usuario (fechas, sucursales, etcétera).

( ) Cada programador le da al sistema una presentación diferente (colores, menús, etcétera).

( ) El dueño o el directivo no respetan las decisiones del líder de proyecto.

Diseño

( ) No existe un diagrama general del sistema.

( ) Al hacer la prueba integral se presentan problemas porque hay campos o rutinas que cada programador manejó con nombres distintos cuando deberían tener un solo nombre.

( ) La consulta de los nombres y tipos de campos es muy lenta.

( ) Hay varias partes del código que hacen prácticamente lo mismo y se programaron varias veces.

( ) Se dificulta encontrar los nombres de programas o tablas porque no existe un estándar.

( ) Los programas corren bien pero son demasiado lentos.

( ) El sistema permite la entrada de datos inconsistentes.

( ) El sistema en general no valida datos típicos (por ejemplo, fechas).

( ) El sistema no permite modificar un dato que iba a cambiar a mediano plazo. Por ejemplo, el monto del salario mínimo general.

[8] Las encuestas fueron aplicadas a aproximadamente 200 estudiantes de un curso intermedio de base de datos entre 2009 y 2011 en México, sin que se notara diferencia entre quienes comenzaban en el medio laboral y quienes ya tenían experiencia, vinieran de instituciones públicas o privadas. Aunque no puede hablarse de una muestra representativa, el resultado parece consistente con diversas experiencias en el campo de las tecnologías de información en el país.

Codificación y pruebas modulares

( ) Cuando se va a probar un módulo no se tiene prediseñado un lote de pruebas.
( ) Los programas no están comentados.
( ) No se aplicaron estándares de presentación ni estándares técnicos.
( ) Los programas "truenan" o arrojan resultados incorrectos.
( ) Un programa en particular no valida datos típicos (por ejemplo, fechas).
( ) Faltas de ortografía.

Pruebas integrales

( ) No se hizo ninguna prueba integral.
( ) Las pruebas integrales no se hicieron con datos reales.
( ) No se hizo el manual de instalación.
( ) No se probó el procedimiento de instalación.
( ) No se hizo el manual de usuario.
( ) La carga de datos se hizo en forma manual, lo cual implica volver a repetirla de la misma forma en el futuro.
( ) No se probó el procedimiento de instalación.

Mantenimiento

( ) No se actualizó la documentación.
( ) No se hicieron rutinas para conservar la coherencia de datos históricos.
( ) No hay ningún soporte documental.
( ) Se "traspapeló" la documentación técnica o de usuario del sistema.
( ) Se "traspapelaron" los discos de instalación.
( ) No existen respaldos probados.

Ejemplos reales de errores metodológicos:[9]

REFLEXIÓN 7.9

Un sistema escolar con materias fijas

En un sistema de control de asignaturas se observó un código parecido al siguiente.

```
if materia = "ADMI" then nombre_materia = "administración"
if materia = "PROG" then nombre_materia = "programación"
```

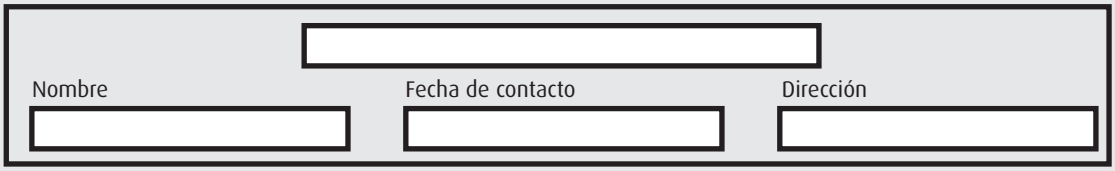
El listado seguía con 30 materias más. Este tipo de código trae problemas futuros. ¿Es posible que en los próximos 10 años cambie la clave o el nombre de una asignatura? La probabilidad es casi de 100%. Entonces estos datos deben estar en un catálogo que el usuario pueda modificar. De lo contrario será necesario alterar el código fuente, probar, actualizar la documentación y poner al día todas las instalaciones en las que se encuentre el programa. El ahorro de unas dos horas nos causará a corto o mediano plazo tener que invertir al menos un par de días, sobre todo si existen decenas o centenares de instalaciones.

[9] Los tres casos presentados en este capítulo se basan en experiencias personales reales. Se omitieron las empresas por motivos de confidencialidad.

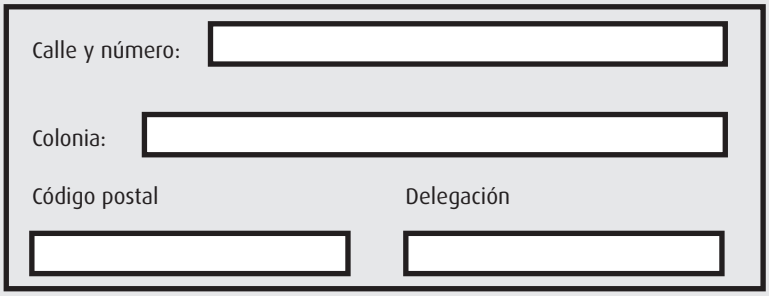
REFLEXIÓN 7.10

Reproceso en el envío de correspondencia

Una empresa deseaba tener un listado de compradores potenciales, el cual capturó por delegación política. Su listado era similar al siguiente:



Cuando llevaban 5500 registros decidieron enviarles correspondencia; el problema es que para hacerlo se necesita el siguiente formato.



Y la dirección se había capturado en un solo campo. ¿Qué implicaba esto?

a) Crear nuevos campos en la base de datos.
b) Modificar pantallas de captura y reportes.
c) Recapturar la dirección de las 5 500 personas.

El error desde el punto de vista práctico fue no separar los campos. Desde el punto de vista conceptual fue mucho mayor: no se siguieron las etapas del ciclo de desarrollo de sistemas, por ello nunca se previeron necesidades futuras inmediatas. Una falla en el análisis provocará cambios posteriores en todas las demás etapas, con sus repercusiones anímicas y económicas.

En este ejemplo, la captura se hizo por segunda vez y tal vez eso consumió dos meses. ¿Qué hubiera sucedido si un error de tal magnitud hubiera surgido en un sistema de $10 000 USD?

REFLEXIÓN 11.3

Un sistema de recaudación con múltiples errores

Una oficina gubernamental publicó vía internet en 2001 un software para capturar los datos de la Declaración de Sueldos y Salarios. El programa fue publicado con una semana de anticipación; el dato que se capturaba en sueldo normal aparecía en el reporte como aguinaldo; las pantallas aparecían cortadas en monitores tipo VGA (en aquel entonces una cuarta parte de las PC no podía poner una resolución mayor) y los ejemplos en el manual no mostraban datos reales.

Una situación así solo puede explicarse —entre otros muchos motivos— porque nunca se siguió una metodología para el desarrollo de este producto. Existieron errores en todas las etapas del desarrollo.

Suponemos que hemos dado argumentos suficientes para sustentar que la falta de una metodología parece ser el "talón de Aquiles" de las áreas de desarrollo de software y de los centros de educación superior. Rara es la institución que tiene un marco conceptual coherente para esta área y se llega al extremo —por cierto, muy frecuente— de fomentar su menosprecio. Para algunos, la documentación y la metodología son "rollos" y pérdida de tiempo.

Además, no es sencillo tener claridad ante la cantidad de sugerencias que se van encontrando (véase cuadro 7.3).

Cuadro 7.3 Algunas "sugerencias" involucradas en el desarrollo de software

Modelos de ciclo de vida	Prototipos
	Versiones sucesivas
	Desarrollo incremental
Metodologías de desarrollo	Cascada
	Programación extrema
	Proceso unificado
Normatividades	ISO
	CMMI
	MoProSoft
Técnicas administrativas	Costo-beneficio
	Benchmarking
	Administración de tiempos
Metodologías de gestión de trabajo	Scrum
	ITIL

Antes de pretender adoptar una metodología, debe tenerse claridad sobre un aspecto básico: ¿qué se puede esperar de ella en realidad? Los propósitos generales son:

- Garantizar que los sistemas de información respondan a las necesidades estratégicas de la empresa con base en los distintos tipos de usuario, y que disminuya la posibilidad de ser un apoyo por completo marginal o un BANO.
- Garantizar que los productos realizados cumplan con los requisitos que debe poseer un buen software. Esto incluye los requisitos establecidos de común acuerdo con las áreas usuarias (requisitos funcionales) y aquellos relacionados con la estructura interna del sistema (requisitos no funcionales).
- Fomentar que los sistemas sean desarrollados por equipos de trabajo flexibles, que integren en forma dinámica los distintos conocimientos y habilidades del personal según las necesidades específicas de cada proyecto en cada una de sus etapas. En todo momento, al menos dos personas deben estar enteradas del desarrollo y manejo de un sistema.
- Fomentar la capacitación continua, considerar los planes generales y los intereses y motivaciones del propio personal.
- Que la administración de los proyectos se haga sobre bases viables en tiempo y costos, se minimicen los riesgos de fallas y se permita la adecuación ágil de planes de trabajo ante situaciones cambiantes.
- Generar el soporte documental para certificaciones e informes a instancias superiores, integrar su elaboración al propio desarrollo del sistema.

Mientras los criterios para elegir e implantar una metodología son:

- Aplicable en el contexto diario. La metodología elegida e implantada debe ser acorde a las necesidades concretas del área de sistemas. No pueden elegirse técnicas que no se adapten a los requerimientos.
- Viable. Que se pueda llevar a la práctica con los recursos y normatividades actuales, considerar un costo-beneficio correcto. Hay mecanismos para lograr cambios en los hábitos de trabajo.
- Consensuada. Deben tomarse en cuenta los diversos involucrados (responsables de atención a usuarios, desarrolladores, líderes de proyectos, etc.). De otra forma, no cubrirá necesidades importantes y generará rechazos.
- Incremental. Casi nunca puede darse un salto gigantesco de un día para otro. En general, conviene tener recomendaciones básicas y metas claras en breve tiempo y que a mediano plazo conduzcan a una buena situación. Cada avance debe aportar un valor agregado.
- Documentación mínima. La documentación que se genere debe ser la indispensable y formar parte integrante de cada una de las etapas. Nunca debe ser un proceso independiente.
- Obligatoria. La documentación acordada será de carácter obligatorio.

Seamos claros: en este libro no se proporcionan todos los elementos para enfrentar un propósito tan grande, pero sí se consigue un acercamiento para enlazar las partes metodológicas a los distintos cursos de programación.

7.7 Un primer acercamiento metodológico: el método de cascada

Un método que aún está vigente es el método de cascada. De hecho, fue la sugerencia para el tema de programación estructurada (capítulos 1 a 3). En este método el desarrollo se divide en etapas que deben darse en forma secuencial. En el caso de que se necesiten modificaciones se seguirá este método para los cambios a realizar.

En el cuadro 7.4 se sintetizan las etapas del método de cascada. Conviene hacer hincapié en que la documentación no es una etapa, se hace a la par que avanza el proyecto. Por otra parte, se incluye la autorización del proyecto, que casi ningún autor considera como parte del método de cascada. Sin embargo, la realidad cotidiana nos empuja a visualizarla como la base de las actividades siguientes.

Como se notará con facilidad, la secuencia es por demás lógica y si se sigue de manera disciplinada contribuye en gran medida al resultado deseado. El problema es que las pruebas se realizan hasta el final y si se detecta un error catastrófico ya se consumió prácticamente todo el presupuesto. Por otra parte, los requisitos se encuentran desde el inicio; ¿qué sucedería si el usuario no tiene claridad sobre el sistema? Por ello el método de cascada es ideal para ambientes predecibles a nivel tecnológico y de requerimientos; la mejor recomendación si no se sale de este contexto.

Cuadro 7.4

Etapa	Objetivo y descripción de la etapa	Documentación entregada	Porcentaje aproximado del tiempo de desarrollo
Autorización del proyecto	**Que se autorice el proyecto:** alcance, tiempos y recursos. **Rubros a cubrir:** evaluación de alternativas; estimación de tiempos; estimación de costo-beneficio; establecer el contenido general; definición de plataformas en que debe ejecutarse. **Técnicas utilizadas:** entrevistas con el directivo, consultas a especialistas, herramientas de planeación (gráficas de Gantt, ruta crítica), etcétera.	Contrato	10
Análisis	**Definir los requerimientos a detalle del nuevo sistema** **Rubros a cubrir:** resultados que arrojará el nuevo sistema (incluyendo reportes); entradas que alimentarán al nuevo sistema (incluyendo pantallas); documentación de cálculos; glosario de términos y estándares de presentación; readecuación de procesos administrativos. **Técnicas utilizadas:** análisis del sistema actual; revisión de reglamentación vigente; entrevistas con el usuario operativo; diagrama de flujo de datos, etcétera.	Especificación de requerimientos	20
Diseño	**Arquitectura general del producto** **Rubros a cubrir:** diseño modular; normalización de bases de datos; script de creación de bases de datos; diccionario de campos; estándares técnicos; algoritmos principales. **Técnicas utilizadas:** pruebas de portabilidad; diseño modular (top-down); reglas de normalización de base de datos; complejidad computacional; explotación de las tablas del sistema de la base de datos, herramientas automatizadas de creación de scripts (por ejemplo, Erwin), etcétera.	Manual técnico	15
Codificación y pruebas por módulo	**Construcción del producto conforme a especificaciones** **Rubros a cubrir:** diseño de lote de pruebas; descripción de rutinas genéricas; estándares de programación; código fuente comentado y probado. **Técnicas utilizadas:** diseño de lote de pruebas; seudocódigo; diagramas de sintaxis; enfoque de caja blanca y caja negra; lenguajes de programación; etcétera.	Código fuente comentado	25
Pruebas integrales	**Prueba integral del producto** **Rubros a cubrir:** diseño de lote de pruebas integral; manual de usuario; manual de instalación (incluye discos de instalación). **Técnicas utilizadas:** carga de datos reales; creación de discos de instalación; pruebas en paralelo; pruebas de carga máxima; prueba integral; revisión de la configuración del nuevo sistema; pruebas de control de accesos y de seguridad en general.	Discos de instalación y manuales	12
Carga y puesta a punto	**Puesta en funcionamiento del sistema** **Rubros a cubrir:** capacitación; cartas de liberación. **Técnicas utilizadas:** carga de datos reales; formas de instalación masiva; formas de soporte cercano a nuevos usuarios.	Adecuaciones por corrección de errores o nuevos requerimientos.	18
Mantenimiento	**Mantener al sistema en un funcionamiento correcto** **Rubros a cubrir:** modificaciones de funcionamiento; adecuación de datos históricos. **Técnicas utilizadas:** contratos de mantenimiento; establecimiento de estándares.	Actualización del sistema con su respectivo soporte documental.	De dos a cuatro veces del desarrollo original.

EJERCICIOS SUGERIDOS

- Un líder de proyecto indicó capturar el nombre completo, dirección y teléfono de clientes potenciales en un archivo tipo texto; cada campo con una longitud fija. Ejemplos:

José Luis Huerta Jiménez	Iris #38, Col. Obrera	55-68-84-27
Juan Hernández López	Té 48, Unidad FOVISSSTE Miramontes	56-91-23-56

  El programador realizó el sistema conforme a los ejemplos señalados. Un mes después —cuando se quiso enviar correspondencia por correo— se detectó que debió capturarse la calle y número en un renglón, y en otro renglón la colonia. Fue necesario reprogramar y corregir 5 000 registros. ¿En qué etapa o etapas del desarrollo se generó el error?

- El siguiente ejercicio tiene el objetivo de percibir que en ocasiones un cambio "mínimo" desde el punto de vista del usuario, implica un esfuerzo considerable para el área de sistemas.

  Un usuario acordó con usted diseñar un programa que realizará la siguiente pantalla:

```
Este programa obtiene la división de dos números sin utilizar el operador
de división.

Teclee el primer número: 13
Teclee el segundo número: 5
1) 13 - 5 = 8
2) 8 - 5 = 3
El resultado es: 2

Oprima cualquier tecla para continuar...
```

  El programa, ya elaborado, tiene el siguiente código:

```
#include <conio.h>
#include <string.h>
main() {
   int cantidad, divisor, sobrante, resultado;
   printf("Este programa obtiene la división de dos números ");
   printf("sin utilizar el operador de división\n\n");
   printf("Teclee el primer número: ");
   scanf("%d", &cantidad);
   printf("Teclee el segundo número: ");
   scanf("%d", &divisor);
   if (divisor == 0)
      printf("No se puede realizar una división entre cero.\n\n");
   else {
      /* se realiza a través de restas sucesivas */
      sobrante = cantidad;
      resultado = 0;
      while (sobrante >= divisor) {
         resultado++;
         sobrante -= divisor;
         printf("%d ) %d - %d = %d\n",
         resultado, sobrante + divisor, divisor, sobrante);
      }
      printf("El resultado es: %d\n", resultado);
   }
   printf("Gracias.");
   printf("\n\nOprima cualquier tecla para continuar...");
   getch();
}
```

Realice los cambios necesarios para obtener la pantalla siguiente e indique qué porcentaje de líneas tuvo que cambiar.

```
Este programa obtiene la división de dos números sin utilizar el
operador de división

Teclee el primer número: 13
Teclee el segundo número: 5
El resultado es: 2
Demostración del resultado
1) 13 - 5 = 8
2) 8 - 5 = 3

Oprima cualquier tecla para continuar...
```

7.8 Enfoques incremental, por versiones y de prototipos

Como ya se mencionó, el método de cascada es útil si se conoce la tecnología que se aplicará y los requisitos del sistema desde el inicio. En caso contrario, debe recurrirse a otro tipo de modelos. Los más conocidos son: incremental, por versiones y de construcción de prototipos. Cada uno responde a necesidades diferentes de los sistemas de información.

REFLEXIÓN 7.12

¿Qué enfoque aplicar a los cursos de programación?

La programación orientada a objetos —que por lo normal se ve en el segundo curso de programación— enlaza de manera natural con los enfoques incremental y de construcción de prototipos, cristalizados sobre todo bajo las metodologías de Scrum, programación extrema (XP) y proceso unificado.

Es recomendable que el estudiante trate de visualizar en primera instancia un sistema completo y amplio, para después limitarlo de manera explícita en cuanto a la funcionalidad a alcanzar durante el semestre.

Desarrollo incremental

El sistema parte de un diseño general y en forma iterativa crea y pone en producción distintos módulos del software.

Un ejemplo típico de este tipo de desarrollo se encuentra en la evolución del lenguaje Java. A diario se agregan diferentes clases que mantienen compatibilidad con las clases anteriores. Al mismo tiempo, se actualiza un catálogo de aquellas clases que desaparecerán a mediano plazo para que los nuevos desarrollos no las utilicen.

La gran ventaja de este modelo de vida del software es que el usuario no tiene que esperar la terminación del proyecto para utilizar el sistema. Partes de este se encuentran en producción a corto plazo. La desventaja es que en ocasiones se pierde eficiencia y claridad conforme avanza el proyecto para que los módulos anteriores sigan trabajando. Es cada vez más difícil obtener resultados adecuados si el diseño arquitectónico del software no es adecuado.

REFLEXIÓN 7.13

El desarrollo exitoso de una nómina

La versión 2.0 de nómina se desarrolló de manera incremental. A partir de un análisis general, se trabajó a través de mini-proyectos que finalizaban en una prueba completa del módulo. El orden fue el siguiente (se presenta solo la tercera parte de los módulos reales para simplificar el ejemplo):

- cálculo de nómina
- cálculo relacionado a la seguridad social
- interfase contable
- interfases para la carga de horas extras
- informes fiscales anuales
- cálculo de ajuste anual de impuestos
- interfase a producción

El sistema comenzó a estar en producción cuando se concluyó el segundo módulo (la parte crítica del sistema). Los demás se desarrollaron y liberaron conforme a un plan de trabajo acordado con anterioridad.

Desarrollo por versiones

En el desarrollo por versiones todas las posibles mejoras y adecuaciones tecnológicas "esperan" hasta el lanzamiento de la siguiente versión, con la única excepción de la corrección de errores.

El código fuente se modifica en su mayor parte, para alcanzar en ocasiones casi 100% del código. Estas adecuaciones se hacen porque las nuevas funcionalidades o las situaciones tecnológicas (portabilidad, lenguajes de programación, etc.) implican un cambio tan fuerte, que es más fácil reprogramar todo el sistema que seguir trabajando sobre el sistema actual.

Es usual que el número de versión esté conformado por dos números separados por un punto (versión n.m). El primer dígito de la versión corresponde a un cambio "mayor" y la segunda a un cambio "menor". En realidad, en ocasiones la numeración y el nombre responden más a criterios comerciales, como en el caso de Windows Vista. En otros casos, las versiones realmente "esconden" un desarrollo incremental como en el caso de Java.

El desarrollo por versiones es típico de sistemas operativos y software comercial.

REFLEXIÓN 7.14

La versión 2.0 de una nómina

La versión 1.0 de un producto de nómina trabajaba para una sola empresa y un único periodo (semanal o quincenal). Se decidió hacer una segunda versión cuando se vendió el software a otra organización con el compromiso de adecuarlo a sus necesidades.

La segunda versión cambió por completo el proceso de cálculo, a través de un formulario programable por el usuario: además se volvió multiempresa y multinómina. El código varió en 90% aunque se hizo con la misma tecnología.

Con las adecuaciones, la segunda versión se vendió a cuatro empresas más. En cada una de ellas se configuró de manera distinta al aprovechar la flexibilidad del diseño y solo se hicieron adecuaciones muy específicas en las interfases que cada una manejaba.

Construcción de prototipos

Los prototipos se construyen básicamente cuando el grupo de desarrollo no está seguro de la viabilidad técnica del sistema porque se incorpora una nueva tecnología o porque se trata de un problema complejo desde el punto de vista técnico. Otra causa es que el usuario pueda "imaginarse" una solución que rompe

sus paradigmas de trabajo. Su finalidad es demostrar que la parte central del desarrollo —de la cual depende que el sistema trabaje bien en su conjunto— funcionará de manera adecuada (por ejemplo: en el proceso de cálculo de nómina vía un formulario configurable).

El esfuerzo que implica es una porción relativamente pequeña del esfuerzo total de desarrollo (de 5 a 10% del total). Solo una parte del código se puede reaprovechar.

El resultado del prototipo puede ser:

- Seguir con el sistema como originalmente está planeado (el prototipo demostró su viabilidad).
- Seguir con el sistema, pero cambiar la estrategia de solución, lo cual obliga a reestructurar el plan de trabajo (no es viable el enfoque propuesto, pero existen alternativas de solución).
- Cancelar el sistema (no es viable el enfoque propuesto y no hay otra solución factible en las condiciones actuales).

REFLEXIÓN 7.15

El prototipo para un sistema de evaluaciones

Una secretaría está desarrollando un sistema para aplicar 720 exámenes en forma simultánea. Desde el punto de vista de desarrolladores y usuarios, la aplicación del examen es el punto crítico del sistema por dos motivos: a) si este módulo fallaba, no tenía sentido realizar los demás; b) era el más complejo desde el punto de vista técnico.

Los desarrolladores decidieron que el primer paso era hacer un prototipo con Java y otro con PHP. Si el primero no funcionaba, aplicarían el segundo. En la primera prueba ambos fallaron por problemas de configuración del código y del servidor; en la siguiente solo se probó el de Java, dando un error inesperado después de 18 minutos (el sistema se bloqueaba en forma repentina).

Un análisis exhaustivo realizado por un equipo interdisciplinario logró detectar la causa: un proceso que el usuario ejecutaba hacia el final de la prueba no cerraba las conexiones a la base de datos en forma adecuada.

El tercer ensayo pasó todas las pruebas de estrés (ya no fue necesario ejecutar la prueba con PHP). Ese éxito parcial liberó de presión anímica al equipo de desarrollo y demostró a los directivos que el proyecto era viable.

Un primer acercamiento a la medición de tiempos de desarrollo en sistemas incrementales

¿Cómo medir los avances del proyecto de manera sencilla y relativamente confiable cuando se trabaja con cualquier metodología de tipo incremental?

Una primera regla de oro: no conviene realizar una estimación de tiempos mientras no se tenga un diagnóstico general del sistema, el cual incluye un análisis general de las peticiones del usuario, la razón y delimitación de las mismas, si no se duplica con otros sistemas existentes y si existen mejores alternativas de solución.

A continuación, brindaremos una sugerencia sencilla y práctica, basada en nuestra propia experiencia, para tener un primer acercamiento a los tiempos de desarrollo de un sistema pequeño o mediano. Creemos que puede ser un buen inicio en este tema para los estudiantes. Esta forma de estimación no se contrapone a que en su momento se apliquen técnicas formales y con mayor riqueza conceptual.

Con base en el diagnóstico general se divide el sistema en módulos. Para cada uno de ellos se realiza un desglose detallado de actividades a fin de tener una primera estimación de tiempos. Después de esta estimación, se le presentan diversas alternativas de solución al usuario y él tendrá que elegir alguna de ellas (u otra en que el área de sistemas y el usuario estén de acuerdo).

El plan de trabajo detallado con compromiso de fechas solo se realiza para el primer módulo. Para los otros módulos solo se especifican fechas genéricas.

Los avances siempre se miden de dos formas:

1. El porcentaje de avance con respecto al módulo.
2. El porcentaje de avance con respecto al sistema en general.

Después del análisis detallado del primer módulo se recalculan los tiempos de las actividades de este módulo (y se ajustan los tiempos de todas las demás actividades conforme a las nuevas consideraciones).

Al final del módulo, se actualizan los requerimientos y los tiempos, se revisa el orden de las prioridades y se confirman o dan de baja módulos enteros. Por último, se elige el siguiente módulo a desarrollar.

Ejemplificaremos con un sistema de control escolar ficticio. Suponga que después del diagnóstico general se divide el sistema en tres iteraciones y se realiza una estimación general de tiempos para cada una de ellas. Al final se tiene la siguiente estimación de tiempos:

Desglose general de actividades del sistema de control escolar (en horas de desarrollo)

Diagnóstico general	80 cubierta
Inscripciones (iteración 1)	800
Asignación de salones (iteración 2)	600
Información académica (iteración 3)	400
Total:	**1 880**

Después de un análisis detallado del módulo de inscripciones (iteración 1) se revisa el desglose de las actividades involucradas y se recalcula el tiempo para cada actividad. El desglose quedaría similar al siguiente:

Desglose detallado de actividades del módulo de inscripciones (en horas de desarrollo)

Análisis detallado	60 cubierta
Diseño de prototipo	120
Prueba de estrés al prototipo	120
Creación de rutinas de seguridad	80
Creación de la presentación del sistema	80
Carga de grupos y horarios	100
(Otras actividades; deben desglosarse)	400
Total:	**960**

Como podrá observarse, existe una diferencia de 20% (960 a 800) con respecto a la estimación original. Eso nos obligará a revisar las estimaciones para los otros módulos. Como primera instancia, habría que aumentar dichas estimaciones en 20%.

Nuevo desglose general de actividades del sistema de control escolar

Diagnóstico general	80 cubierta
Inscripciones (iteración 1)	960 en desarrollo
Asignación de salones (iteración 2)	720
Información académica (iteración 3)	480
Total:	**2 240**

Ahora suponga que se realizan las siguientes actividades del módulo (diseño del prototipo y prueba de estrés del prototipo). ¿Cuál sería en este momento el avance del proyecto?

Avance del módulo = Actividades realizadas / Tiempo total del módulo
= 300 / 960, es decir, 31%.
Avance global del proyecto = Actividades realizadas / Tiempo total del proyecto
= 380 / 2 240, es decir, 17%.

Los siguientes cuadros son una síntesis en una hoja de cálculo de los avances del proyecto. La hoja de control de porcentaje de avances sería muy similar aunque más detallada. El concepto debe mantenerse aunque se usen herramientas automatizadas para control de proyectos.

EJERCICIO SUGERIDO

La planeación del desarrollo Caleidosystem estaba planteada como sigue:

Diagnóstico general	320 horas
Módulo 1: alta básica de usuarios y de cursos	180 horas
Módulo 2: módulo de búsqueda de información	600 horas
Módulo 3: evaluaciones automatizadas	600 horas
Módulo 4: administración del curso	520 horas
Total:	2 200 horas

El módulo 1 se concluyó en un tiempo de 190 horas. ¿Cuál es la inversión en horas esperada del proyecto hasta ese momento y cuál es el porcentaje de avance? El módulo 2 se llevó 550 horas en su desarrollo. ¿Cuál es la nueva estimación de necesidades de inversión en horas y cuál es el nuevo porcentaje de avance?

REFLEXIÓN 7.16

¿Cómo se estima el tiempo de desarrollo en los cursos?

Si se realizara una encuesta sobre la estimación de los tiempos de desarrollo en los sistemas escolares se llegaría a una conclusión preocupante: casi nadie lo usa, con lo cual se establecen metas y tiempos muchas veces irrealizables. Por ello, muchas veces el estudiante modifica sobre la marcha los alcances del sistema o sacrifica la calidad. Tal vez eso signifique bajar la calificación o cambiar la documentación del análisis. La pregunta clave es: ¿qué costumbres creará el alumno para cuando se incorpore al mercado laboral?

7.9 Gestión de proyectos con Scrum[10]

Scrum es una forma de gestionar proyectos de software fácil de explicar, una metodología de gestión del trabajo. Puede aplicarse a distintos modelos de calidad (como podría ser CMMI) puesto que estos te dicen qué tienes que hacer; es decir, te dicen que tienes que gestionar el proyecto, pero no te dicen cómo. Ahí es donde entra SCRUM como modelo de gestión del proyecto.

Los elementos básicos de SCRUM son:

- Una lista llamada *Product Backlog* con las funcionalidades de la aplicación.
- De la lista anterior, se toman las primeras funcionalidades, se descomponen en tareas y son anotadas en una lista que se llama **"Sprint Backlog"**. Estas tareas serán realizadas en el siguiente mes (aunque podría tomarse como base otro periodo menor).

[10] Esta sección se sintetizó de: Gracia, Joaquín. "Gestión de proyectos con Scrum", http://www.ingenierosoftware.com/equipos/scrum.php [consultado el 15 de enero de 2014]

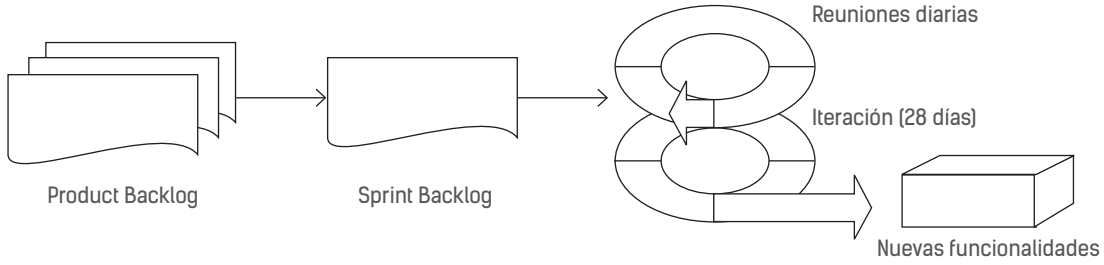


Figura 7.5 El proceso básico de Scrum.

Además de estos elementos tenemos unas cuantas reglas básicas y sencillas que tenemos que cumplir.

- Las tareas no pueden cambiar una vez que pasan del **Product Backlog** al **Sprint Backlog**, estas no se pueden cambiar para que el trabajo de un mes quede fijado.
- Al final del mes, a este periodo se le llama **Sprint**, se debe tener un ejecutable con las funcionalidades del **Sprint Backlog**.
- Todo el mundo puede añadir funcionalidades al **Product Backlog**, pero solo una persona puede decir al final si pasan al Sprint Backlog. A esta persona se le denomina **Product Owner** y es el responsable del producto final: define funcionalidades, establece fechas de entrega, cuida la utilidad del proyecto y acepta o rechaza los resultados.
- Cada día se hace una reunión de menos de 15 minutos con todo el equipo: ingenieros y gestor (llamado **Scrum Master**) en la que cada miembro del equipo expone solo los siguientes temas: ¿Qué es lo que se hizo el día anterior? ¿Qué es lo que se va a hacer hoy? ¿Qué impedimentos tengo para realizar mi trabajo? Si se tiene que tratar otro tema se hace otra reunión solo con las personas implicadas.
  El Scrum Master hace las veces de líder, gestor y facilitador que elimina barreras y aumenta la productividad, es un "puente" entre el cliente y el equipo de desarrollo.
- Al final del mes, es decir, al final del **Sprint**, se presenta el producto y se toman del Product Backlog ordenando las funcionalidades que se cubrirán en el siguiente mes.
  A estas reuniones, coordinadas por el **Scrum Master**, asiste el **Product Owner** y otros usuarios del sistema. En ellas:
  - Se liberan los incrementos del producto, ya en funcionamiento.
  - Se fomenta la retroalimentación y la lluvia de ideas.
  - Se informan las funciones, diseño, fortalezas, debilidades, esfuerzo del equipo y puntos futuros de problemas.
  - Se reportan las sorpresas.
  - No se acuerda ningún tipo de compromiso. Esto se hace en la junta de planeación Pre-Sprint.
  - Cualquier cosa puede ser modificada. El proyecto puede ser cancelado.

La duración del **Sprint** se puede modificar a lo largo del proyecto. Hay gente que prefiere Sprints más largos al comienzo, mes y medio o dos meses, ya que al principio cuesta más obtener un ejecutable, y al final Sprints más cortos, una semana o dos, cuando se está en la fase final de refinamiento. Pero básicamente el proceso es el mismo de principio a fin.

Para gestionar el proyecto solo se necesitan dos listas, el **"Product Backlog"** y el **"Sprint Backlog"**. Podemos usar Excel para manejarlas, aunque también existen herramientas automatizadas que controlan el proceso de Scrum.

REFLEXIÓN 7.17

Elaboración de exámenes psicométricos aplicando Scrum

Una empresa de exámenes psicométricos comenzó su labor automatizando el cálculo de una prueba. Debido a la demanda de mercado y a que el diseño fue muy fácil de utilizar, prosiguió incorporando varias pruebas hasta alcanzar doce en total en un lapso de dos años. El orden era establecido por el usuario. Cada una de ellas se incorporaba a producción tan pronto pasaba la etapa de pruebas.

7.10 Desarrollo empleando programación extrema (XP)

La programación extrema (XP-eXtreme Programming) es una metodología ágil para el desarrollo de proyectos de software que realiza un plan de proyecto basado en versiones del producto acordadas a partir de funcionalidades concretas. Una vez entregada la versión del proyecto y al cumplir con los requisitos (no un producto, sino una versión funcionando), el proceso vuelve a iniciarse con un conjunto mayor de funcionalidades.

La programación extrema puede dividirse en cuatro principios sobre los que se itera hasta que el proyecto finaliza (el cliente aprueba el proyecto). Estas fases o principios son: planificación, diseño, desarrollo y pruebas. A primera vista parece que no se añade nada nuevo a las metodologías tradicionales —de hecho, es prácticamente el método de cascada—, pero existen recomendaciones significativas en los detalles de cada fase y en los objetos que se marcan en cada una de ellas (iteración tras iteración) donde están las mayores diferencias (véase figura 7.6).

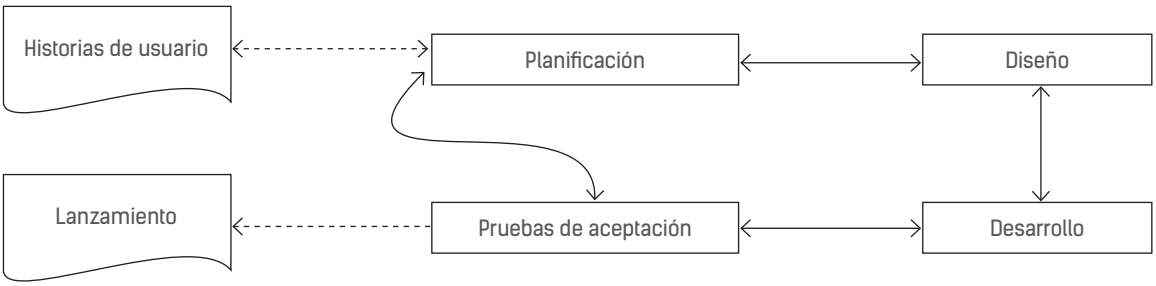


Figura 7.6 El modelo básico de la programación extrema.

Planificación

La **planificación** empieza con la confección de "Historias de usuario". De manera similar a los casos de uso, se trata de breves frases escritas por el cliente (no más de tres líneas), en las que se describe una prestación o un proceso sin ningún tipo de tecnicismo (es el usuario o el cliente quien las escribe).

Estas historias de usuario servirán para realizar la planificación de lanzamientos, así como para las pruebas de aceptación con el cliente. Para cada historia deberíamos poder estimar su tiempo ideal de desarrollo, que debería ser de 1, 2 o 3 semanas como máximo. Si el tiempo de desarrollo es mayor, se deberá partir la historia en trozos que no excedan de esas estimaciones.

A continuación podemos pasar a la propia planificación del próximo (o primer) lanzamiento del proyecto. En la reunión de planificación se debe implicar al cliente, al gestor del proyecto y a los desarrolladores. El objetivo será planificar los siguientes lanzamientos y poner en orden las historias de usuario que faltan por desarrollar. Deberá ser el cliente quien dicte el orden de las historias de usuario, y los desarrolla-

dores quienes estimen el tiempo que les llevaría desarrollarlo de manera ideal, sin tener en cuenta dependencias, ni otros trabajos o proyectos, pero sí las pruebas.[11]

Las historias de usuario se suelen reflejar en tarjetas o trozos de papel, que se agrupan y clasifican encima de la mesa durante la planificación. El resultado deberá ser un conjunto de historias que tengan sentido y que puedan llevarse a cabo en poco tiempo.

Aquí introducimos un nuevo concepto, la velocidad del proyecto. Esta velocidad nos servirá para decidir cuántas historias de usuario vamos a incluir en el próximo lanzamiento (si estamos planificando con base en el tiempo, ya fijado), o bien, cuánto tardaremos en publicar el lanzamiento (si estamos planificando por alcance, ya fijado). La velocidad del proyecto es simplemente el número de historias de usuario completadas en la iteración anterior. Si se trata de la primera iteración, habrá que hacer una estimación inicial.

La velocidad puede aumentarse si en el transcurso del desarrollo se finalizan las historias de usuario previstas antes de tiempo.[12] Entonces los desarrolladores pedirán historias de usuario que estaban planeadas para el siguiente lanzamiento.[13] Este mecanismo permite a los desarrolladores recuperarse en los periodos duros, para después acelerar el desarrollo si es posible. Recordemos que todas las historias deben tener una duración máxima, y que cuanto más parecido sea el volumen de trabajo estimado en cada una de ellas, más fiable será la medida de velocidad del desarrollo.

Este método de planificación rápido y adaptativo nos permite hacer un par de iteraciones para tener una medida fiable de lo que será la velocidad media del proyecto y estimar con más detalle el plan de lanzamientos (además de haber empezado ya con el desarrollo del mismo) en el intervalo de tiempo en que otras metodologías tardarían en documentar, planificar y realizar una estimación completa, que quizá no sería tan fiable (véase sección 7.8 en lo referente a la estimación de tiempos en desarrollos incrementales).

En la reunión de planificación, al inicio de cada iteración, también habrá que incorporar las tareas que hayan generado las pruebas de aceptación que el cliente no haya aprobado. Estas tareas se sumarán también a la velocidad del proyecto, y el cliente, que es quien escoge las historias de usuario que hay que desarrollar, no podrá elegir un número mayor que el de la velocidad del proyecto.

Los desarrolladores convertirán las historias de usuario en tareas (esas sí en lenguaje técnico) de una duración máxima ideal de tres días de programación cada una. Se escribirán en tarjetas y se pondrán encima de la mesa para agruparlas, eliminar las repetidas y asignarlas a cada programador. Es de suma importancia evitar añadir más funcionalidades que las que la historia de usuario requiera estrictamente; esta tendencia de los gestores de proyectos o analistas acostumbrados a las metodologías tradicionales debe evitarse en modelos iterativos como este, ya que desvirtúan las estimaciones y el principio de lanzamientos frecuentes.[14]

Con las tareas resultantes se volverá a comprobar que no se excede la velocidad del proyecto, al eliminar o añadir historias de usuario hasta llegar al valor esperado. Es posible que historias de usuario diferentes tengan tareas en común y esta fase de la planificación permitirá fijarlas, para así poder aumentar la velocidad del proyecto si se incluyen más historias de usuario en esta iteración.

---

[11] En términos de gestión, la programación extrema tiene grandes similitudes con Scrum, pero ahonda en varios aspectos de la forma técnica de llevar a cabo la programación.

[12] Existen premisas que no quedan explícitas en la metodología: los desarrolladores trabajarán con un sentido ético (no tardarán más por contar con mayor tiempo disponible para la entrega) y el ambiente laboral lo propiciará (respetará las estimaciones y un margen razonable de incertidumbre en la estimación de tiempos de desarrollo). Sin este contexto será difícil aplicar esta técnica de gestión.

[13] En este punto hay una similitud conceptual con el sistema kanban de producción, cuya descripción queda fuera de los límites del presente libro.

[14] En términos generales, esta estrategia es muy útil en entornos donde el usuario "sabe lo que quiere" y existe un dominio pleno de los aspectos técnicos. De no ser así, se debiera recurrir a prototipos y la metodología adecuada sería de proceso unificado.

En el momento de planificar el equipo de trabajo encargado de cada tarea o historia, es importante la movilidad de las personas, intentar que cada dos o tres iteraciones todo el mundo haya trabajado en todas las partes del sistema.[15]

Diseño

Durante el diseño de la solución, y de cada historia de usuario que lo requiera, debe buscarse la máxima simplicidad posible. Un diseño complejo siempre tarda más en desarrollarse que uno simple y casi siempre es más fácil añadir complejidad a un diseño simple que quitarla de uno complejo, por ello debe evitarse añadir funcionalidad no contemplada en esa iteración.

Asimismo, es importante y se ha probado muy útil encontrar una "metáfora" (concepto clave) para definir el sistema. Es decir, un proceso o sistema que todos conozcan (el cliente, el gestor del proyecto, los desarrolladores) y que puedan identificar con el proyecto que se está desarrollando. Encontrar una buena metáfora ayudará a tener:

- Visión común: todo el mundo estará de acuerdo en reconocer dónde está el núcleo del problema y en cómo funciona la solución.
- Vocabulario compartido: la metáfora nos ayudará a sugerir un vocabulario común para los objetos y procesos del sistema.
- Generalización: la metáfora puede sugerir nuevas ideas o soluciones. Por ejemplo, en la metáfora "el servicio de atención al cliente es una cadena de montaje", nos sugiere que el problema va pasando de grupo en grupo de gente, pero también nos hace pensar en qué pasa cuando el problema llega al final de la cadena.
- Arquitectura: la metáfora dará forma al sistema, nos identificará los objetos clave y nos sugerirá características de sus interfases.

Nombre de clase:	Superclase:	Subclases:
Responsabilidad principal:		
Responsabilidades:	Colaboradores:	

Cuadro 7.5 Las tarjetas CRC de XP, un puente entre las historias de usuario y diseño.

Para el diseño del sistema, se sugiere llevar a cabo reuniones entre todos los desarrolladores implicados, donde se toman las decisiones mediante el uso de tarjetas CRC (Clase, Responsabilidad y Colaboración). Cada tarjeta representa un objeto o clase del sistema, en la que su nombre aparece escrito en la parte superior, sus responsabilidad es en la parte izquierda y los objetos con los que colabora en la parte derecha.

[15] La idea suena muy atractiva, pues se disminuye la dependencia de algún programador en específico. Sin embargo, no es sencilla de aplicar si en algunas partes del sistema se requieren conocimientos altamente especializados de programación.

El proceso de diseño se realiza creando las tarjetas (al inicio solo se escribirá el nombre en ellas, el resto se irá completando) y situándolas próximas a las que comparten interfases o llamadas. Las tarjetas correspondientes a objetos que heredan o son interfases de otros pueden situarse encima o debajo.

Este mecanismo ayuda a que todos participen y aporte sus ideas en el diseño al mover las tarjetas encima de la mesa según este va progresando. Cuando tengamos el diseño definitivo, también será más fácil de mantener en la memoria por parte de los participantes y es entonces cuando entraremos en detalle en cada objeto y haremos los diagramas necesarios.

Por último, es clave también el concepto de refactorización, es decir, el hecho de realizar cambios necesarios en las partes de código que lo requieran sin modificar su funcionalidad o su interacción con el resto del sistema.

Codificación

Una primera diferencia importante entre esta metodología y las llamadas "tradicionales" es la disponibilidad del cliente, que debe ser total. En lugar de limitarse a escribir durante semanas una hoja de requisitos, el cliente debe participar en las reuniones de planificación, tomar decisiones, y estar disponible para los desarrolladores durante las pruebas funcionales y de aceptación.

Debido a la movilidad de los desarrolladores durante el proyecto, que participan en cada iteración en partes distintas del mismo, es de suma importancia acordar unos estándares de codificación y respetarlos en el desarrollo. Deberemos ser tan concretos como sea posible, no dejando al libre albedrío temas como la indentación en el código, la sintaxis y nombres de variables, etcétera.

En lo que respecta a la propia codificación en sí, la programación extrema nos propone que antepongamos la creación de las pruebas unitarias al propio desarrollo de las funcionalidades. Las pruebas unitarias son pequeños trozos de código que prueban las funcionalidades de un objeto, de modo que esa prueba pueda incorporarse a un proceso de pruebas automatizado. La mayoría de los lenguajes tienen hoy en día librerías para crear y ejecutar pruebas unitarias.

La idea que se fragua detrás de esta propuesta es que creando primero las pruebas que deberá pasar nuestro código tendremos una idea más clara de lo que debemos codificar y nos guiaremos por ello, implementando solo lo que nos permita pasar la prueba.[16]

Otra característica diferencial de la programación extrema es el *pair programming* o la programación por parejas. Se ha demostrado que dos programadores, trabajando en conjunto, lo hacen al mismo ritmo que cada uno por su lado, pero el resultado obtenido es de mucha más calidad. Estar los dos sentados frente al mismo monitor y pasándose el teclado cada cierto tiempo, provoca que mientras uno está concentrado en el método que está codificando, el otro piense en cómo afectará este al resto de objetos; las dudas y propuestas que surjan reducen de manera considerable el número de errores y los problemas de integración posteriores.

En una metodología incremental como esta, la integración con lo que ya se ha desarrollado en iteraciones anteriores es clave. No hay una solución única a ese problema. Según el proyecto y el equipo de trabajo, se pueden proponer soluciones, como por ejemplo designar "dueños" para cada objeto, que conozcan todas las pruebas unitarias y de integración que debe pasar, y se ocupe de integrarlo en cada lanzamiento. Otra alternativa es que cada pareja de programadores se haga responsable de hacerlo cuando todas las pruebas de la tarea que estaban realizando pasan 100%. De este modo, añadimos al paradigma *release-often* (distribuye o implanta a menudo) el *integrate-often* (integra a menudo), al reducir en gran medida las posibilidades de problemas de integración.

[16] Recomendamos tomar este idea —conocida como desarrollo guiado por pruebas— para la enseñanza y la programación a nivel profesional, independientemente de que se trabaje o no con programación extrema.

Pruebas

La construcción de las pruebas unitarias nos ahorrará mucho más tiempo del que invertimos programándolas, buscando pequeñas fallas y protegiéndonos contra ellas en forma permanente durante el resto del desarrollo. Las pruebas unitarias no deben "brincarse" o dejarse al final de todo el desarrollo. Cuanto más complicada sea la prueba, más necesaria es para asegurar que después el desarrollo hace lo que se requiere.

Las pruebas unitarias ayudarán también a la refactorización, ya que asegurarán que los cambios que hayamos introducido en la iteración actual no afectan a la funcionalidad. Cuando encontremos un fallo o *bug* en las pruebas de aceptación con el cliente, o durante el uso, deberemos crear una prueba unitaria que lo compruebe. Así aseguraremos que no vuelve a surgir en las iteraciones siguientes.

Las pruebas de aceptación se crearán a partir de las historias de usuario. Una historia puede tener una o varias pruebas, según sea la funcionalidad que hay que probar. El cliente es responsable de definir las pruebas de aceptación, que deberían ser automatizables en la medida de lo posible. Estas pruebas son del tipo "caja negra", en el sentido de que solo definen el resultado que debe tener el sistema ante unas entradas concretas. Las pruebas de aceptación que no tengan éxito generarán nuevas tareas para la próxima iteración, lo que afecta la velocidad del proyecto, y proporciona además una puntuación del éxito o fracaso de cada historia de usuario o de cada equipo de trabajo.

REFLEXIÓN 7.18

Programación extrema y la enseñanza de la programación

Existen varios cuestionamientos hacia la programación extrema. Al menos habría que destacar la posible ambigüedad en los requerimientos y que de manera implícita parte del hecho de un ambiente honesto y con tecnología conocida. Pero, independientemente de estas críticas —a nuestro juicio válidas—, la programación extrema incorpora aportaciones valiosas en todas las etapas: el involucramiento activo del cliente, la retroalimentación en los tiempos de desarrollo, el desarrollo guiado por pruebas, la programación por pares y la refactorización del código, que pueden ser llevadas a otras metodologías como cascada, Scrum o proceso unificado. También pueden y deben ser incorporadas a la enseñanza. De hecho, nosotros lo hemos hecho a lo largo del libro.

7.11 Proceso unificado

En este apartado se explica con detalle el proceso unificado, para ello se inicia con las características generales de este tipo de proceso.

Características generales del proceso unificado

En la actualidad, la metodología de proceso unificado (PU) es la mejor opción para el desarrollo de sistemas cuando no existe el dominio tecnológico de las herramientas o lenguajes de programación a aplicar, o cuando no se tiene claridad en la conceptualización a aplicar desde el punto de vista del usuario. El proceso unificado establece un marco conceptual coherente, flexible y dirigido para la creación de software de mediana y gran envergadura. Entre otras ventajas, reduce en gran medida la producción de BANOS.

El Lenguaje Unificado de Modelado (UML) es la base para toda su diagramación. En la actualidad existen herramientas para diagramación de UML muy amigables y algunas de ellas gratuitas.

No obstante sus fortalezas, habría que destacar un riesgo en la aplicación de esta metodología: no constituye un conjunto de pasos concretos; es un marco general, lo cual implica que debe adaptarse a las condiciones particulares de cada organización y proyecto. Además, se deben elegir solo los diagramas cuya realización justifique el tiempo invertido en su realización; es decir, que el costo-beneficio sea adecuado. En muchísimas ocasiones, es suficiente con media docena de diagramas para un proyecto.

La otra desventaja radica en que muchos dicen entender el enfoque y terminan aplicando un método de cascada que traiciona el enfoque básico del proceso unificado. El hecho de hacer diagramas aislados con UML no implica que se lleve bien esta metodología.

En resumen, para sacar provecho del proceso unificado hay que entender bien su enfoque y concretarlo a las necesidades específicas.

Las características generales de este proceso son:

- **Dirigido por casos de uso.** "Un caso de uso especifica una secuencia de acciones, incluyendo variantes, que el sistema puede llevar a cabo, y que producen un resultado observable de valor para un actor concreto."[17] "Son 'fragmentos' de funcionalidad que el sistema ofrece para aportar un resultado de valor para sus actores."[18] Al documentar todos los casos de uso, se reflejan en forma automática todos los requerimientos funcionales del sistema desde el punto de vista de los principales actores involucrados. Si se respeta este enfoque se evita codificar funcionalidades que no responden a ninguna necesidad concreta.

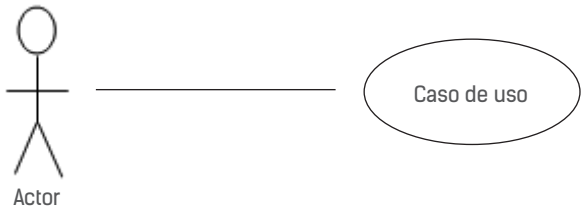


Figura 7.7

  El desarrollo bajo casos de uso permite liberaciones de módulos sin necesidad de que el sistema se encuentre terminado.

  La documentación de la etapa de requerimientos será sobre todo a través de los diagramas de casos de uso. A estos diagramas deberán agregarse otros diagramas que se juzguen pertinentes. Por ejemplo: diagramas de actividades.
- **Centrado en la arquitectura.** ¿Por cuáles casos de uso debe comenzarse el desarrollo? Por aquellos que representan la razón de ser de un sistema, o aquellos que son indispensables y consumen altos volúmenes de recursos. En otras palabras, por los casos de uso por los que, si fallan, el sistema no tiene sentido. Por ejemplo, el cálculo de nómina para un sistema de nómina; la inscripción de alumnos para un sistema de inscripciones o el cierre contable anual. En consecuencia, al inicio del proyecto debe probarse que la arquitectura propuesta del sistema puede ejecutar dichos casos de uso.

  El argumento es sencillo: si el sistema no es viable, esto debe conocerse lo antes posible a fin de que el desarrollo completo se convierta en un BANO.

  El método de cascada tradicional lleva las pruebas hasta el final. Los errores críticos se suelen detectar en esta etapa, cuando es muy difícil cualquier replanteamiento del proyecto, tanto administrativo como técnico. Cuando se detecta un error crítico en el método de cascada, casi siempre se termina por cancelar el proyecto cuando este ya consumió casi todo su presupuesto.

  El proceso unificado, por el contrario, trata en primer lugar de demostrar la viabilidad de la arquitectura. Esto provoca dos efectos: por un lado, es posible realizar ajustes; si esto no fuera viable, el proyecto se cancelaría, pero cuando ha consumido una parte relativamente baja de su presupuesto.
- **Iterativo e incremental.** La metodología del proceso unificado plantea que se trabaje sobre iteraciones controladas, cada una de las cuales pasa por todos los flujos de trabajo (desde los requisitos

[17] Véase Booch, Grady, Rumbaugh, James y Jacobson, Ivan. *El proceso unificado de desarrollo de software*. Pearson, España, 2000, p. 39.
[18] Ibid, p. 129.

hasta las pruebas). Las primeras iteraciones deben demostrar la viabilidad del proyecto y de la arquitectura del sistema, por lo cual tienen pocas actividades de codificación; las siguientes deberán incorporar funcionalidades. Al término de cada iteración se debe replantear el análisis de riesgos, la estimación de tiempos y el plan de trabajo en general.

Al inicio del proyecto se debe plantear la cantidad de iteraciones que se considerarán en el desarrollo. Como recomendación, cada iteración debe durar un tiempo máximo de cuatro meses.

Las diferencias principales entre el método de cascada y el proceso unificado

- El proyecto global se divide en iteraciones (mini-proyectos), con lo cual se disminuye el tiempo que transcurre entre la etapa de requerimientos y la de pruebas y, con ello, la posibilidad de cambios de requerimientos (véase figura 7.8).

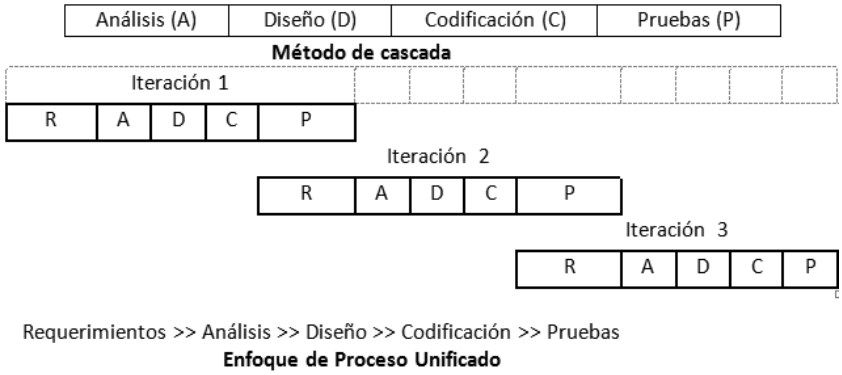


Figura 7.8 El enfoque de iteraciones en el proceso unificado en comparación con el método de cascada.

- Al comenzar por los casos de uso que definen la arquitectura se demuestra la viabilidad del sistema desde el inicio. De esta forma se evita trabajar en desarrollos completos que en la etapa final de pruebas se descubre que no son factibles. En consecuencia, se reducen los BANOS en gran medida.
- La diagramación se unifica con UML, lo que facilita el intercambio de información entre los desarrolladores.

REFLEXIÓN 7.19

Sugerencia pedagógica sobre el proceso unificado

Es necesario incluir formalmente el tema del proceso unificado en alguna asignatura del plan de estudios (por ejemplo, ingeniería de software). Sin embargo, las habilidades a largo plazo dependen más de lo que el alumno realiza de manera cotidiana que de la teoría expuesta. La única forma de que el estudiante logre asimilar el proceso unificado es que esta metodología se introduzca dentro de las diferentes asignaturas:

- El levantamiento de casos de uso debe realizarse en aquellas asignaturas que se aboquen a la fase de requerimientos con base en usuarios reales. Se aconseja llegar hasta un diseño de pantallas y reportes.
- La fase de elaboración puede verse en asignaturas en donde se aborden estructuras de datos y complejidad computacional, en cuyo caso es conveniente partir de diagramas de casos de uso ya levantados y con base en ellos construir prototipos.
- La transformación de clases a código se acomoda bien en un curso de programación de Java.

Si estos enlaces no se realizan de manera planeada, por lo regular el alumno hará sus sistemas sin aplicar ninguna metodología (aunque "recite" la teoría de manera impecable). Y eso es lo que reflejará en el campo laboral.

Las fases y flujos del proceso unificado

Las iteraciones planteadas para el desarrollo de un sistema se agrupan en cuatro fases, cada una de ellas con un propósito general específico. Cada fase puede tener una o más iteraciones y estas incluyen todos los flujos de trabajo (desde los requisitos hasta las pruebas).

Cada iteración abarca varios casos de uso que, en conjunto, cubren un área de interés completa para usuarios específicos. La idea es tender hacia liberaciones paulatinas del desarrollo. Se sugiere que cada iteración tenga entre uno y dos meses de duración.

Las fases involucradas en el proceso unificado son (véase cuadro 7.6):

- **Fase de inicio.** Está encaminada a realizar un análisis de negocio y verificar si el sistema está acorde con los objetivos de la empresa y cuáles serán los alcances del mismo. Incluye un esbozo de posibles arquitecturas.
- **Fase de elaboración.** En la cual se diseña y prueba la arquitectura del sistema.
- **Fase de construcción.** Centrada en la codificación de las funcionalidades del sistema.
- **Fase de transición.** Abarca las pruebas integrales y la puesta a punto: el camino desde la versión "beta" hasta el producto final.

Cuadro 7.6 Fases y flujos de trabajo del proceso unificado

Fases: / Flujos de trabajo:	Inicio		Elaboración		Construcción			Transición		Mantenimiento		
Requisitos	↓	↓										
Análisis												
Diseño												
Implementación												
Pruebas	↓	↓										
	Iter. #1	Iter. #2	----	----	----	----	----	----	----	----	Iter. #n–1	Iter. #n

- **Fase de mantenimiento.**[19] Incluye todas aquellas modificaciones al producto cuando ya se encuentra en producción, hasta que el sistema se elimina porque ya no responde a las expectativas de la empresa. Esta fase representa entre 65 y 80% del esfuerzo total; el resto lo representa el desarrollo del sistema (las fases de inicio, elaboración, construcción y transición).

La documentación del proceso unificado

Primera "regla de oro": debe ser parte del desarrollo y aportar algo concreto a este. De otra forma, nunca se llevará a cabo. Los casos de uso pueden ser un buen ejemplo. Los requerimientos deben ser levantados directamente en casos de uso. Pudiera ser de manera simultánea a las entrevistas con el usuario o al pasar en limpio las conclusiones de dichas entrevistas. Dejarlas para después carece de sentido.

Segunda "regla de oro": deben tenerse herramientas que hagan ágil el levantamiento de la información, que pudiera ser pagado o gratuito.

[19] La mayoría de los autores no considera la fase de mantenimiento como parte del proceso unificado, argumentando que el mantenimiento equivale a un mini-desarrollo. No obstante, conviene destacarlo desde un inicio por sus múltiples implicaciones en diversos rubros del proyecto, entre ellos el costo-beneficio y el manejo de la documentación.

Tercera "regla de oro": solo deben hacerse aquellas actividades de documentación que se justifiquen por costo-beneficio. Si una documentación consume más de lo que aporta, tan solo debe dejar de hacerse. Se trata de que la documentación aporte al proyecto, no de una carga inútil de trabajo.

Cuarta "regla de oro": no toda la documentación debe mantenerse a lo largo de la vida útil del sistema. Alguna solo tiene sentido para determinadas etapas.

La fase de inicio

En la fase de inicio se desarrolla una descripción del producto final, se parte de una idea original y de la recopilación de las necesidades y expectativas de los distintos usuarios. Se pretende responder a los siguientes cuestionamientos:

a) ¿Cuáles son las principales funciones del sistema para sus usuarios más importantes? ¿Cuáles funcionalidades quedarán fuera del alcance del sistema?
b) ¿Cómo contribuirá el sistema a los objetivos de la empresa?
c) ¿Existen funcionalidades que ya realizan los sistemas actuales? ¿Cuál será la relación de este sistema con ellos?
d) ¿Cómo podría ser la arquitectura del sistema?
e) ¿Cuál es el plan del proyecto y cuánto costará desarrollar el producto? El plan del proyecto abarca hasta la etapa de pruebas. No obstante, el costo del proyecto incluye una estimación del presupuesto destinado al mantenimiento.
f) ¿Cuáles son los principales riesgos de la realización del sistema y de su mantenimiento?

Dichos cuestionamientos se resuelven a través de distintas técnicas:[20]

a) Matriz para el diagnóstico general de los requisitos.
b) Análisis costo-beneficio.
c) Evaluación de alternativas.
d) Estimación de riesgos.
e) Medición del esfuerzo de un proyecto con el contraste de las diferentes técnicas: memoria histórica; consulta experta; métodos de estimación por puntos de función y desglose de funcionalidades.

El documento con que culmina la fase de inicio debe incluir:

a) Especificación de los requerimientos funcionales.
b) Análisis costo-beneficio para los principales usuarios.
c) Alternativas de solución planteadas y alternativa elegida.
d) Riesgos potenciales para el desarrollo del sistema.
e) Estimación del esfuerzo (horas-hombre) para el desarrollo.
f) Plan preliminar de trabajo.
g) Especificación de los costos involucrados.
h) Esbozo de posibles arquitecturas del software.

Fase de elaboración

En esta fase se especifican en forma detallada la mayoría de los casos de uso del producto y se diseña y prueba la arquitectura del sistema. Al finalizar, se actualiza la estimación de los recursos necesarios para

[20] Lamentablemente, no pueden incluirse todas estas técnicas en este libro. Les recomendamos a los lectores buscarlas en bibliografía sobre administración de proyectos de desarrollo de software.

realizar el proyecto, incluyendo dentro de estos costos y tiempos. La idea principal es llegar a una definición muy aproximada de requerimientos, arquitectura y plan de trabajo a fin de tener los riesgos lo bastante controlados como para comprometer el desarrollo entero mediante un contrato.

Durante esta fase también se deben diseñar todas las reglas generales del sistema: imagen (tipos de letras, colores, menús, etc.); rutinas genéricas relativas a la codificación; recomendaciones de seguridad; estándares de programación, etc. De cualquier forma, al culminar esta fase (y, en general, al finalizar cada iteración) se debe "jugar" con las variables de tiempo, costos y alcances del desarrollo.

Si se tuviera que hacer una analogía con una casa, podríamos decir que al terminar esta fase se tienen resueltas dos situaciones: un diseño arquitectónico factible y avalado por el usuario, y el prototipo de acabados que se manejarán.

La construcción de prototipos

Los prototipos son desarrollos de software que abarcan la solución de problemas técnicos esenciales para el desarrollo del sistema, ligados principalmente a altos volúmenes de información, nuevos algoritmos o utilización de nuevas herramientas. Su aplicación es indispensable cuando se presentan situaciones técnicas nuevas que ponen en duda la viabilidad del proyecto: tecnologías nuevas, altos volúmenes de información, algoritmos nuevos, etc. A través de ella se concreta el criterio del proceso unificado, "centrado en la arquitectura".

¿Qué sucede si se descubre que la arquitectura planteada no es viable? En esencia hay cuatro caminos:

1. Retomar una de las alternativas planteadas en la fase de inicio.
2. Desarrollar otra alternativa con los nuevos elementos conocidos.
3. Delimitar los requerimientos.
4. Cancelar el proyecto.

Si se continúa el proyecto, es posible readecuar el alcance de las funcionalidades, sobre todo porque tal vez existe a esta altura una condicionante en los costos o los tiempos del sistema. En muchas ocasiones, la arquitectura en general es viable, pero se delimitan aspectos que deben tomarse en cuenta (configuraciones, mejoras en el algoritmo, optimización de rutinas, uso de herramientas especiales, etcétera).

Existen varias recomendaciones genéricas para la construcción de prototipos:

- Deben reflejar las variables relacionadas con la situación planteada, sin dejar de lado casos reales que puedan causar que el sistema falle en el mundo real (por ejemplo: combinación de tecnologías o errores de datos). Dicho de otro modo, el prototipo va encaminado a un mundo "real", no a un mundo "ideal".
- La especificación del requisito debe ser formal, sin dejar ambigüedades o sobreentendidos.
- Puede dejar sin instrumentar diversos puntos que no interfieren en la solución técnica del problema. Por ejemplo: uso de colores o tipos de letra.
- Su código puede ser aprovechado en forma parcial o incluso desechado. De hecho, en la gran mayoría de los casos la solución técnica es una aportación que justifica plenamente el desarrollo del algoritmo.
- En muchos casos, el prototipo puede desechar o acotar la solución técnica planteada originalmente. En estas situaciones, se puede hablar de que la fase de elaboración fue exitosa, pues obligó a replantear estrategias de solución antes de que se consumieran muchos recursos del proyecto.
- Cuando se juzgue pertinente, se deben documentar los aspectos de configuración que permiten que el prototipo trabaje de manera adecuada y aquellos que impidan su funcionamiento correcto.

Fase de construcción

Durante esta fase se crea el producto. La arquitectura del sistema y su imagen general (establecidos en la fase de elaboración) se toman como base para realizar el sistema completo, aunque pueden incorporarse mejoras.

Existen varios "estilos" en las personas dedicadas al desarrollo. Destacaremos dos:

Algunos programadores "natos" tienden a escribir líneas de código cuando todavía no existen los elementos necesarios: requerimientos bien definidos, una arquitectura probada y una imagen diseñada. Por lo regular, se reescriben grandes porciones de código, sobre todo al unificar rutinas hechas por diferentes programadores; la documentación es casi inexistente y en muchas ocasiones el sistema completo resulta un BANO.

Algunos programadores "académicos" tienden a realizar diagramas excesivos y posponer la construcción del sistema hasta que hay un acuerdo total sobre todos los detalles. Por lo regular, los tiempos se alargan de modo peligroso sin que el cliente obtenga un resultado concreto, a pesar de que se tiene una gran cantidad de documentación. Al igual que en los programadores "natos", en muchas ocasiones el sistema completo resulta un BANO.

Es necesario evitar ambos extremos. El trabajo de construcción debe comenzarse tan pronto se tengan las bases mínimas necesarias. Para usar una frase coloquial: "ni tanto que queme al santo, ni tanto que no lo alumbre".

Al final de esta fase, el producto contiene todos los casos de uso que la dirección y el cliente han acordado para el desarrollo de esta versión. La pregunta decisiva es: ¿cubre el producto las necesidades de algunos usuarios de manera suficiente como para hacer una primera entrega?

Esta fase es la que por lo común tiene más iteraciones y la que consume la mayor parte de los recursos del proyecto. El diseño del proyecto es uno de los rubros más importantes en esta etapa. Las iteraciones deben establecerse de manera que pueda haber liberaciones parciales del sistema. Como recomendación general, hay que buscar que no pase más de medio año sin poner en producción una parte del sistema.

REFLEXIÓN 7.20

Una inconsistencia en la conexión con la base de datos

En un sistema realizado con ASP clásico se puso como estándar que todas las páginas conectaran con la base de datos; así se utiliza un archivo de conexión llamado conexión.inc

El segundo año cambió la ubicación del servidor y se modificó la dirección contenida en conexión.inc. Dos días después, los usuarios reportaron múltiples inconsistencias en su información. El motivo: un programador había puesto la cadena de conexión directamente en su código, por lo cual varios resultados se grabaron en la base de datos anterior.

La corrección de la información fue ardua, aunque en este caso sin gran impacto. El comentario final del equipo de desarrollo fue agridulce: "Tuvimos suerte. ¿No crees?"

La coherencia entre la fase de construcción y las fases previas

Durante la fase de construcción es necesario cuidar que se respeten todos aquellos **elementos** establecidos en fases previas, sin que esto constituya una camisa de fuerza. Es necesario enfatizar en:

- El seguimiento de las especificaciones establecidas en los casos de uso.
- El respeto a los estándares de presentación manejados y el uso de archivos de configuración de imágenes (hojas de estilo o similares).
- El respeto a los estándares de codificación establecidos.

¿Qué sucedería si no se respetaran estas recomendaciones? Las consecuencias serán diversas, desde un producto sin gran impacto en la presentación hasta problemas críticos que impidan la integración del sistema.

Muchos de estos problemas se detectarán hasta el mantenimiento del sistema.

Fase de transición

Esta fase cubre el periodo durante el cual un número reducido de usuarios con experiencia prueba el producto e informa de defectos y deficiencias. Constituye el sinuoso camino entre la versión beta y la versión final. Los desarrolladores, además de corregir los problemas, incorporarán algunas de las mejoras sugeridas.

Durante esta fase se realizan las pruebas en paralelo, los manuales de usuario, la capacitación, la carga y puesta a punto del sistema, así como la formación de una línea de ayuda y asistencia.

El manual del usuario

El manual del usuario debe contener las instrucciones para el manejo del sistema, tanto para los casos normales como para los casos excepcionales. Siempre va dirigido al usuario final y por tanto debe estar en los términos que él maneja. Los ejemplos siempre deben llenarse con datos similares a los datos reales y ser consistentes con los que el sistema recibe o arroja. En la actualidad se prefiere que los manuales del usuario se instalen en forma automática y estén disponibles en todo momento para el usuario. Queda en desuso los manuales impresos en papel (por lo menos en el área de software).

REFLEXIÓN 7.21

El manual del usuario de un sistema tributario

Un sistema tributario dejaba dudas con respecto al manejo, pues su redacción era confusa e insuficiente. Cuando el usuario buscaba, esperanzado, el ejemplo, este venía con anotaciones como la que sigue (el ejemplo fue simplificado por razones de espacio):

```
A|A|A|0|0|0|0
```

¿Resultado? El usuario quedaba aún más confundido.

Después de una docena de pruebas y frases que no podemos transcribir, el usuario logró deducir lo que solicitaban:

```
Apellido paterno | apellido materno | nombre de pila |
        ingresos totales | ingresos gravables |
           ingresos exentos | impuesto pagado
```

y que la cadena esperada era similar a la siguiente:

```
LUNA|JIMENEZ|LAURA|40000|38500|1500|2858
```

El manual de instalación

El manual de instalación debe indicar todos los aspectos que tienen que ver con la instalación y configuración del sistema y su relación con el software con el cual convivirá. Entre otros aspectos:

- Sistema operativo: directorios a crear, permisos y variables de entorno. Posibles variaciones en la instalación con respecto a las distintas versiones del sistema operativo.
- Base de datos: configuración de la base de datos; versión del sistema manejador de la base de datos; scripts de instalación; rutinas contenidas en la base de datos; usuarios a crear.
- Redes: direcciones lógicas y físicas de la red bajo la cual se instalará.
- Herramientas de desarrollo: versiones de las herramientas de desarrollo; directrices de compilación.
- Discos automáticos para la instalación.
- Si existen opciones para retomar la información del sistema en sus versiones anteriores.
- Posibles variaciones de configuración según el país (manejo de fechas, monedas, caracteres, etcétera).
- Navegadores con los cuales es compatible.
- Configuración del servidor de internet: versión; configuración que debe tener.

Estos aspectos se detallarán para todo el equipo involucrado: equipo cliente, servidor de internet y servidor de base de datos.

Fase de mantenimiento

La fase de mantenimiento no está considerada en la metodología del proceso unificado. Sin embargo, conviene involucrarla desde el inicio en la conceptualización del sistema, pues alrededor de 80% de los costos de un sistema corresponde a esta fase.

El mantenimiento involucra todas aquellas modificaciones al producto a partir del momento en que las especificaciones originales ya fueron concluidas:

- Las modificaciones al sistema por cambios de requerimientos cuando las especificaciones originales ya se habían terminado y el sistema no se encuentra aún en producción. En este caso se cae en el absurdo de comenzar el mantenimiento antes de aprovechar un desarrollo finalizado. Obviamente, el esfuerzo original constituye un BANO.
- Las adecuaciones al sistema por cambios en los requerimientos, internos o externos, de la organización. Estos cambios pueden darse por aspectos legales, mejoras en los procesos de información, nuevas especificaciones tecnológicas, compra-venta de empresas, etcétera.
- Eliminación del sistema cuando este ya no responde a las expectativas de la organización.

REFLEXIÓN 7.22

El desarrollo exitoso de una nómina

La versión 2.0 de nómina se desarrolló de manera incremental. A partir de un análisis general, se trabajó a través de mini-proyectos que finalizaban en una prueba completa del módulo.

La primera iteración consistió en una revisión general de todos los procesos que debían incluirse y el planteamiento de alternativas de solución. Se eligió realizarlo en el mismo lenguaje de programación de la versión 1.0, pero cambiando el diseño por completo, lo cual implicaba una reprogramación total. Había dos alternativas de solución: una programación "tradicional" cálculo por cálculo y otra programación "innovadora", por medio de la creación de un formulario (intérprete que permitía la creación automática de fórmulas). Esta segunda alternativa reducía 40% el tiempo de desarrollo, pero era la primera vez que se intentaba realizar. Se decidió construir un prototipo reducido de la segunda opción para probar su viabilidad. Con ello se dio por concluida la fase de inicio.

A continuación, se detallaron las especificaciones del prototipo, se codificó y se probó. Las pruebas del prototipo fueron exitosas y la arquitectura funcionó de manera adecuada. Con ello se dio por concluida la fase de elaboración.

El desarrollo de los módulos (fase de construcción) se hizo en el orden siguiente (se presenta solo la tercera parte de los módulos reales para simplificar el ejemplo): cálculo de nómina, cálculo relacionado a la seguridad social, interfase contable, interfases para la carga de horas extras, Informes fiscales anuales, cálculo de ajuste anual de impuestos.

El sistema comenzó a estar en producción cuando se concluyó el segundo módulo, después de un mes de pruebas en paralelo, durante las cuales se corrigieron errores leves y se detectaron mejoras sencillas que optimizaban en forma significativa la funcionalidad. Cada mejora requería el detalle de especificaciones, codificación y pruebas. Esta puede considerarse como la etapa de transición.

Los demás módulos se desarrollaron y liberaron conforme a un plan de trabajo acordado con anterioridad. Para cada uno se hacían pruebas, se revisaban los cálculos y se hacían las correcciones necesarias. (Puede decirse que se combinaba construcción y transición, lo cual es permitido. El proceso unificado debe adaptarse a las necesidades específicas de cada proyecto.)

Meses después, se hizo un reporte especial mensual para producción (nuevos requerimientos por decisión de la empresa) y se modificaron los informes para el pago de impuestos (nuevos requerimientos por cambios legales). Estas adaptaciones se encuentran dentro de la fase de mantenimiento.

La refactorización del código

La refactorización del código se da cuando se integra una nueva funcionalidad. Debe revisarse que la nueva funcionalidad no afecte al código que ya está funcionando.

Las consideraciones generales son las siguientes:

- No debe haber ningún problema si la nueva funcionalidad no modifica ninguna de las clases ya existentes.
- No debe haber ningún problema si modifica la parte interna de una clase, pero conserva sus variables y métodos públicos.
- No debe haber ningún problema si agrega variables y métodos públicos. Una situación similar se da si se agregan campos o tablas a la base de datos.
- Si modifica las variables o los métodos públicos, hay que revisar todas aquellas partes del sistema que los utilizan para readecuarlas. Una situación similar se da si se cambian nombres de campos o tablas.
- La refactorización es un proceso indispensable, relativamente sencillo, pero laborioso. Si se omite, partes del sistema que ya trabajaban en forma adecuada pueden dejar de funcionar.

El flujo de trabajo de los requisitos

Los casos de uso

¿Cómo identificar los requerimientos funcionales del sistema para sus usuarios más importantes? La respuesta obvia parece ser: "hay que preguntarle al usuario y asunto terminado". Pero esta respuesta es incompleta.

El enfoque "tradicional" es que el usuario sabe lo que quiere y basta con que un analista transforme el punto de vista del usuario en especificaciones que los programadores puedan seguir, normalmente sin una diagramación formal.

Un enfoque "realista" supone que no siempre el usuario sabe lo que quiere, además de que con dificultad puede expresarlo de manera clara y completa. Hay al menos tres variables que vuelven complejo el levantamiento de requisitos: un sistema tiene varios tipos de usuarios diferentes, rara vez los usuarios conocen las posibilidades tecnológicas y el usuario nunca dirá lo que para él es obvio (lo cual sucede casi con todas las personas). Para evitar ambigüedades, hay que uniformar el tipo de diagramación y la forma de documentar los requisitos.

La base para el levantamiento de los requisitos son los casos de uso. "Un caso de uso especifica una secuencia de acciones, incluyendo variantes, que el sistema puede llevar a cabo, y que producen un resultado observable de valor para un actor concreto."[21] Los casos de uso tratan de captar la visión del usuario, por ello siempre se expresan en el "lenguaje del cliente", si se utilizan términos comprensibles para este.

Como primer acercamiento, el término "actor" en UML puede interpretarse como usuario en el área de sistemas y como cliente para el modelo ISO. Sin embargo, un "actor" también puede ser un sistema externo que requiera alguna información del sistema actual.

Los actores son abstracciones de usuarios o sistemas reales, "terceros fuera del sistema que colaboran con el sistema".[22] Cada uno de ellos es un usuario "típico", que interviene en distintos casos de uso. Por ejemplo: una escuela puede tener miles de alumnos inscritos, pero esos miles de alumnos estar representados por unos cuantos actores: alumnos regulares, alumnos irregulares, alumnos de idiomas, alumnos fuera de reglamento y alumnos de posgrado. La elección de los casos de uso como base de los requisitos no es casual. El levantamiento correcto de los casos de uso implica que se consideren los distintos actores involucrados y los procesos de interés para cada uno de ellos.

[21] *Ibid*, p.39.
[22] *Ibid*, p.128.

Los actores se reflejan por el dibujo de una figura humana, los casos de uso con una elipse y su vínculo a través de una línea recta.

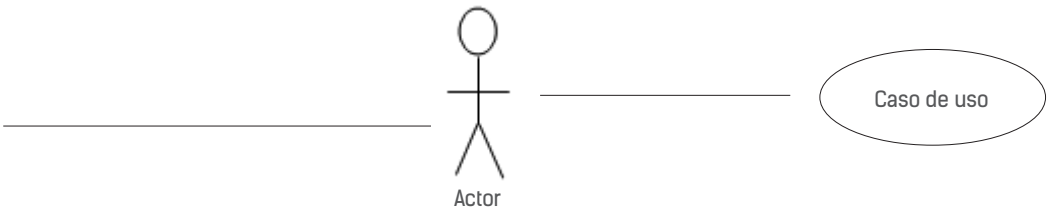


Figura 7.9

"En su forma más simple, el caso de uso se obtiene si se habla con los usuarios habituales y se analiza con ellos las distintas cosas que deseen hacer con el sistema. Se debe abordar cada cosa discreta que quieran, darle un nombre y escribir un texto descriptivo breve (no más de unos cuantos párrafos).

"Durante la elaboración, esto es todo lo que necesitará para empezar. No trate de tener todos los detalles justo desde el principio; los podrá obtener cuando los necesite. Sin embargo, si considera que un caso de uso tiene ramificaciones arquitectónicas de importancia, necesitará más detalles a la mano. La mayoría de los casos de uso se pueden detallar durante la iteración dada, a medida que se construyen."

Como primer acercamiento, los casos de uso deben reflejar los objetivos del usuario más que las interacciones con el sistema. Conforme se refinan, los objetivos se complementan con el proceso, pero después de analizar si no existen otras soluciones aparte de la primera que se le ocurre el usuario. En otras palabras, en primer lugar se pregunta: "¿por qué se hace esto?", en lugar de: "¿cómo se hace esto?".[23]

Cuando se construyen los casos de uso se debe tratar de vincular el requerimiento con el actor que en verdad hace uso de él; es decir, quien necesita la información. Por ejemplo, si nómina produce un archivo contable que después carga contabilidad, no debe admitirse este requerimiento como válido sin hablar antes con el actor "contabilidad", que es el usuario real. Por otra parte, debe tomarse en cuenta que los usuarios piensan en razón de las posibilidades tecnológicas que conocen; por ello el analista debe captar la finalidad que se persigue y sugerir al usuario otras alternativas de solución. La descripción final de los casos de uso debe describir cómo se logrará el objetivo del usuario si se aprovechan las posibilidades tecnológicas existentes y asequibles.

Cuando se comienza a trabajar con casos de uso, es frecuente poner actividades como casos de uso. Por citar un ejemplo, algunos ponen "preguntar al vendedor si hay boletos", "pagar", "recibir el cambio" como tres casos de uso separados, cuando en realidad forman parte, junto con otras actividades, del caso de uso "comprar boletos". Por lo regular los casos de uso tienen las siguientes características:

- Las actividades van encaminadas a satisfacer un objetivo único para el usuario.
- El caso de uso es "atómico". Puede interpretarse por sí mismo sin recurrir a otros casos de uso.
- Suceden en una sola unidad de tiempo (una plática, la interacción con un sistema, etc.). Un caso de uso no se prolonga por varios días.
- Se relacionan con los mismos actores.

También se suelen considerar casos de uso separados como uno solo. Un ejemplo típico es cuando se realiza el pago del servicio del automóvil y se llena una encuesta de satisfacción. En realidad, existen dos casos de uso distintos, aunque se realizan "en paralelo", pues se interactúa con objetivos diferentes y los usuarios de la información son distintos.

Además de los vínculos entre actor y casos de uso, existen dos vínculos muy frecuentes en los casos de uso:

[23] Fowler, Martin, Scott, Kendall. *UML gota a gota*. Pearson, México, 1999, p.50.

- Las relaciones **extiende (extends)**. Se emplean cuando un caso de uso hace más de lo normal de manera frecuente, pero solo en algunos es una variación de la conducta normal. Por ejemplo, solicitar autorización del gerente del banco para depósitos que sobrepasen determinado importe. Se identifican con una flecha que va del caso de uso que extiende al caso de uso normal.[24]
- Las relaciones **usa (uses)**. Se utilizan cuando se tiene una porción de comportamiento similar en más de un caso de uso y no se quiere copiar la descripción de tal conducta. Por ejemplo, solicitar el historial crediticio es un caso de uso que puede ser usado por otros: solicitar crédito automotriz, solicitar tarjeta de crédito, etcétera.

Un ejemplo de diagrama de casos de uso se muestra en la figura 7.10.

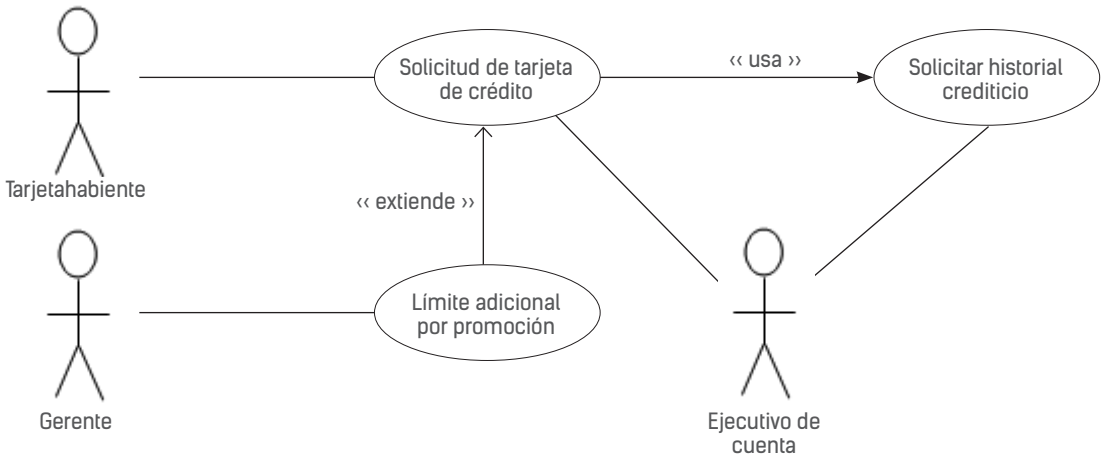


Figura 7.10 Ejemplo de caso de uso con "usa" y "extiende".

Los casos de uso se deben complementar con aquellos diagramas que se juzguen necesarios para entender el problema, por lo menos en su conceptualización general. Algunos de los complementos sugeridos:

- Los diagramas de actividades, que permiten visualizar un proceso.
- Los diagramas de clase, cuyo objetivo es identificar los conceptos más importantes y sus principales atributos. En este momento no es necesario asignar tipos de datos ni operaciones a cada clase.
- La matriz para el diagnóstico general de los requisitos. Su finalidad es delimitar los requisitos funcionales y ver la interacción de estos con otros sistemas de la empresa. A nuestro juicio, es indispensable aplicar esta matriz en el flujo de trabajo de requisitos durante la fase de inicio. En las siguientes fases, su utilidad tiende a disminuir.

Los casos de uso y la matriz de perfiles de usuario

Como resultado de los casos de uso, se obtiene de manera natural una matriz de perfiles de usuario, que conviene formalizar en un cuadro donde cada actor corresponde a un perfil de permisos diferente. En realidad, la matriz de perfiles de usuario puede deducirse a través de los casos de uso. Pero la práctica ha demostrado que cuando no se formaliza se tienden a pasar por alto los aspectos de permisos y seguridad (véase cuadro 7.7). La matriz de usuarios también puede detallar el tipo de permiso del actor: "consulta", "alta", "baja", "cambio", etcétera.

[24] Desde el punto de vista conceptual, esta flecha es muy similar a la flecha que marca la herencia de clases. Se puede interpretar como que el caso de uso "extendido" es un caso especializado que incluye al uso normal más otras actividades.

Cuadro 7.7 Matriz de funcionalidades y actores derivada de los casos de uso

Funcionalidades››	Funcionalidad 1	Funcionalidad 2	Funcionalidad 3	....	Funcionalidad *n*
Actores					
Actor 1					
Actor 2					
Actor 3					
.....					
Actor *m*					

Conviene también preparar un listado de los principales usuarios del sistema y el perfil de usuario que tendrán. A lo largo del desarrollo se deberán emplear usuarios de prueba con un perfil típico.

Es muy común que todo el desarrollo se realice con un perfil de acceso total. A unos días de la liberación se crean los usuarios y se prueban los perfiles. En general esto arroja problemas de seguridad, de configuración del sistema y de facilidad de uso, que pueden ir desde problemas menores hasta errores graves.

Los casos de uso y los requisitos no funcionales

Al levantar los requisitos funcionales se detecta que varios de ellos solo pueden cumplirse si se cubren algunos requisitos técnicos. En la mayoría de los casos, los requisitos no funcionales son específicos de un solo caso de uso. En estas situaciones, el caso de uso correspondiente puede tener "observaciones técnicas". Los requisitos no funcionales que atañen a más de un caso de uso deben mantenerse en un documento aparte de requisitos adicionales. Los requerimientos no funcionales se obtienen a partir de los requerimientos funcionales; por ello, nunca debería colocarse un requerimiento no funcional "por costumbre" o "porque es la mejor tecnología".

Todos los requisitos no funcionales representan, en conjunto, un grupo de necesidades que deben ser cubiertas por la arquitectura del sistema. Entre ellas:

a) Tiempo de respuesta del sistema.
b) Seguridad de la información.
c) Datos que se requieren en tiempo real.
d) Limitaciones presupuestales.
e) Detección de usuarios con capacidades distintas.
f) Diferentes formatos de información (texto, imágenes, audio, video, etcétera).

Además de los requerimientos funcionales, existen otras condicionantes organizativas. Entre ellas, estándares que la organización ha puesto porque le representan un mejor costo-beneficio desde el punto de vista tecnológico. Por ejemplo, los requerimientos funcionales y no funcionales de un proyecto pueden satisfacerse con sistemas manejadores de bases de datos gratuitos. No obstante, el estándar de la organización es Oracle, porque se juzga que es la mejor opción para aplicarla al conjunto de sistemas de la empresa. La elección, por política de la empresa, es Oracle (claro está, siempre y cuando satisfaga los requerimientos).

Los casos de uso y el glosario de términos

¿Para qué sirve un glosario de términos? Básicamente para aminorar la ambigüedad de lo que se expresa en los casos de uso, a fin de que usuarios y analistas le den un mismo significado a las expresiones. Los términos a incluir se derivan en forma directa de aquellos expresados en los casos de uso.

El glosario de términos debe expresarse en términos del usuario. Debe contener aquellos conceptos manejados por este que no son del conocimiento del analista, así como los conceptos manejados por el analista que no son del conocimiento del usuario. Se exceptúan los conceptos que son de "dominio público", según el perfil que se haya elegido como referencia.

Por ejemplo, no sería necesario incluir en un glosario el término "IMSS (Instituto Mexicano del Seguro Social)" si se supone que el analista y los usuarios son mexicanos y el sistema no saldrá de dicho país. No obstante, si es un sistema para uso internacional sí es necesario aclararlo.

El flujo de trabajo de análisis

Los objetivos del flujo de trabajo de análisis

El flujo de trabajo de análisis cubre un objetivo básico: tener una visión general del sistema, que abarca tanto las interfases hacia el usuario (que se derivan en forma directa de los casos de uso) como las que el usuario no percibe visualmente (que sirven para el manejo de procesos y datos). Dentro de estas últimas, se encuentran rutinas genéricas que sirven para varios casos de uso y apuntalan la reutilización de código. Conforme se avanza en el objetivo anterior, se cubre otro objetivo implícito: se afinan los casos de uso levantados con base en las dudas surgidas en esta labor de análisis. Aunque no se menciona de manera explícita dentro de las metodologías del proceso unificado, es muy recomendable que se incorpore un nuevo objetivo: detallar las interfases de usuario (pantallas, reportes y otras modalidades).

Al finalizar este flujo de trabajo se esperaría tener los siguientes tres elementos debidamente documentados:

- Una arquitectura preliminar global del sistema.
- Los casos de uso actualizados.
- El diseño de las interfases de usuario.

REFLEXIÓN 7.23

¿Hasta dónde llevar la diagramación UML en el aula?

En muchos casos se le solicitan al alumno diagramas UML para todos los aspectos del sistema. El resultado final en muchos casos es que el estudiante tarda tres veces más en realizar los diagramas que la codificación del sistema, con el consiguiente resentimiento hacia la documentación del sistema y el menosprecio de la diagramación. En otros casos, el estudiante entra en una etapa de documentación que parece eterna, sin llegar al código.

Desde nuestro punto de vista, son indispensables los diagramas de casos de uso; los diagramas de colaboración y secuencia tienen que aplicarse de tal forma que el alumno les vea una utilidad real y solo deben elaborarse en partes muy específicas del sistema; el diagrama de análisis puede aplicarse para la elaboración y explicación de sistemas pequeños y medianos, aunque pudieran permitirse ligeras variaciones entre dicho diagrama y el diagrama final del sistema, que quedará reflejado a través de diagramas de clase.

Por otra parte, el diseño de las interfases de usuario merece una revisión cuidadosa por parte del docente para que estas reflejen lo expresado en los casos de uso y se piense en la facilidad de uso del sistema. Esta revisión deberá realizarse antes de la construcción del sistema.

La matriz de palabras clave para el análisis del sistema

¿Cómo pasar de los casos de uso (cuyo contenido a detalle está en lenguaje del usuario y en un texto escrito) a los diagramas de análisis, que se encuentran en un modelo técnico? Una de las herramientas que pueden ser útiles es una matriz de palabras clave. Esta matriz se construye al identificar todas aquellas palabras que deberán reflejarse en el sistema, ya sea a través de clases, métodos o atributos. En teoría, todos los términos funcionales del sistema deben nacer a partir de los casos de uso y ninguna palabra clave debe quedar fuera del sistema.

Se sugieren las siguientes columnas para la matriz:

Cuadro 7.8

Palabra clave	Se reflejará como... (clase, atributo, etcétera)	Sinónimos (unificar términos)	¿Irá en glosario?	Observaciones

La matriz de palabras clave, al igual que los diagramas de análisis, es un puente entre la etapa de requisitos y el diseño. No tiene porqué mantenerse a lo largo de toda la vida del sistema.

Los diagramas de análisis

El diagrama de análisis está constituido por tres tipos de diagramas de clase:

a) El tipo borde (*boundary*), representa las interfases hacia el usuario. En él se reflejan los informes, pantallas de captura y cualquier otra modalidad de interacción con el usuario final.
b) El tipo control (*control*), sirve para representar las clases que contendrán las reglas de negocio. En él están los procesos del sistema.
c) La clase entidad (*entity*), refleja las clases relacionadas con los datos del sistema.

Estos diagramas proporcionan un vistazo global muy rápido y su elaboración es muy sencilla. A manera de ejemplo, observe el siguiente diagrama, que refleja una sencilla petición de información acerca de promociones por parte de un profesor. Es fácil de identificar el actor, las pantallas de captura ("solicitar promoción" y "respuesta de promoción"); la clase de proceso ("verificar requisitos") y la de datos ("datos del docente"). Las clases de borde por lo común se realizarán a través de herramientas semiautomatizadas como ambientes visuales o entornos de desarrollo web; las de proceso, serán código escrito en forma directa por el programador. Las clases de entidad, por su parte, se concretan en clases para la persistencia de datos y tablas de bases de datos relacionales.

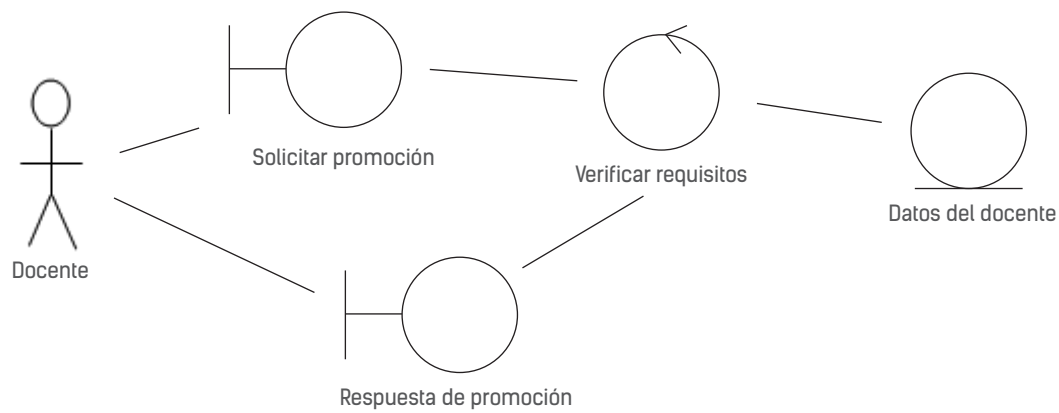


Figura 7.11 Diagrama de análisis de una promoción docente.

¿Vale la pena realizar el diagrama de análisis? Esta pregunta tiene al menos tres variantes:

a) ¿Vale la pena hacerlo durante el flujo de análisis?
b) ¿Vale la pena actualizarlo durante los flujos de trabajo posteriores?
c) ¿Vale la pena actualizarlo para el mantenimiento del sistema?

Recuerde que la documentación es una situación de costo-beneficio; solo debe elaborarse la documentación cuando el beneficio obtenido es mayor al costo de realizarla. Cada empresa de software debe dar su propia respuesta. En principio, recomendaríamos hacer este diagrama como un primer acercamiento a la arquitectura, como un "puente" entre la etapa de requisitos y el diseño. Al llegar al diseño, dejar de actualizar estos diagramas.

Los diagramas de colaboración

Los diagramas de colaboración y secuencia expresan a detalle la interacción de diversas clases para un caso de uso. Son útiles en particular cuando dicha interacción no es obvia. Sin embargo, debe tenerse cuidado en su uso, sobre todo por el tiempo que consumen en su desarrollo. Habría que restringirlo solo a aquellos casos en que se juzga indispensable su uso.

El planteamiento de las interfases al usuario

La arquitectura preliminar reflejada en el análisis permite llevar a detalle las interfases con el usuario (pantallas y reportes), que constituyen el resultado más visible de este flujo de trabajo. De hecho, el usuario normal le entiende más a estas pantallas que a los casos de uso, por lo cual es conveniente que se sometan a consideración de los diversos actores que participarán en el sistema.

Para el diseño de las pantallas y reportes deben utilizarse las herramientas que proporcionen un mejor costo-beneficio (ambientes integrados en modo visual o web, reporteadores, simuladores, etc.). En términos generales, dichas herramientas se pueden elegir bajo dos enfoques:

a) Cualquier herramienta que permita elaborarlas de manera simulada, con poco esfuerzo y con una presentación similar a la presentación final, a sabiendas de que el código que está ahí no será utilizado después.
b) En la herramienta definitiva, con el propósito de que el código sea aprovechado en gran medida. En este caso, se requiere que previamente se haya realizado el diseño gráfico y ya se tengan todos los estándares de presentación (colores, tipos de letra, logotipos, etc.). La labor previa del diseñador gráfico es indispensable.

Cualquier ruta que se siga debe ser validada antes de ser presentada al usuario por un analista, que piensa en los requerimientos del usuario y la facilidad de uso del sistema, y por un programador que revisa la viabilidad y dificultad de lograr lo expresado en dichas interfases. Al final, ambos enfoques deben llegar a un punto de acuerdo.

Siempre surge un cuestionamiento interesante: ¿las interfases hacia el usuario deben desarrollarse durante el análisis o el diseño? Si se realizan durante el análisis se corre el riesgo de presentar un resultado que no sea viable, pues aún no se tienen todos los elementos técnicos. Si se deja al diseño, desaparece este riesgo, pero a costa de construir un software que tal vez sea muy diferente a las expectativas del usuario final.

Ante esta disyuntiva, nos inclinamos porque esas interfases sean elaboradas durante el análisis y revisadas en el diseño. El criterio es más práctico que conceptual. Para un gran número de usuarios y un gran número de analistas, las dudas "finas" de los requisitos surgen a la hora de esbozar cómo quedará el sistema al final. Proseguir sin este paso, resulta riesgoso para ambas partes.

El flujo de trabajo del diseño

Los objetivos del diseño se pueden definir en tres vertientes:

a) Garantizar que la arquitectura del software en construcción soportará todos los requisitos funcionales y no funcionales especificados en la etapa de requisitos y afinados en la etapa de análisis.
b) Permitir el desarrollo del software por un equipo de trabajo con diferentes habilidades (desde diseño gráfico hasta creación de algoritmos especializados), evitando que el desarrollo esté "esclavizado" a un solo programador.
c) Facilitar el mantenimiento del software.

Para lograr estos objetivos existen algunas reglas que deben respetarse:

- Garantizar la consistencia de la información, evitando la redundancia de la misma, lo cual implica que toda información manejada por el sistema debe tener una fuente única. La estrategia concreta se logra con diferentes técnicas, como las expresadas en el cuadro 7.9.
- Partir el sistema en módulos lo más independientes posibles a fin de que cada módulo pueda ser construido y probado de manera independiente y que el acoplamiento con otros módulo sea relativamente bajo.
- Adquirir un conocimiento profundo de las posibilidades y restricciones de las herramientas tecnológicas utilizadas: sistemas operativos, sistemas manejadores de base de datos, lenguajes de programación, componentes reutilizables (adquiridos o desarrollados por la propia organización), reporteadores, etcétera.
- Utilizar un esquema por capas que permita explotar de manera debida los conocimientos y habilidades de los diferentes miembros del equipo de trabajo. En general, la capa de vista está ligada a personas con nociones de diseño gráfico y que hacen hincapié en la usabilidad del sistema; la capa de reglas de negocio "captura" las reglas de la organización; y la capa de datos se centra en aspectos de explotación de la información con alta eficiencia y garantía de concurrencia.

Cuadro 7.9 Algunas técnicas para evitar la redundancia de datos en el software

Para evitar la redundancia en...	Se aplica...
Base de datos	Proceso de normalización de base de datos.
Tipos de letra y colores	Hojas de estilo.
Conexión a internet	Archivo de configuración.
Reglas de negocio	Clases que controlen las reglas de negocio, a las cuales llamarán las demás cuando deseen aplicar dichas reglas. Estas clases forman parte de la capa "modelo" en el enfoque modelo-vista-controlador.
Acceso a bases de datos	Clases que controlen el acceso a las bases de datos, a las cuales llamarán las demás cuando deseen dicho acceso. Estas clases forman parte de la capa "modelo" en el enfoque modelo-vista-controlador, procurando separarlas de las clases de reglas de negocio.
Direcciones de red	Configuración de la conexión en un solo archivo, el cual utilizarán todos. Dicho archivo deberá contener solo direcciones lógicas.

Para lograr estos objetivos se parte de la arquitectura esbozada en el análisis y se especifican a detalle las clases y subsistemas con los cuales trabajará el sistema, tanto aquellos adquiridos como los desarrollados por los programadores de la organización. Estas clases tienen que controlar en su conjunto la presen-

tación hacia el usuario final, las reglas de negocio y la persistencia de la información (dentro de ellas, las bases de datos). Por otra parte, se revisan las interfases de pantallas y reportes esbozados en el análisis y se les hacen las adecuaciones necesarias para que sean acordes al diseño definitivo. Cuando se tiene un diagrama de clases se está a un paso de la construcción física del software.

El diseño y los diagramas de clase

La forma de representar las clases empleando la notación de UML (Lenguaje Unificado de Modelado – *Unified Modeling Language*) es mediante un rectángulo con secciones. La primera sección (el nombre de clase) es forzosa. Las demás son opcionales, pero debe respetarse el orden indicado. La segunda parte indica los atributos; la tercera, los métodos; en la cuarta se colocan comentarios. En este tipo de diagramas la parte privada se maneja como – y la parte pública con +. El diagrama será similar al esquema de la figura 7.12. El mecanismo de herencia se señala mediante una flecha que va de la clase hija a la clase padre.[25]

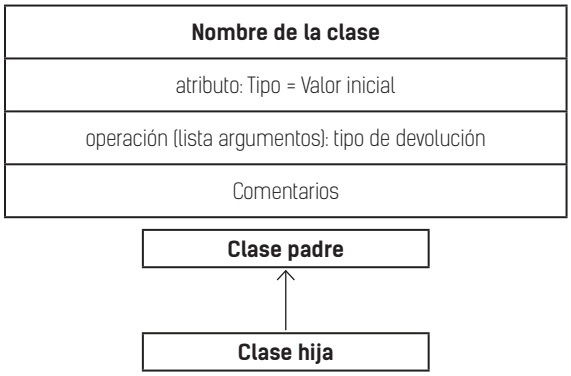


Figura 7.12 Representación de una clase en UML.

El diseño y el diagrama de capas

El diagrama de capas permite establecer el software requerido en los diferentes niveles para que un software pueda ser ejecutado. Esto tiene al menos tres finalidades: prever los requerimientos de hardware; visualizar los aspectos de portabilidad y analizar posibles cambios requeridos por cambios en el entorno. El análisis de la portabilidad de un sistema se facilita si existe un buen diagrama de capas.

La utilidad del diagrama de capas se visualiza en el ejemplo de la figura 7.13. Como puede observarse, la instalación del sistema en el lado del servidor requiere Internet Information Server (IIS) y SQL Server 2000. También necesita el sistema operativo Windows 2000 o XP Professional, pero este es un requerimiento indirecto, que se desprende de los ambientes en los cuales se ejecutan IIS y SQL Server 2000. La pregunta: ¿correrá en Windows Vista? no debe plantearse de esa forma. Más bien, debe decirse: ¿en qué versiones de Windows Vista corren sin problema el IIS 5.1 y SQL Server 2000? O incluso: ¿las herramientas de IIS y SQL Server 2000 pueden sustituirse por otras que mantengan compatibilidad con las versiones manejadas y que se ejecuten sin problemas en Windows Vista?

[25] Fowler, Martin. *UML gota a gota, op. cit.*, 1999.

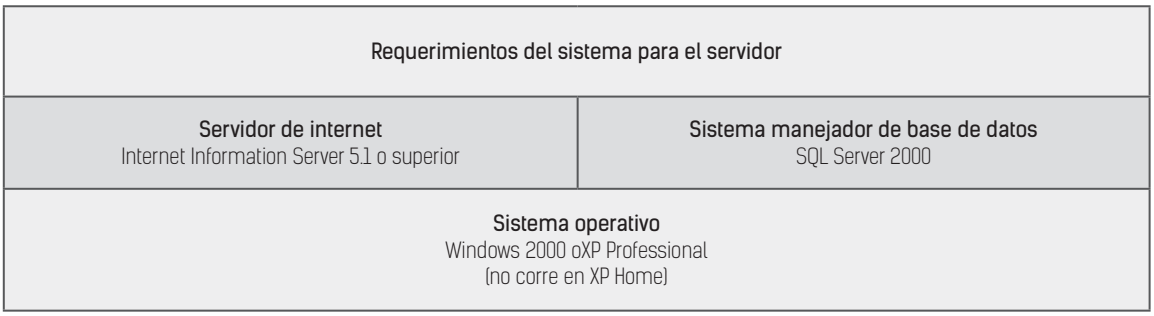


Figura 7.13 Ejemplo de un diagrama de capas.

Modularidad

La división modular o modularidad es un punto primordial en el flujo de trabajo de diseño y debe quedar terminada durante la etapa de elaboración. Su finalidad es dividir el código en partes relativamente independientes, cada una de ellas con un propósito específico y con entradas y salidas definidas con claridad. Los objetivos de esta división son:

- Facilitar la programación, pues es más fácil hacer un módulo que el sistema completo.
- Permitir un trabajo interdisciplinario, que logre incorporar las diferentes habilidades y conocimientos del equipo de trabajo: especialistas en programación, analistas, diseñadores gráficos, especialistas en base de datos, etc. Por lo regular, cada parte solo requiere la participación de un especialista pero está diseñada de tal modo que combina de manera adecuada con las otras partes del sistema.
- Facilitar el mantenimiento, pues cada parte es relativamente independiente y puede modificarse sin afectar al sistema en su conjunto.
- Permitir la reutilización del código en otros sistemas.
- Evitar la redundancia. Cualquier dato del sistema (reglas de negocio, configuración, colores, etc.) debe aparecer solo en una parte del sistema.
- Comprar software de propósito específico que ayude al proyecto y mejore su relación costo-beneficio.

La modularidad no se refleja en un único diagrama. Es un concepto que debe quedar implícito en el diagrama arquitectónico del sistema o, al menos, en el diagrama de clases. En ellos deben estar indicadas:

- Clases que aglutinen rutinas de uso general para el sistema. Por ejemplo, conexión a la base de datos.
- Archivos que guarden información de uso general para el sistema. Por ejemplo, hojas de estilo.
- La división por capas, reflejado por lo regular en modelos reconocidos (por ejemplo, modelo vista-controlador) e incluso modelos específicos (Struts, Java Server Faces, etcétera).

El flujo de trabajo de implementación

Durante la codificación se transforman las clases a código, basándose en todos los documentos previos que la metodología ha arrojado (véase cuadro 7.10), todos ellos —o casi todos— diagramados en UML.[26] Cada uno de estos requisitos se cristaliza en conceptos específicos. Por ejemplo, los estándares de presen-

[26] De nuevo debe insistirse en un concepto: la metodología "manda" sobre la diagramación UML. El proceso unificado sería demasiado ambiguo sin UML, mientras UML sin metodología pierde su potencial. ¿Qué diagramas realizar? Como ya se ha mencionado, aquellos cuyo costo invertido en hacerlos sea menor al beneficio obtenido. Indispensables: casos de uso y diagramas de clase. A partir de ellos, agregar los que sean justificables. Para comenzar con UML, recomendamos el libro de Martin Flower, *UML gota a gota, op. cit.*

tación quedan expresados en las hojas de estilo. En muchas ocasiones, la transformación a código se basa en herramientas automatizadas. Algunas recomendaciones a seguir en la implementación se sintetizan en el cuadro 7.11.

Cuadro 7.10 Elementos previos a la codificación de clases.

Requisitos	Casos de uso detallados
	Matriz de accesos por perfil
	Diagrama de actividades
Análisis:	Diagrama de análisis
	Estándares de presentación
	Diseño de interfases de usuario
	Diagrama de estados
Diseño	Estándares de seguridad y configuración
	Diagramas de clase, de paquetes, multiplicidades y componentes
	Diagrama entidad-relación
Implementación	Estándares de programación, seguridad y presentación hacia el usuario
	Esquema de pantallas a obtener

Cuadro 7.11 Algunas consideraciones básicas en la implementación (codificación)

Procesos batch	Verificar que los procesos batch reportan en forma adecuada los casos anómalos (inconsistencia de datos, carga inconclusa, etc.) y pueden volverse a ejecutar sin problema.
Validaciones en las pantallas	Que sea comprensible por sí misma, con ayuda de las instrucciones de la pantalla o, al menos, con la ayuda en línea.
	Que use los estándares de presentación establecidos (por ejemplo, hojas de estilo).
	Que valide si se han llenado todos los campos necesarios.
	Que valide la consistencia de cada dato (por ejemplo, fechas coherentes).
	Que no permita exceder la longitud máxima permitida.
	Que funcionen con las resoluciones de monitor especificadas.
Manejo de reportes	En caso de reportes, que existan filtros por las opciones más comunes manejadas por el usuario.
	Que exista la posibilidad de imprimir hojas específicas para reportes grandes.
	Que exista alguna forma de exportación del reporte a un formato electrónico de uso general (por ejemplo, PDF).
	Que exista la posibilidad de previsualización del reporte antes de su impresión.
Límites al tamaño de archivos a cargar	Que el tamaño de archivo a cargar (imágenes, sonido, etc.) no exceda lo establecido en las especificaciones.
Manejo de excepciones	Todas las operaciones que puedan arrojar un error (de conexión, de división entre cero, de formato erróneo, etc.) deben manejar excepciones.
	Que las excepciones devuelvan un texto orientador para facilitar la ubicación de errores.
Estándares en el nombre de identificadores	Que los nombres de los identificadores (variables y clases) estén conforme a los estándares de programación establecidos.
	Los nombres de las clases deben comenzar con mayúsculas.
	Los nombres de las variables y paquetes deben comenzar con minúsculas.
	Se debe utilizar la notación de camello para el nombre de clases y variables.
	Todos los nombres de tablas deben ser en plural o singular. No revolver nombres en singular con nombres en plural. La misma política se aplica para los nombres de campos.

Perfiles de acceso a nivel aplicación	Que los distintos usuarios solo puedan acceder a las funcionalidades permitidas.
	Que los distintos usuarios no puedan acceder a las pantallas del sistema "brincándose" su autentificación.
	Que las pantallas no proporcionen información que permita "brincarse" la seguridad de los sistemas. Por ejemplo: la clave de acceso en la llamada a otras páginas web.
Pruebas	Que se hayan probado con casos típicos y especiales, con usuarios reales independientes del equipo de desarrollo, y se hayan corregido todas las observaciones de los usuarios.
	Que todas las mejoras sugeridas para el módulo y otros módulos se hayan incorporado o hayan quedado reflejadas en un documento de mejoras futuras.
	Si el caso lo requiere, que se hayan realizado pruebas de carga máxima y pruebas en paralelo.
Acceso a la base de datos	Verificar que todo el código acceda a la base de datos a través de las rutinas genéricas realizadas para tal fin (si es el caso, a través del pool de conexiones).
	Las conexiones a la base de datos deben cerrarse, independientemente de si se termina de manera adecuada el proceso o por una excepción.
	Que las consultas a la base de datos que devuelva un gran volumen de información explote en forma adecuada la indexación a la base de datos.
	Revisar la codificación para verificar que todos hacen la conexión a la base de datos y cierran las conexiones a nivel finally. En el caso de consultas grandes, que estas sean eficientes; es decir, que se realizan a través de campos indexados.
Diseño de la base de datos	Que no exista redundancia en la base de datos y esta se encuentre normalizada.
	Que los perfiles de acceso a la base de datos correspondan a la matriz de permisos establecida.
	Que se hayan establecido las integridades referenciales correspondientes.
	Que los índices manejados sean los adecuados para hacer eficientes las búsquedas comunes.
	Que los nombres de campos y tablas se encuentren conforme a los estándares de programación.

7.12 Los niveles de madurez de procesos de CMM

Los niveles de madurez de procesos de CMM permiten determinar el "nivel de madurez" de un área de desarrollo de software. Podríamos decir que miden más a la escuela que al propio estudiante. En México, fueron adaptados para la aplicación a empresas de software en una norma oficial de aplicación no obligatoria conocida como MoProSoft (Norma NMX-I-006-NYCE-2004), elaborada por Tecnológico Nyce. Sin duda, pueden ser un referente para evaluar a una institución de computación o informática.

Antes de entrar a la descripción de cada uno de los niveles de CMM, conviene presentar la escala que aplica MoProSoft para describir si se logra o no un requisito de la capacidad del proceso.[27]

"N: No logrado (0 a 15%), hay muy poco o incluso ninguna evidencia de cumplimiento del atributo definido.
"P: Parcialmente logrado (>15 a 50%), hay alguna evidencia de una aproximación sistemática al cumplimiento del atributo.
"L: Considerablemente logrado (*) (>50a 85%), hay evidencias claras de una aproximación sistemática al cumplimiento significativo del atributo. La ejecución del proceso puede variar en algunas áreas o unidades de trabajo.
"F: Completamente logrado (**) (más de 85%), hay evidencias claras de una aproximación sistemática para el cumplimiento total del atributo. No hay debilidades significativas a lo largo de las unidades de trabajo."

[27] Tomado de Norma NMX-I-006-NYCE-2004, elaborada por Tecnológico Nyce.

Los niveles de madurez en CMM[28]

"(…) Uno de los objetivos del modelo de madurez de capacidades (CMM, por sus siglas en inglés) es proporcionar lineamientos para la selección de las actividades del ciclo de vida. El CMM asume que el desarrollo de los sistemas de software se hace más predecible cuando una organización usa un proceso de ciclo de vida bien estructurado, visible para todos los participantes en el proyecto y que se adapta al cambio. El CMM usa los siguientes cinco niveles para caracterizar la madurez de una organización.

Nivel 1: Inicial

Una organización que está en el nivel inicial aplica actividades *ad hoc* para desarrollar software. Pocas de estas actividades están bien definidas. El éxito de un proyecto en este nivel de madurez depende, en general, de los esfuerzos heroicos y habilidades de individuos clave. Desde el punto de vista del cliente, el modelo de ciclo de vida del software, si es que existe, es una caja negra: después de proporcionar el enunciado del problema y negociar el acuerdo del proyecto, el cliente debe esperar hasta el final del proyecto para inspeccionar los productos a entregar del proyecto. Durante todo el proyecto el cliente no tiene una forma efectiva para interactuar con la administración del proyecto.

Nivel 2: Repetible

Cada proyecto tiene un modelo de ciclo de vida del software bien definido. Sin embargo, los modelos difieren entre proyectos, al reducir la oportunidad del trabajo en equipo y la reutilización de conocimiento. Los procesos básicos de administración del proyecto se usan para el seguimiento de los costos y la calendarización. Los nuevos proyectos se basan en la experiencia de la organización con proyectos similares anteriores, y el éxito es predecible para proyectos en dominios de aplicación similares. El cliente interactúa con la organización en momentos bien definidos, como las revisiones del cliente y la prueba de aceptación del cliente, sí se permiten algunas correcciones antes de la entrega.

Enfoque: establecer los controles de administración de proyectos básicos.

- **Administración de los requerimientos:** los requerimientos se establecen como línea base en un acuerdo de proyecto y se mantienen durante el proyecto.
- **Planeación y seguimiento del proyecto:** se establece un plan de administración del proyecto de software al inicio del proyecto y se le da seguimiento durante la ejecución del mismo.
- **Administración del subcontratista:** la organización selecciona y administra de manera efectiva a subcontratistas de software calificados.
- **Administración del aseguramiento de la calidad:** todos los productos a entregar y las actividades del proceso se revisan y auditan para verificar que se apeguen a los estándares y lineamientos adoptados por la organización.
- **Administración de la configuración:** un conjunto de artículos de la administración de la configuración se define y mantiene durante todo el proyecto.

Nivel 3: Definido

Este nivel usa un modelo de ciclo de vida del software documentado para todas las actividades administrativas y técnicas de toda la organización. Al inicio de cada proyecto se produce una versión personalizada del modelo durante la actividad *Modelado del ciclo de vida*. El cliente conoce el modelo estándar y el modelo seleccionado para el proyecto específico.

[28] Tomado de Bruegge, Bernd y Dutoit, Allen H. *Ingeniería de software orientado a objetos*. Prentice Hall, Pearson, México, 2002, pp. 468-471.

Enfoque: establecer una infraestructura que permita un solo modelo de ciclo de vida del software efectivo en todos los proyectos.

- **Enfoque del proceso de la organización:** la organización tiene un equipo permanente para mantener y mejorar el ciclo de vida del software.
- **Definición del proceso de la organización:** se usa un modelo de ciclo de vida del software estándar para todos los proyectos de la organización. Se usa una base de datos para la información y documentación relacionadas con el ciclo de vida del software.
- **Programa de entrenamiento:** la organización identifica las necesidades de entrenamiento para proyectos específicos y desarrolla programas de entrenamiento.
- **Administración integrada del software:** el software se construye de acuerdo con el ciclo de vida del software y con los métodos y herramientas definidos.
- **Coordinación entre grupos:** los equipos del proyecto interactúan con los demás equipos para resolver los requerimientos y problemas.
- **Revisiones entre iguales:** el desarrollador examina los productos a entregar en un nivel entre iguales para identificar defectos potenciales y áreas donde se necesiten cambios.

Nivel 4: Administrado

Este nivel define las medidas para las actividades y los productos a entregar. Los datos se recopilan en forma constante durante todo el proyecto. En consecuencia, el modelo de ciclo de vida del software puede comprenderse y analizarse de modo cuantitativo. Al cliente se le informa de los riesgos antes de que se comience el proyecto y conoce las medidas utilizadas por la organización.

Enfoque: comprensión cuantitativa de los procesos y productos a entregar del ciclo de vida del software.

- **Administración cuantitativa de procesos:** las medidas de productividad y calidad se definen y miden en forma constante a lo largo del proyecto. Es esencial que estos no se usen de inmediato en el proyecto, en particular para valorar el desempeño de los desarrolladores, sino que se guarden en una base de datos para que puedan compararse con otros proyectos.
- **Administración de la calidad:** la organización ha definido un conjunto de objetivos de calidad para los productos de software. Supervisa y ajusta los objetivos y productos para entregar productos de alta calidad al usuario.

Nivel 5: Optimizado

Los datos de mediciones se usan en un mecanismo de retroalimentación para mejorar al modelo de ciclo de vida del software a lo largo de la vida de la organización. El cliente, los gerentes del proyecto y los desarrolladores se comunican y trabajan juntos durante el desarrollo del proyecto.

Enfoque: dar seguimiento a los cambios de tecnología y procesos que pueden causar cambios en el modelo del sistema o en los productos a entregar, incluso durante todo un proyecto.

- **Prevención de defectos:** se analizan las fallas de proyectos anteriores usando datos de la base de datos de mediciones. Si es necesario se toman acciones específicas para prevenir que vuelvan a repetirse.
- **Administración del cambio de tecnología:** se investiga en forma constante a los facilitadores de tecnología y a las innovaciones, y se comparten en toda la organización.
- **Administración del cambio de procesos:** se refina y cambia en forma constante el proceso de software para manejar las ineficiencias identificadas por las mediciones del proceso del software. El cambio constante es la norma y no la excepción.

REFLEXIÓN 7.24

Los niveles de madurez y la cotidianidad del aula

Sugerimos la aplicación de los niveles de madurez de procesos para medir el grado de avance del desarrollo de software en una escuela.

Nivel 0: incompleto. El programa no se ejecuta de manera adecuada (no tiene corrección).
Nivel 1: realizado. Se ejecuta bien pero no se aplicaron normas metodológicas.
Nivel 2: administrado. Se ejecuta bien y se aplicaron normas metodológicas básicas.
Nivel 3: establecido. Se ejecuta bien y se aplicaron normas metodológicas básicas consensuadas a nivel de grupo o de la escuela.
Nivel 4: predecible. Se ejecuta bien y se aplicaron normas metodológicas básicas consensuadas a nivel de grupo o de la escuela. Existen indicadores para medir aspectos básicos del desarrollo (tiempos de desarrollo, entrega oportuna, cumplimiento de la funcionalidad y respeto de estándares).
Nivel 5: optimizado. Se ejecuta bien y se aplicaron normas metodológicas básicas consensuadas a nivel de grupo o de la escuela. Existen indicadores para medir aspectos básicos del desarrollo (tiempos de desarrollo, entrega oportuna, cumplimiento de la funcionalidad y respeto de estándares), con los cuales se verifica si la metodología es adecuada y acorde a la tecnología existente.

7.13 Megasinapsis: banco de información sobre desarrollo de software

Suponemos a esta altura que quienes hayan adquirido este libro se darán cuenta de las implicaciones que exige un desarrollo de software, por demás inalcanzables en un libro introductorio. Tal vez tengan una sensación agridulce. Por una parte, un acercamiento práctico y sencillo a la programación; por otro, temas que se dejaron de lado. ¿Se podría intentar un manual de referencia "completo"? Es posible, pero se sacrificaría la concisión del material y el costo asequible para cualquier estudiante.

¿Qué hacer con muchos temas que quisieran ser abordados? Recomendamos un banco de información conformado de manera colectiva que recae en un equipo de trabajo prácticamente voluntario coordinado, justo, por uno de los autores: www.megasinapsis.com.mx ¿El lema? *Enlacemos el conocimiento* (véase figura 7.14).

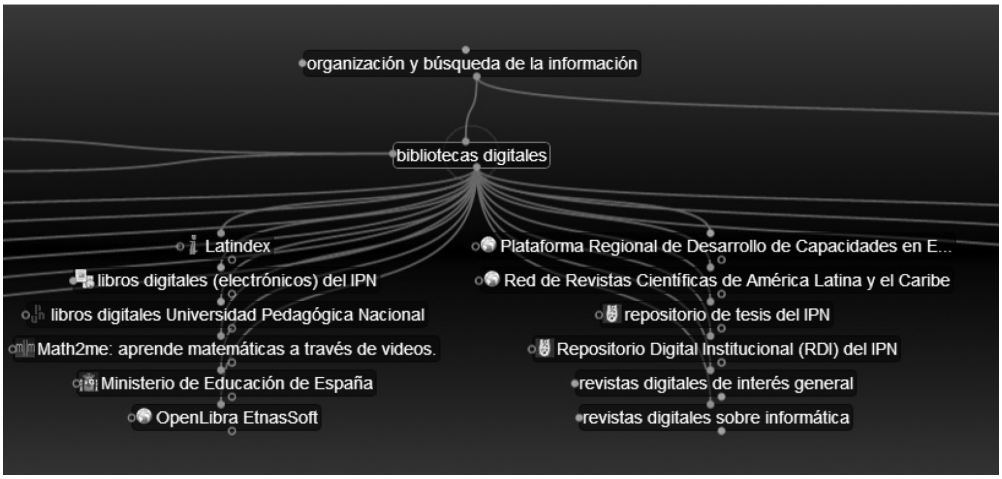


Figura 7.14 El sitio www.megasinapsis.com.mx; complemento de acceso gratuito al material de este libro.

El enfoque de este sitio es novedoso: "los bancos de información viva y organizada de la comunidad a partir de la documentación más valiosa de su interés (producida por ella misma o externa), con independencia de la diversidad de los formatos; una forma de 'sobrevivencia' ante la inmediatez de las redes sociales y lo limitado del material que se encuentra disponible por los canales 'oficiales' que se dan en la propia comunidad."[29]

La entrada al sitio es gratuita y los materiales van desde un enfoque introductorio hasta temas avanzados de desarrollo de software y formación integral de la juventud. Este libro es, pues, un comienzo rápido y accesible hacia el desarrollo de software; el sitio, una biblioteca digital colectiva para profundizar en los temas. Hay que dejar claro un punto: la entrada al sitio no está condicionada a la compra del libro, pues tanto el banco de información como el libro son esfuerzos complementarios que por vías diferentes tienen el mismo objetivo: la divulgación del conocimiento.

Si hemos avanzado en ello, el esfuerzo ha valido la pena.

[29] López Goytia, José Luis, César Cruz, Ana Karen. "*Proyecto megasinapsis: enlacemos el conocimiento*", NotiUpiicsa, año 4, número 7, octubre-diciembre de 2013.

Apéndice

¿Cómo saber si un algoritmo es eficiente?

A.1 Introducción

Recordemos dos definiciones importantes que se han manejado a lo largo del libro:

- **Corrección** es la capacidad de los productos de software para realizar con exactitud sus tareas [y sin errores], tal como se definen en las especificaciones.
- **Eficiencia** es la capacidad de un sistema de software para exigir la menor cantidad posible de recursos de hardware, como tiempo del procesador, espacio ocupado de memoria interna y externa o ancho de banda utilizado en los dispositivos de comunicación.[1]

Se puede analizar la eficiencia de un algoritmo siempre y cuando cumpla con la primera condición: que sea correcto; es decir, que realice la labor que debe hacer. Si es así, entonces se puede pasar al segundo cuestionamiento: ¿lo hace de manera eficiente? Por ello, la mejor manera de establecer la eficiencia de un algoritmo es comparar el consumo de recursos con otros algoritmos que realizan la misma labor. La estimación del consumo de recursos es indispensable para intentar resolver otro cuestionamiento: ¿cómo se comportará el sistema cuando el volumen de datos crezca de manera significativa?

Este tipo de preguntas resulta crucial, pues puede ayudar a anticipar el desempeño de sistemas con fuerte cantidad de transacciones. Por ejemplo, sistemas bancarios con millones de transacciones al día o sistemas de inscripciones de instituciones con más de 100 000 alumnos, como la UNAM o el IPN.

El concepto va más allá de los algoritmos computacionales para adentrarse a los sistemas de información en su conjunto. Citemos un ejemplo actual: en México se cambió el régimen fiscal para los pequeños contribuyentes, quienes tienen que entender el nuevo esquema e ir al Servicio de Administración Tributaria (SAT) para realizar una parte del trámite correspondiente. ¿Está preparado el SAT para recibir a más de 3 millones de pequeños contribuyentes (ahora Repecos)? ¿Qué sucederá con las personas que no tienen la mínima familiaridad con aspectos fiscales y de cómputo, muchas de ellas mayores de 60 años? ¿Se consideró la problemática del medio rural? Quizá deba entenderse el contexto: los pequeños contribuyentes pueden ser empresas formales y con conocimientos sobre tecnologías de información, pero también incluyen muchas misceláneas, papelerías, tiendas de abarrotes, etcétera, atendidas por dos o tres familiares y no suelen llevar contabilidad alguna. El "algoritmo" de cobro de impuestos tiene que ser acorde con la realidad en un contexto diverso que abarca más de 3 millones de casos.

Sirva el ejemplo anterior para recalcar un concepto: debemos tratar de anticipar el comportamiento de un algoritmo o un sistema de información cuando trabaje con la carga máxima y también debemos preguntarnos si es la forma más eficiente de resolver un problema. Debemos anticipar que la respuesta no es obvia. ¡Bienvenidos al universo de la complejidad computacional!

A.2 La función de trabajo, un primer acercamiento al consumo de recursos

La función de trabajo es una de las formas más comunes de acercarse a la medición del consumo de recursos de los algoritmos.

Para comenzar un acercamiento intuitivo al tema, compare estos dos códigos que cubren exactamente la misma tarea (véase programa A.1).

[1] Los conceptos citados fueron tomados de Meyer, Bertrand. *Construcción de software orientado a objetos*. PrenticeHall, segunda edición, Madrid, 1999, pp. 3-13.

REFLEXIÓN A1

Acercarse a la complejidad por múltiples caminos

La forma más precisa de acercarse a la función de trabajo es desde un punto de vista analítico. Sin embargo, a veces no es posible tener acceso a un algoritmo concreto, o bien, su análisis puede ser demasiado complejo, sobre todo en el ámbito de aplicación. En estos casos los otros métodos de medición son perfectamente justificables desde un enfoque práctico y didáctico, sobre todo porque muchos estudiantes no creen los resultados obtenidos hasta que lo observan "en vivo".

Por último, conviene hacer hincapié en que los efectos principales se observan en un gran volumen de datos. Es ideal combinar números aleatorios —la estrategia más socorrida— con archivos de texto sumamente grandes y bases de datos de volumen considerable.

Programa A.1 Dos subrutinas equivalentes de sumatoria.

```
longintsumatoria (int a) {
longintsuma = 0;
inti;
if (a == 0)
return 0;
else {
for (i = a; i> 0; i--)
suma += i;
return(suma);
 }
}


longintsumatoria (intdato) {
return (dato * (dato + 1) / 2);
}
```

De hecho, constituyen rutinas intercambiables y equivalentes; cualquiera de ellas puede combinar con el mismo programa principal (véase programa A.2).

REFLEXIÓN A2

Manejemos con claridadel concepto de eficiencia

La eficiencia suele confundirse con otras características del software. Por ejemplo, si un software falla se suele decir que no es eficiente, cuando en realidad el software no es correcto. O si no es fácil de manejar se dice igualmente que no es eficiente, cuando en realidad lo que no tiene es facilidad de uso.

Seamos rigurosos en cuanto a la delimitación del concepto, pues de otra manera es muy difícil de cuantificar.

Programa A.2 Programa completo de sumatoria.

```
// Programa A.2: lista de sumatoria de números
// aplicando la fórmula de sumatoria.
#include <conio.h>
#include <stdio.h>
longintsumatoria (int a);
int main() {
int j;
```

```
intsumaux;
printf("Este programa genera una tabla de sumatorias.\n\n");
printf(" NUMERO SUMATORIA\n");
for (j=0; j<=15; j++) {
sumaux = sumatoria(j);
printf("%10d %10d\n", j, sumaux );
 }
printf("\n\nOprima cualquier tecla para continuar...\n");
getch();
}

longintsumatoria (intdato) {
return(dato*(dato+1)/2);
}
```

En este caso, resulta fácil detectar que la subrutina que aplica la fórmula de sumatoria en forma directa es más eficiente que la que hace la sumatoria por sumas sucesivas. Pero no basta una conclusión cualitativa, hay que medir la cantidad de recursos consumidos. Al menos existen tres enfoques complementarios para medir la eficiencia de un algoritmo.

- Medición física: se coloca código para pedir hora inicial y final de la ejecución, o bien, se realiza "reloj en mano".
- Pistas de auditoría: se establecen líneas de auditoría que indiquen el número de veces que se utiliza la instrucción más empleada.
- Función de trabajo: se deduce de manera analítica una fórmula que indique el número de veces que se utiliza la instrucción más empleada.

¿Cómo deberían utilizarse estos enfoques para el ejemplo de la sumatoria?

Medición física del tiempo. Se coloca un código para pedir la hora inicial antes de entrar a la parte de la sumatoria. No lo haga antes de pedir datos al usuario, pues puede falsear el resultado. Después de terminar la sumatoria se pide de nuevo la hora del sistema y, por diferencia, se puede obtener el tiempo transcurrido en la sumatoria en esa corrida del programa. Este método podría realizarse con un cronómetro físico, a falta de código fuente.

Lo ideal sería correr el programa con diferentes valores (10, 100, 1000, 10000, etc.) y crear una tabla para tratar de deducir el comportamiento del algoritmo conforme aumenta el número de datos (véase programa A.3).

Programa A.3 Sumatoria solicitando la medición de tiempos.

```
#include <conio.h>
#include <stdio.h>
longintsumatoria (int a);
int main() {
int j;
longintsumaux;
printf("Este programa genera una tabla de sumatorias.\n\n");
printf(" NUMERO SUMATORIA\n");
// código para pedir hora inicial
for (j=0; j<=15; j++) {
sumaux = sumatoria(j);
printf("%15d %15d\n", j, sumaux );
 }
// código para pedir hora final
```

```
// obtener y desplegar la diferencia de tiempos
printf("\n\nOprima cualquier tecla para continuar...\n");
getch();
}

longintsumatoria (int a) {
longintsuma = 0;
inti;
if (a == 0)
return 0;
else {
for (i = a; i> 0; i--)
suma += i;
return(suma);
 }
}
```

Pistas de auditoría. Se coloca un contador que aumente cuando pasa por el código de las instrucciones que se trata de rastrear (por lo regular, los comandos que más se usan). Al final de la ejecución se pide que se despliegue el resultado del contador. Al igual que en la medición física, se aconseja crear una tabla para tratar de deducir el comportamiento del algoritmo conforme aumenta el número de datos (véase programa A.4).

Programa A.4 Sumatoria con contador sobre la instrucción de mayor uso.

```
#include <conio.h>
#include <stdio.h>
longintsumatoria (int a);
int main() {
int j;
longintsumaux;
printf("Este programa genera una tabla de sumatorias.\n\n");
printf(" NUMERO SUMATORIA\n");
// aquí se inicializa un contador dado de alta como variable global
for (j=0; j<=15; j++) {
sumaux = sumatoria(j);
printf("%15d %15d\n", j, sumaux );
 }
// aquí se despliega el valor final del contador
printf("\n\nOprima cualquier tecla para continuar...\n");
getch();
}

longintsumatoria (int a) {
longintsuma = 0;
inti;
if (a == 0)
return 0;
else {
for (i = a; i> 0; i--)
// aquí se suma 1 al contador
suma += i;
return(suma);
 }
}
```

Deducción de la función de trabajo. Se trata de deducir de manera analítica el comportamiento del algoritmo conforme aumenta el número de datos para obtener al final una fórmula que lo refleje.

En nuestro caso, observe que la instrucción que más se repite está dentro de un ciclo, el cual pasa una vez para el número 1, dos veces para el número 2, tres veces para el número 3, y así de manera sucesiva hasta el 15. ¿Qué valor reflejará el contador al final?

```
f(n) = 1 + 2 + 3 + 4 + 5 + .. + 15
 = 15 * (15 + 1) / 2
 = 120
```

En términos generales:

```
f(n) = n * (n+1) / 2
= (n2 + n) / 2,
```

En este momento conviene aclarar que las tres mediciones de la eficiencia del algoritmo reflejan el mismo comportamiento y se aplicarán aquellas que sean adecuadas al contexto. El contador es más sencillo, siempre y cuando contemos con el código fuente; la función analítica es la más precisa, aunque necesitamos el algoritmo y en ocasiones no es fácil deducir la fórmula; mientras que la medición del tiempo no necesita acceso al código fuente ni al algoritmo, pero puede verse afectada por problemas externos.

En otras palabras, los tres métodos deben convergir hacia la misma conclusión. Las pistas de auditoría deben reflejar lo que indica la función analítica. La medición física debe reflejar el comportamiento de la función de trabajo, con independencia de las variaciones según la máquina empleada (aunque pudieran verse afectadas si hay variaciones en la velocidad de la red de cómputo). En cualquier caso, el tiempo que ocupa el algoritmo en realizar su tarea es proporcional al número de operaciones que arroja la función de trabajo.

Ahora bien, ¿qué pasaría si el volumen de datos se hace sumamente grande (tiende al infinito)? Conforme crece el número de datos se pierde la importancia de *n* e, incluso, de la división entre 2. Por eso muchas veces se conserva solo la expresión con mayor impacto, lo que se llama notación O grande. En este caso, el "orden" de este algoritmo es $O(n^2)$.

$$\lim_{n \to \infty} (n^2 + n) / 2 = n^2$$

por eso se dice, de manera un tanto simplificada, que el **orden de la función** es n^2; es decir, que el orden del algoritmo es de naturaleza cuadrática.

Con pocos datos, el resultado no es significativo. Sin embargo, ¿qué pasaría si se deseara generar la tabla hasta 1 000?

```
f(n) = 1000 * 1001 / 2
 = 500500
```

¿Y para medio millón de registros?

```
f(n) = 500000 * 500001 / 2
 = 125000250000
```

No se extrañe si la medición física toma varios minutos.

Pareciera exagerado probar con 500 000 registros. Solo dos ejemplos: el número de registros en el histórico anual de una nómina de una empresa de unos 500 empleados ronda, justamente, el medio millón

de registros; algún banco de tamaño mediano y cobertura nacional realiza, en un día de baja actividad, un millón de transacciones. La función de trabajo en estos casos es parte de la vida cotidiana.

A estos niveles ganamos poco tratando de cambiar el hardware. Por lo regular, la eficiencia se mejora más por el algoritmo empleado. Es relativamente sencillo demostrar que el código simplificado de la sumatoria

```
long int sumatoria (int dato) {
   return(dato * (dato+1) / 2);
}
```

es de naturaleza lineal, pues hay tres operaciones fijas para devolver la sumatoria de cualquier número: una multiplicación, una suma y una división. Así, para sacar la tabla de los primeros 15 números se tendrán que aplicar 45 instrucciones. Si se generaliza:

`f(n) = 3 * n` , donde n es el límite de la tabla que comienza en 1.

Al aplicar la fórmula para medio millón de registros se tendría que:

```
f(n) = 500000 * 3
 = 1500000
```

Si comparamos los dos algoritmos para medio millón de datos, uno ocupa 1 500 000 operaciones, mientras que el otro emplea 125 000 250 000. ¡El tiempo se redujo prácticamente a una milésima parte en estas circunstancias! Por eso el análisis de los algoritmos constituye uno de los mejores caminos para lograr la eficiencia en el desarrollo de software.

EJERCICIOS SUGERIDOS

- Reproducir y corroborar los ejercicios, incorporar tanto la medición del tiempo que consume el algoritmo, como las pistas de auditoría. Ejecutar ambos programas, llenar la siguiente tabla para cada caso.

Número de datos	Operaciones realizadas	Tiempo ocupado
5		
50		
500		
5 000		
50 000		
500 000		

- Llevar a cabo el programa de torres de Hanoi (muy conocido en el ámbito de la computación) y analizar su comportamiento a través de los tres métodos comentados: medición física del tiempo, pistas de auditoría y deducción de la función de trabajo.

A.2 La eficiencia en los métodos de ordenamiento

En esta sección se hará una comparación entre tres algoritmos: selección directa, burbuja y quicksort a fin de comparar la eficiencia de los mismos.

El **algoritmo de selección directa** recorre todos los datos hasta encontrar el elemento mayor y lo intercambia con el último ítem. Después vuelve a explorar todo (excepto el último ítem, que ya fue colo-

cado en su lugar), identifica el elemento mayor y de nuevo lo intercambia, aunque ahora con el penúltimo elemento. Esta operación la realiza hasta que solo resta comparar y, en su caso, intercambiar dos elementos.

Una variante es localizar el elemento menor e intercambiarlo en el primer lugar. El resultado final y la eficiencia son los mismos.

A manera de ejemplo, suponga los siguientes cinco datos:

índice ››	0	1	2	3	4
dato ››	58	38	61	99	16

El primer recorrido explora los cinco elementos y ubica al mayor en la posición 3. Este elemento es intercambiado con el último elemento. El arreglo al final del primer ciclo quedaría como sigue:

índice ››	0	1	2	3	4
dato ››	58	38	61	16	99

Como se observa, el último elemento ya está colocado en su lugar. El problema se ha reducido a cuatro elementos, sobre los cuales puede realizarse un procedimiento similar.

El código final se muestra en el programa A.5. La parte central del algoritmo se encuentra en las líneas 21-36. Como observará, se tienen dos ciclos anidados. El ciclo interno —controlado por el contador j permite hacer un recorrido para ubicar el dato mayor; el valor de este dato queda en la variable `mayor` y su ubicación en `ubic`. Al finalizar este recorrido, este dato es intercambiado por el último elemento en las líneas 32-35. Este proceso se repetirá varias veces, pero sin tomar en cuenta los datos ya acomodados, lo cual es realizado por el ciclo externo controlado por el contador `i`.

Las variables comparaciones e intercambios no son parte del algoritmo; miden el número de comparaciones e intercambios realizados para completar el ordenamiento.

```
1  Programa A.5 Ordenamiento por selección directa.
2  1
3  48
4  49 /* Programa A-5: ordenamiento de números aleatorios utilizando arreglos
5  y algoritmo de selección. */
6  #include<conio.h>
7  #include<stdlib.h>
8  #include <stdio.h>
9  #include <time.h>
10
11 #define LIM 1000
12 int main()
13 {
14 inti, j, num[LIM], ubic, mayor, aux;
15 int comparaciones, intercambios;
16
17  //llena el arreglo con números aleatorios
18 srand(time(NULL));
19  for(i=0; i<LIM; i++)
20 num[i] = rand();
21
22  //ordenamiento
23  comparaciones = 0;
24 intercambios = 0;
25  for (i=LIM-1; i>=1; i--) {
26 ubic = i;
27  mayor = num[i];
```

```
 for(j=i-1; j>=0; j--) {
 if(num[j]>mayor) {
mayor = num[j];
ubic = j;
 }
 comparaciones++;
 }
//intercambia el ítem mayor con el último
aux = num[i];
num[i] = num[ubic];
num[ubic] = aux;
 intercambios++;
 }

 //despliegue de los números
for(i=0; i<LIM; i++)
printf("\n%d", num[i]);

//despliegue del número de comparaciones e intercambios
printf("\n\nHabía %d datos.", LIM);
printf("\nSe realizaron %d comparaciones.", comparaciones);
printf("\nSe realizaron %d intercambios.", intercambios);

printf("\n\nOprima cualquier tecla para continuar\n");
getch();
}
```

El **algoritmo de burbuja** compara cada elemento con el siguiente (0 contra 1; 1 contra 2; etc.) hasta llegar al último. En cada comparación, si el elemento en menor posición es mayor, realiza un intercambio. De esta manera se asegura que al final de este primer recorrido el elemento de mayor valor estará en el último lugar del arreglo. Luego vuelve a explorar todo (excepto el último ítem, que ya fue colocado en su lugar). Esta operación la realiza hasta que solo resta comparar y, en su caso, intercambiar dos elementos.

Al igual que en la selección directa, una variante es localizar el elemento menor e intercambiarlo en el primer lugar. El resultado final y la eficiencia son los mismos. A manera de ejemplo, suponga los siguientes cinco datos:

índice ››	0	1	2	3	4
dato ››	58	38	61	99	16

Durante el primer ciclo compararía 58 contra 38 y los intercambiaría de lugar. El arreglo quedaría así:

índice ››	0	1	2	3	4
dato ››	38	58	61	99	16

Si se continúa con el ciclo, compararía 58 contra 61, sin realizar cambio alguno. Después, 61 contra 99 sin modificar valores. Por último, 99 contra 16; en este caso sí se invertirían los datos. El arreglo después de esta primera iteración es:

índice ››	0	1	2	3	4
dato ››	38	58	61	16	99

Como podrá observarse, en la última posición está el dato de valor mayor. Lo que prosigue es realizar un nuevo ciclo, pero ya sin tocar la posición 4.

El código final se muestra en el programa A.6. La parte central del algoritmo se encuentra en las líneas 20-35. Como observará, se tienen dos ciclos anidados. El ciclo interno —controlado por el contador `j` permite hacer un recorrido para intercambiar datos, hasta que el mayor queda en la última posición. Al finalizar este recorrido, este dato es intercambiado por el último elemento en las líneas 27-31. Este proceso se repetirá varias veces, pero sin tomar en cuenta los datos ya acomodados, lo cual es realizado por el ciclo externo controlado por el contador `i`.

Las variables, comparaciones e intercambios no son parte del algoritmo; miden el número de comparaciones e intercambios realizados para completar el ordenamiento.

```
1  Programa A.6 Ordenamiento por burbuja.
2  48 /* Programa A-6. Ordenamiento de números aleatorios utilizando arreglos
3  y algoritmo de burbuja.*/
4  #include <conio.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <time.h>
8
9  #define LIM 1000
10 int main()
11 {
12 inti, j, num[LIM], aux;
13 int comparaciones, intercambios, huboIntercambios=1;
14
15  //llena el arreglo con números aleatorios
16 srand(time(NULL));
17  for(i=0; i<LIM; i++)
18 num[i] = rand();
19
20  // ordenamiento
21  comparaciones = 0;
22  intercambios = 1;
23  i = LIM - 1;
24 while (i >= 0 &&huboIntercambios> 0) {
25 huboIntercambios = 0;
26 for(j=0; j<i; j++) {
27 if(num[j]>num[j+1]) {
28  aux = num[j];
29 num[j] = num[j+1];
30 num[j+1] = aux;
31 intercambios++;
32 huboIntercambios++;
33  }
34  comparaciones++;
35  }
36  i--;
37  }
38
39  //despliegue de los números
40 for(i=0; i<LIM; i++)
41 printf("\n%d", num[i]);
42
43 //despliegue del número de comparaciones e intercambios
44 printf("\n\nHabía %d datos.", LIM);
45 printf("\nSe realizaron %d comparaciones.", comparaciones);
46 printf("\nSe realizaron %d intercambios.", intercambios);
47 printf("\n\nOprima cualquier tecla para continuar\n");
```

```
48  getch();
49  }
50
```

A.3 Breve comparación de la eficiencia entre selección directa y burbuja

¿Cuál es más eficiente: selección directa o burbuja? De manera intuitiva puede decirse que los dos son de naturaleza cuadrática y se aproximan en el número de comparaciones. También podría suponerse que la burbuja maneja más intercambios, por lo que resulta menos eficiente. Sin embargo, cuando los números están casi ordenados puede ser que termine antes de hacer todos los ciclos, en cuyo caso casi siempre es más eficiente que la selección directa. La tabla A.1 representa una medición de comparaciones e intercambios en una corrida con base en números aleatorios al emplear los códigos citados, la cual puede juzgarse representativa del caso "promedio".

Tabla A.1 Comparación entre algoritmos de selección y burbuja en caso promedio

Algoritmo	Comparaciones			Intercambios		
	10 datos	100 datos	1000 datos	10 datos	100 datos	1000 datos
Selección directa	45	4 950	499 500	9	99	999
Burbuja	45	4 929	497 420	28	2 497	247 063

¿Qué sucede en lo relativo al tiempo físico de las corridas? En definitiva, existen diferencias marcadas según el hardware bajo el cual se ejecute el código, pero en cualquier caso es proporcional al número de comparaciones más el número de intercambios (un intercambio "vale" aproximadamente tres comparaciones). Esta afirmación también podría corroborarse si se pone un contador de tiempo antes y después del núcleo de los algoritmos, ejercicio que deberán resolver los estudiantes.

Como ya se mencionó, otra forma de aproximarse al comportamiento de un algoritmo se deduce de manera matemática la función de trabajo. A manera de ejemplo, ¿cómo podría saberse el número de comparaciones que realiza la selección directa?

Suponga que son 10 datos. En el primer recorrido efectúa nueve comparaciones, posteriormente ocho, después siete… hasta llegar a una. Si generalizamos este criterio, el algoritmo efectúa *n* comparaciones, después n–1, n–2… hasta llegar a 1.

Ahora bien, si la sumatoria de un número es n * (n + 1) / 2, pero el algoritmo comienza de n-1 en lugar de n,en consecuencia, la fórmula resultante es:

$$
\begin{aligned}
f(n) &= n * (n + 1) / 2 - n \\
&= (n^2 + n - 2n) / 2 \\
&= (n^2 - n) / 2
\end{aligned}
$$

A manera de comprobación, si aplicamos la función a 1 000 datos las comparaciones serían 499 500, ¡justo lo que nos arroja la prueba de nuestro código! Podríamos intentar "afinar" alguno de los dos algoritmos, pero la ganancia sería marginal. Para una mejora realmente significativa debe cambiarse el enfoque, como lo hace quicksorto el ordenamiento por árboles binarios.

Algoritmo por quicksort

Es difícil entender porqué el algoritmo de quicksort reduce el tiempo de procesamiento en tal magnitud. La ganancia es en verdad sorprendente: más de 99% cuando el número de datos es muy grande. ¿Cómo lo logra?

Suponga que la función de trabajo de un algoritmo parecido a la selección directa es exactamente $n^2/2$ y se requieren ordenar 12 datos. El número de comparaciones a realizar bajo este hipotético algoritmo sería

$$f(n) = 12^2 / 2$$
$$= 72$$

Ahora, suponga que dividimos la lista en dos sublistas de seis elementos cada una y las ordenamos por separado. El número de comparaciones para ordenar cada lista sería 62/2 = 18. Como son dos sublistas realizamos 36 comparaciones. Ahora bien, ¿cuál sería el costo de mezclarlas? 12, pues se haría un recorrido lineal de cada una. En total 48 comparaciones. Para dejar más o menos clara la situación se recurrirá a la tabla A.2.

Tabla A.2 Ordenamiento basado en dividir la lista original en dos sublistas.

Lista original:

63	84	72	27	76	30	18	12	94	97	69	15

Dos sublistas ordenadas por separado:

27	30	63	72	76	84

12	15	18	69	94	97

Las dos listas, ya mezcladas, se podrían colocar en otro arreglo:

12	15	18	27	30	63	69	72	76	84	94	97

Con el solo hecho de haber creado dos sublistas y ordenarlas por separado, se pasó de 72 comparaciones a 48. Una ganancia de 33%. ¿Qué pasaría si llevamos el asunto más lejos y a su vez partimos las sublistas? Habría otra ganancia de 33%. ¡Descuento sobre descuento!, dirían las tiendas comerciales.

Pero queda un problema por resolver: la gran molestia de copiar las dos listas premezcladas a un arreglo final. Ahí está la creatividad de quicksort: lograr aplicar la idea de dos sublistas sobre el mismo arreglo.

Explicada la idea básica, pasemos a la explicación del algoritmo. La idea central es tomar el primer elemento como "pivote" y "maniobrar" de manera que al final queden en su lugar, todos los elementos menores a él a su izquierda y todos los elementos mayores que él a su derecha. Pueden existir diferentes variantes bajo esta premisa.

Tabla A.3 Primeros pasos del algoritmo quicksort y situación final.

Situación inicial:

	84	72	27	76	30	18	12	94	97	69	15
izq											der

Situación después del primer paso:

15	84	72	27	76	30	18	12	94	97	69	
	izq										der

Situación después del segundo paso:

15		72	27	76	30	18	12	94	97	69	84
	izq									der	

Situación después del tercer paso:

15		72	27	76	30	18	12	94	97	69	84
	izq								der		

Situación final después de pasar una vez por todos los datos:

15	12	18	27	30		76	72	94	97	69	84

Al comenzar se sitúan dos índices a los extremos izquierdo y derecho de la lista, respectivamente. A partir de esta situación inicial se aplica un ciclo que se refleja en la tabla A.3 (se explican los primeros tres pasos para que el estudiante trate de entender el algoritmo de manera inductiva y lo termina de analizar en forma directa en el código que se muestra en el programa A.7).

La primera comparación se hace entre el elemento señalado por izq (63) con el elemento señalado por der (15). Como el 63 es mayor se invierten los datos y se avanza el índice izq. Observe la situación después del primer paso: el 15 ya fue colocado "en su lugar": a la izquierda del 63, que en todo momento es el "pivote". En la segunda comparación, el 63 se compara contra el 84. Como el mayor está en izq., se invierten los elementos y se disminuye el índice der. En la tercera comparación, el 63 se compara contra el 69. Como el elemento señalado por izq. es menor, simplemente se disminuye el índice der.

Cabe destacar que siempre se compararán los elementos señalados por izq. y der.; en ningún momento se mueve el índice que señala al "pivote". El ciclo se termina cuando se juntan los índices izq. y der. En ese instante, todos los elementos menores al "pivote" están a su izquierda y todos los mayores a él, a su derecha. El paso siguiente es llamar a la rutina quicksort de manera recursiva para ordenar las sublistas resultantes.

```
1  Programa A.7 Ordenamiento por quicksort.
2     /* Programa A.7. Ordenamiento de números aleatorios utilizando arreglos
3  y algoritmo quicksort.*/
4  #include <stdio.h>
5  #include <conio.h>
6  #include <time.h>
7  #include <stdlib.h>
8
9  void quicksort (intlimIzq, intlimDer);
10 int comparaciones, intercambios, huboIntercambios;
11
12 // declarar arreglo
13 #define LIM 1000
14 Intdatos[LIM];
15
16 int main() {
17 int i;
18
19  //llena el arreglo con números aleatorios
20 srand(time(NULL));
21  for(i=0; i<LIM; i++)
22 datos[i] = rand();
23
24  //llamada a la rutina de ordenamiento
25  comparaciones = 0;
```

```
 intercambios = 1;
quicksort(0, LIM-1);

 //despliegue de los números
for(i=0; i<LIM; i++)
printf("\n%d", datos[i]);

//despliegue del número de comparaciones e intercambios
printf("\n\nHabía %d datos.", LIM);
printf("\nSe realizaron %d comparaciones.", comparaciones);
printf("\nSe realizaron %d intercambios.", intercambios);

printf("\n\nOprima cualquier tecla para continuar\n");
getch();
}

void quicksort (intlimiteIzq, intlimiteDer) {

intizq = limiteIzq;
int der = limiteDer;

if (izq< der) {
//declaración de variables
int DERECHA = 0, IZQUIERDA = 1, ubicacion = IZQUIERDA;
 //control de dirección de avance
intaux; //variable auxiliar para invertir datos

while (izq< der) {
 comparaciones++; //suma 1 al contador de comparaciones
if (ubicacion == IZQUIERDA) {
if (datos[izq] > datos[der]) {
 intercambios++; //suma 1 al contador de intercambios
aux = datos[izq]; //invertir valores
 datos[izq] = datos[der];
 datos[der] = aux;
izq++; //recorrer pivote izquierdo
ubicacion = DERECHA; //invertir la dirección de avance
 }
else
 der--; //recorrer pivote derecho
 }
else { //se aplica la lógica contraria
if (datos[izq] > datos[der]) {
 intercambios++; //suma 1 al contador de intercambios
aux = datos[izq];
 datos[izq] = datos[der];
 datos[der] = aux;
 der--;
ubicacion = IZQUIERDA;
}
 else
zq++;
 }
 } // while
 quicksort(limiteIzq, izq-1);
 quicksort(izq+1, limiteDer);
 } // if
} // quicksort
```

¿Y qué sucedió con el aumento de eficiencia que se prometió? La tabla A.4 muestra la ganancia en tiempo con respecto a la selección directa. El resultado ya se había anticipado: para 1 000 datos hubo casi 3 000 intercambios más, pero una ganancia de 98% en el número de comparaciones; ganancia que aumentará conforme crezca el volumen de datos.

Tabla A.4 Comparación entre algoritmos de selección directa y quicksort en un caso promedio

Algoritmo	Comparaciones			Intercambios		
	10 datos	100 datos	1000 datos	10 datos	100 datos	1000 datos
Selección directa	45	4 950	499 500	9	99	999
Quicksort	45	566	10 333	14	239	3 830

Como es fácil observar, la eficiencia del software es un tema que puede abordarse desde el primero o segundo cursos de programación. A partir de ahí debe insistirse en él cuando se toquen temas alusivos al contexto: eficiencia de sentencias en base de datos y pruebas de esfuerzo, entre otros.

Mejores y peores casos en los algoritmos

Hasta aquí se ha trabajado con números aleatorios, con la suposición de que los números se hallarán de esa forma o alguna muy parecida. Sin embargo, no siempre es así. Podríamos actuar bajo la premisa de que los números están prácticamente ordenados u ordenados de manera inversa a la deseada. El resultado aparece en la tabla A.5.

Tabla A.5. Comparación entre algoritmos en el "mejor" y "peor" de los casos

	Comparaciones			Intercambios		
	aleatorios	ordenados	orden inverso	aleatorios	ordenados	orden inverso
Selección directa	499 500	499 500	499 500	999	999	999
Burbuja	499 347	999	499 500	250 759	0	499 500
Quicksort	12 352	499 500	499 500	3 628	0	500

Primero lo que ya sabíamos: en el caso de los números aleatorios quicksort es un algoritmo mucho más eficiente que selección directa o burbuja. Ahora destaquemos situaciones que tal vez no eran tan obvias:

1. La forma en que se codificó selección directa hace que se mantenga constante bajo cualquier situación.
2. Si todos los datos están ordenados, el mejor algoritmo resulta ser burbuja, pues desde el primer recorrido percibe esta situación y ya no es necesario pasar a ninguna otra iteración.
3. Quicksort pierde toda su eficiencia con listas casi ordenadas en su totalidad. Esto tiene una explicación relativamente sencilla: la eficiencia de quicksort se basa en dividir la lista en dos sublistas; entre más cercanas a la mitad, mejor. Con datos ordenados el pivote siempre quedará hacia un extremo, con lo cual el algoritmo pierde su sentido (por lo menos, en la versión que programamos).

Suponemos que con este ejercicio fundamentamos una afirmación: debe tenerse cuidado en la elección del algoritmo. El mejor algoritmo en términos de un contexto (por ejemplo, números aleatorios) puede perder su eficiencia en otro contexto (por ejemplo, una lista casi ordenada en su totalidad)[2].

EJERCICIOS SUGERIDOS

- Reproducir los ejercicios y corroborar los resultados. Después de ellos, adaptar los códigos para que se realice un ordenamiento a partir de datos de un archivo; colocar los datos ordenados en otro archivo.
- Llevar a cabo un ordenamiento a través de árboles binarios y analizar su comportamiento. ¿Es más o menos eficiente que quicksort?

A.4 Búsqueda lineal y binaria

Una de las aplicaciones más notorias y didácticas de la eficiencia de los algoritmos se encuentra en la búsqueda. Destacaremos dos tipos de búsquedas, que de alguna manera sirven como punto de partida para todas las demás.

La búsqueda lineal, en la cual es necesario recorrer dato por dato, como los archivos de texto o los viejos casetes. Su orden es `f(n) = t * n`, donde t es el tiempo necesario para buscar cada unidad de medida (explorar un archivo, levantar una encuesta, descargar 1 Mb de internet, etc.). Su naturaleza es lineal.

El código del programa A.8 refleja una búsqueda binaria. Observe que se emplea un único ciclo para buscar el dato de manera secuencial (véanse líneas 25 a 29). Este ciclo finaliza cuando se encuentra el dato o cuando se ha recorrido toda la información (se presupone que no existen datos repetidos).

```
1   Programa A.8 Búsqueda lineal.
2
3   39 /* Programa A.8. Búsqueda lineal sobre un arreglo de datos.*/
4   #include <stdio.h>
5   #include <conio.h>
6
7   intmain() {
8
9   int i; //contador
10  intdatoBuscado; //dato que se tratará de localizar
11  intposicion = -1; //localización del dato. -1 si no se halla
12  int NUMDATOS = 6; //número de datos del arreglo
13  int datos[]= {50, 67, 78, 7, 32, 40}; //arreglo de ejemplo
14
15   //desplegar arreglo original
16  printf("Los datos del arreglo son:\n ");
17  for (i=0; i< NUMDATOS; i++)
18  printf("%d) %d\n ", i, datos[i]);
19  printf("\n");
20
21   //leer el dato
22  printf("Por favor, indique el dato a buscar: ");
23  scanf("%d", &datoBuscado);
```

[2] Nos sentimos obligados a externar una "confesión": en la tabla aparecen cero intercambios en burbuja y quicksort cuando los datos están completamente ordenados, situación por demás lógica. No obstante, la forma en que se codificó hace que en ambos casos se realice un intercambio y una comparación adicional. En la tabla conservamos el resultado esperado para asentar mejor el concepto, pero notificamos de este *warning* para que al estudiante no le extrañe esta situación a la hora de reproducir los ejercicios.

```
 //buscar el dato
i = 0;
 while (posicion == -1 &&i< NUMDATOS)
if (datos[i] == datoBuscado)
posicion = i;
else
 i++;

 //desplegar la posición
if (posicion == -1)
printf("\nNo se encontró el dato.");
else
printf("\n\nEl dato está en la posición %d", posicion);

printf("\n\n\n Oprima cualquier tecla para terminar...");
getch();
{
```

La **búsqueda binaria** parte mitad por mitad los datos y tiene una naturaleza logarítmica. Por ejemplo, para hallar un dato entre un millón, realiza solo 20 intentos. Hay que destacar que para realizar la búsqueda binaria es necesario que los datos estén ordenados antes, por lo cual el ordenamiento se vuelve una situación muy relevante.

Uno de los mejores métodos para acercarse de manera intuitiva a la búsqueda binaria es el juego "Estoy pensando en un número entre1 y1 000. ¿Cuál es?". A cada intento del oponente responde con "menor" o "mayor", según sea el caso. El juego termina cuando se adivina el número. En este caso, es relativamente fácil deducir que se puede resolver al menos en 10 intentos, pues en cada uno de ellos el campo de búsqueda se puede reducir a la mitad. (Si a la primera oportunidad expresamos "estás pensando en el 500" y la otra persona nos contesta "menor", queda claro que el número está entre 1 y 499).

Todo lo anterior suena obvio, pero no lo es cuando se dice "podemos localizar un dato en 20 intentos de entre un millón de datos, siempre y cuando estén ordenados con anterioridad". Lo esencial es que el estudiante lo corrobore.

El programa A.9 muestra un código que ejemplifica la búsqueda binaria con información previamente ordenada dentro de un arreglo. Consiste —al igual que la búsqueda lineal— en un solo ciclo (véanse líneas 25 a 34), dentro del cual se mueven los límites de búsqueda (inicio y final). Concluye cuando se encuentra el elemento buscado o cuando se han "descalificado" todos los elementos.

```
Programa A.9 Búsqueda binaria.
44 /* Programa A.9. Búsqueda binaria sobre un arreglo de datos.*/
#include <stdio.h>
#include <conio.h>

main() {

intdatoBuscado; //dato que se tratará de localizar
intposicion = -1; //localización del dato. -1 si no se halla
int NUMDATOS = 6; //número de datos del arreglo
int datos[]= {7, 32, 40, 50, 67, 78}; //arreglo de ejemplo ordenado
int inicio = 0, final = NUMDATOS-1; //límites de la búsqueda
int mitad; //posición a buscar (mitad de los extremos)

 //desplegar arreglo original
printf("Los datos del arreglo son:\n ");
```

```
for (inti=0; i< NUMDATOS; i++)
printf("%d) %d\n ", i, datos[i]);
printf("\n");

 //leer el dato
printf("Por favor, indique el dato a buscar: ");
scanf("%d", &datoBuscado);

 //buscar el dato
while (posicion == -1 && inicio <= final) {
 mitad = (inicio + final) / 2;
if (datos[mitad] == datoBuscado)
posicion = mitad;
else
if (datos[mitad] >datoBuscado)
 final = mitad - 1;
else
 inicio = mitad + 1;
 }

 //desplegar la posición
if (posicion == -1)
printf("\nNo se encontró el dato.");
else
printf("\n\nEl dato está en la posición %d", posicion);

printf("\n\n\n Oprima cualquier tecla para terminar...");
getch();
}
```

EJERCICIO SUGERIDO

En este caso, además de corroborar los códigos, se sugiere resolver un caso de estudio: ¿cómo logra Google arrojar resultados en un tiempo tan breve?

A.5 Implicaciones prácticas de la función de trabajo

La tabla A.6 sintetiza en gran parte lo visto en este apéndice.[3]

Tabla A.6 Notación O grande para diferentes casos típicos

Notación	Nombre	Ejemplo
O (1)	Constante	Determinar si un número es par o impar.
O (log n)	Logarítmica	Realizar una búsqueda binaria
O (n)	Lineal	Buscar un elemento en una lista desordenada (búsqueda lineal)
O (n log n)	Cuasilineal	Ordenar una lista lo más rápido posible (algoritmo de quicksort o por árboles binarios).
O (n2)	Cuadrática	Ordenar una lista con ordenamiento de inserción, selección directa o burbuja.
O (nc), c > 1	Polinomial	Multiplicación de matrices.
O (cn)	Exponencial	Realizar los movimientos de las torres de Hanoi.

[3] Adaptado de *Enciclopedia de conocimientos fundamentales*. UNAM-Siglo XXI, México, 2010, tomo 5, p. 563.

REFLEXIÓN A.3

La eficiencia: un concepto que debería analizarse en varias materias

Las características del software —y dentro de ellas la eficiencia— debe ser una base conceptual teórica-práctica común a todas las asignaturas de carreras sobre computación e informática, pues son aplicables a diversos contextos de la vida real. No es el único caso: deberían tener el mismo tratamiento todas las características del software, entre ellas la portabilidad, la seguridad y la robustez.

Desglosemos brevemente algunas implicaciones prácticas de la tabla:

- **Pruebas de carga máxima.** Ya que las implicaciones mayores de la función de trabajo se notan al crecer el número de datos, es indispensable que en la labor académica y en el campo laboral se realicen pruebas con el número de datos más grande que soportará el sistema. Incluso, combinar análisis matemáticos con simuladores y pruebas reales. Este es uno de los campos más descuidados en escuelas y empresas.
- **El ambiente de programación.** Mucha de la programación en web se realiza en redes LAN con una muy buena velocidad. Sin embargo, en los ambientes reales muchos sistemas se ejecutarán también en velocidades menores. Cuando un desarrollo hecho en PC o red de área local se migra a su ambiente real, la velocidad cae muchas veces a niveles que vuelven inaplicable el sistema. Merecen especial cuidado los accesos a grandes volúmenes de datos, los diseños que aplican imágenes grandes y la programación visual. Se refleja en mayor medida en el campo de la multimedia y los videojuegos.
- **Consultas a bases de datos.** En la actualidad es raro que se tengan que realizar programas de ordenamiento y búsqueda. Basta con añadir un índice sobre una tabla de la base de datos. Una búsqueda sobre la clave principal es sumamente rápida pues se comporta como búsqueda binaria. Esto se debe a que la clave principal se mantiene ordenada en todo momento. Una búsqueda sobre otro campo dará como resultado una búsqueda lineal, pues este campo no está indexado. Debe tenerse especial cuidado en tablas con un número muy grande de registros.

Suele pensarse que la solución es crear muchos índices secundarios. Pero esto, usado de manera indiscriminada, afecta el tiempo de inserción, borrado y actualización de registros. El justo medio dependerá de la aplicación concreta.

En cualquier caso, una búsqueda como la siguiente está "prohibida" en archivos grandes (si se analiza un poco se notará que hay que buscar registro por registro y, además, letra por letra de cada campo: un proceso de naturaleza cuadrática).

```
SELECT nombrePila FROM personas
WHERE nombrePila LIKE "%JOSE%"
```

Una tabla de 500 000 datos montada en una máquina "vieja" tardó 16 segundos en contestar. Impensable para un sistema web. Este tema casi siempre queda de lado en un curso sobre base de datos. Y es, justamente, el que más afecta a la eficiencia de un sistema de este tipo.

Bibliografía

Aho, Alfred V.; Sethi, Ravi, y Ullman, Jeffrey D. *Compiladores. Principios, técnicas y herramientas*. Addison Wesley Longman, México, 1998, 820 pp.

Bobrowski, Steve. *Oracle 8i para Windows NT, edición de aprendizaje*. McGraw-Hill Osborne, España, 2000.

Booch, Grady; Rumbaugh, James, y Jacobson, Ivan. *El proceso unificado de desarrollo de software*. Pearson, España, 2000.

Bruegge, Bernd y Dutoit, Allen H. *Ingeniería de software orientado a objetos*. Prentice Hall, Pearson, México, 2002.

Bustamante, Paul, *et al*. *Aprenda C++ avanzado como si estuviera en primero*. Escuela Superior de Ingenieros, Campus Tecnológico de la Universidad de Navarra, San Sebastián, 2004.

Deitel, Harvey M. y Deitel, Paul. *Cómo programar en C/C++ y Java*. Pearson, México, 2004, 1113 pp.

Eckel, Bruce. *Piensa en Java*. Pearson–Prentice Hall, segunda edición España, 2002.

Fowler, Martin y Scott, Kendall. *UML Gota a gota*. Pearson, México, 1999.

Frohock García, Marci, *et al*. *Running SQL Server 2000.* McGraw-Hill, España, 2001.

Gracia, Joaquín. *Gestión de proyectos con Scrum*. http://www.ingenierosoftware.com/equipos/scrum.php [consultado el 15 de enero de 2014].

Horstmannn, Cay S. y Cornell, Gary. *Java 2 Fundamentos*. Prentice Hall, España, 2003.

http://books.coreservlets.com/

http://www.bloodshed.net/dev/devcpp.html

http://www.jcreator.com

http://zinjai.sourceforge.net/

http://www.w3api.com/wiki/Java:Math.PI

Keogh, Jim. *J2EE Manual de Referencia.* McGraw-Hill, España, 2003.

Kernighan, Brian W. y Ritchie, Dennos M. *El lenguaje de programación* C. Prentice Hall, México, 1986.

López Goytia, José Luis y César Cruz, Ana Karen. "Los BANOS: un problema no reconocido", *Noti-Upiicsa*, Año 4, No. 6, julio-septiembre 2013.

López Goytia, José Luis y César Cruz, Ana Karen. "Proyecto Megasinapsis: enlacemos el conocimiento", *Noti-Upiicsa*, Año 4, No. 7, octubre-diciembre 2013.

López Goytia, José Luis y Oviedo Galdeano, Mario. *Hacia un replanteamiento de la enseñanza de la programación*. 3er. Congreso Internacional de Educación Media Superior y Superior 2010. Gobierno del Distrito Federal, México, 2010.

McConnell, Steve. *Desarrollo y gestión de proyectos informáticos*. McGraw-Hill, España, 1998, 691 pp.

Meyer, Bertrand. *Construcción de software orientado a objetos.* Prentice Hall, segunda edición, Madrid, 1999, 1198 pp.

Peñaloza, Ernesto. *Fundamentos de programación C/C++*. AlfaOmega, cuarta edición, México, 2004.

Pfaffenberger, Bryan. *Diccionario de términos de computación*. Pearson, México, 1999.

Schach, Stephen R. *Análisis y diseño orientado a objetos con UML y el proceso unificado.* McGraw-Hill, México, 2005.

Schach, Stephen R. *Ingeniería de software clásica y orientada a objetos.* McGraw-Hill, sexta edición, México, 2006.

Schildt, Herbert. *Manual de Bolsillo.* McGraw-Hill, segunda edición, España, 1992.

Tecnológico Nyce. Norma [Mexicana] NMX-I-006-NYCE-2004. México, 2004.

Ullman, Jeffrey D. y Widow, Jennifer. *Introducción a los sistemas de bases de datos*. Pearson, México, 1999.

UNAM. *Enciclopedia de conocimientos fundamentales*. UNAM-Siglo XXI, México, 2010.

Varios autores. *Programación en Java 2.* McGraw-Hill (Schaum), España, 2005.

www.dell.com

www.megasinapsis.com.mx

www.tiobe.com