Agradecimientos

Este libro nunca se hubiera hecho realidad sin el apoyo, consejos y contribuciones realizadas por numerosas personas.

En primer lugar, tengo que dar las gracias con especial atención a José Luis Raya Cabrera, en primer lugar por su paciencia infinita con mi persona, por sus consejos y correcciones realizadas sobre el estilo y los contenidos. Como especialista en redes de comunicaciones y sistemas operativos y con una abultada experiencia en la publicación de manuales y documentación didáctica, su apoyo ha resultado vital en la consecución de esta obra.

Gracias a todo el equipo de la editorial Ra-Ma (*http://www.ra-ma.es*) por el trabajo realizado y por la oportunidad que nos ha brindado para hacer realidad este manual.

Por último, nos gustaría agradecer al lector la confianza depositada en nosotros. Esperamos que los conocimientos adquiridos le sirvan para su desarrollo profesional e intelectual y abran sus puertas a nuevos aprendizajes.

Introducción

Este libro surge con el propósito de acercar al lector a los aspectos más importantes que encierran los sistemas operativos en red, ante la creciente demanda de personal cualificado para su administración. Con tal propósito, puede servir de apoyo también para estudiantes del Ciclo Formativo de Grado Superior de Informática de Informática y las Ingenierías Técnicas.

Hoy en día, existen muchos usuarios y profesionales de la Informática que discuten las ventajas e inconvenientes de algunos sistemas operativos de red y prefieren limitarse al uso exclusivo de uno de ellos. Aquí no hay preferencia por ningún sistema en particular, ni se intenta compararlos para descubrir cuál es el mejor de todos, sino enriquecer los contenidos al exponer sus principales características, manejo y métodos para conseguir la coexistencia entre ellos.

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

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

1 Confección de interfaces de usuario

OBJETIVOS DEL CAPÍTULO

- ✓ Generar interfaces gráficas de usuario mediante editores visuales utilizando la funcionalidad del editor y adaptando el código generado.
- ✓ Crear una interfaz gráfica utilizando los asistentes de un editor visual.
- ✓ Utilizar las funciones del editor para localizar los componentes de la interfaz.
- ✓ Modificar las propiedades de los componentes para adecuar a las necesidades de la aplicación.
- ✓ Analizar y modificar el código generado por el editor visual.
- ✓ Asociar a los eventos las acciones correspondientes.

1.1 LENGUAJES DE PROGRAMACIÓN. TIPOS. PARADIGMAS DE PROGRAMACIÓN

Un lenguaje de programación es un lenguaje formal diseñado para expresar procesos que pueden ser llevados a cabo por máquinas como las computadoras.

Pueden usarse para crear programas que controlen el comportamiento físico y lógico de una máquina, para expresar algoritmos con precisión, o como modo de comunicación humana.

Está formado por un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones. Al proceso por el cual se escribe, se prueba, se depura, se compila (de ser necesario) y se mantiene el código fuente de un programa informático se le llama programación.

Existe un error común que trata por sinónimos los términos 'lenguaje de programación' y 'lenguaje informático'. Los lenguajes informáticos engloban a los lenguajes de programación y a otros más como, por ejemplo, HTML (lenguaje para el marcado de páginas web que no es propiamente un lenguaje de programación, sino un conjunto de instrucciones que permiten estructurar el contenido de los documentos).

Permite especificar de manera precisa sobre qué datos debe operar una computadora, cómo deben ser almacenados o transmitidos y qué acciones debe tomar bajo una variada gama de circunstancias. Todo esto, a través de un lenguaje que intenta estar relativamente próximo al lenguaje humano o natural. Una característica relevante de los lenguajes de programación es precisamente que más de un programador pueda usar un conjunto común de instrucciones que sean comprendidas entre ellos para realizar la construcción de un programa de forma colaborativa.

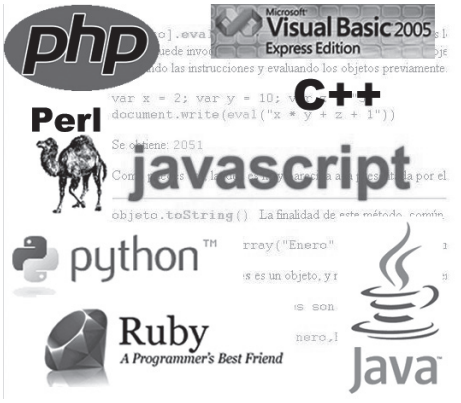

Figura 1.1. *Lenguajes de programación*

1.1.1 TIPOS DE LENGUAJES DE PROGRAMACIÓN

Existen dos tipos de lenguajes claramente diferenciados; los lenguajes de bajo nivel y los de alto nivel.

El ordenador solo entiende un lenguaje conocido como código binario o código máquina, consistente en ceros y unos. Es decir, solo utiliza 0 y 1 para codificar cualquier acción.

Los lenguajes más próximos a la arquitectura hardware se denominan lenguajes de bajo nivel y los que se encuentran más cercanos a los programadores y usuarios se denominan lenguajes de alto nivel.

1.1.1.1 Lenguajes de bajo nivel

Son lenguajes totalmente dependientes de la máquina, es decir, que el programa que se realiza con este tipo de lenguajes no se pueden migrar o utilizar en otras máquinas.

Al estar prácticamente diseñados a medida del hardware, aprovechan al máximo las características del mismo.

Dentro de este grupo se encuentran:

- El **lenguaje máquina**: este lenguaje ordena a la máquina las operaciones fundamentales para su funcionamiento. Consiste en la combinación de ceros y unos para formar las órdenes entendibles por el hardware de la máquina.
  - Este lenguaje es mucho más rápido que los lenguajes de alto nivel.
  - La desventaja es que son bastantes difíciles de manejar y usar, además de tener códigos fuente enormes, donde encontrar un fallo es casi imposible.
- El **lenguaje ensamblador** es un derivado del lenguaje máquina y está formado por abreviaturas de letras y números llamadas nemotécnicos.
  - Con la aparición de este lenguaje se crearon los programas traductores para poder pasar los programas escritos en lenguaje ensamblador a lenguaje máquina. Como ventaja con respecto al código máquina es que los códigos fuente eran más cortos y los programas creados ocupaban menos memoria. Las desventajas de este lenguaje siguen siendo prácticamente las mismas que las del lenguaje ensamblador, añadiendo la dificultad de tener que aprender un nuevo lenguaje difícil de probar y mantener.

1.1.1.2 Lenguajes de alto nivel

Son aquellos que se encuentran más cercanos al lenguaje natural que al lenguaje máquina.

Están dirigidos a solucionar problemas mediante el uso de EDD. EDD es la abreviatura de Estructuras Dinámicas de Datos, algo muy utilizado en todos los lenguajes de programación. Son estructuras que pueden cambiar de tamaño durante la ejecución del programa. Nos permiten crear estructuras de datos que se adapten a las necesidades reales de un programa.

Se tratan de lenguajes independientes de la arquitectura del ordenador. Por lo que, en principio, un programa escrito en un lenguaje de alto nivel, lo puedes migrar de una máquina a otra sin ningún tipo de problema.

Estos lenguajes permiten al programador olvidarse por completo del funcionamiento interno de la/s maquina/s para la/s que está/n diseñando el programa. Tan solo necesitan un traductor que entienda el código fuente como las características de la máquina.

Suelen usar tipos de datos para la programación y hay lenguajes de propósito general (cualquier tipo de aplicación) y de propósito específico (como FORTRAN para trabajos científicos).

1.1.1.3 Generaciones

La evolución de los lenguajes de programación se puede dividir en 5 etapas o generaciones.

- **Primera generación**: lenguaje máquina.
- **Segunda generación**: se crearon los primeros lenguajes ensambladores.
- **Tercera generación**: se crean los primeros lenguajes de alto nivel. Ej.: C, Pascal, Cobol…
- **Cuarta generación**: son los lenguajes capaces de generar código por sí solos. Son los llamados RAD, con lo cuales se pueden realizar aplicaciones sin ser un experto en el lenguaje. Aquí también se encuentran los lenguajes orientados a objetos, haciendo posible la reutilización de partes del código para otros programas. Ej.: Visual Basic, Natural Adabes…
- **Quinta generación**: aquí se encuentran los lenguajes orientados a la inteligencia artificial. Estos lenguajes todavía están poco desarrollados. Ej.: LISP.

Tabla 1.1 Generaciones de los lenguajes de programación

Generación	Nombre	Particularidad
Primera	De maquina	Específico de cada procesador, uso de código binario.
Segunda	Ensamblador	Uso de nemotécnicos que nos abstraen de la máquina.
Tercera	De procedimientos	Lenguajes estructurados con comandos cercanos al lenguaje común.
Cuarta	Orientado a procesos	Programas orientados a problemas específicos.
Quinta	Natural	Incluye inteligencia artificial y sistemas expertos.

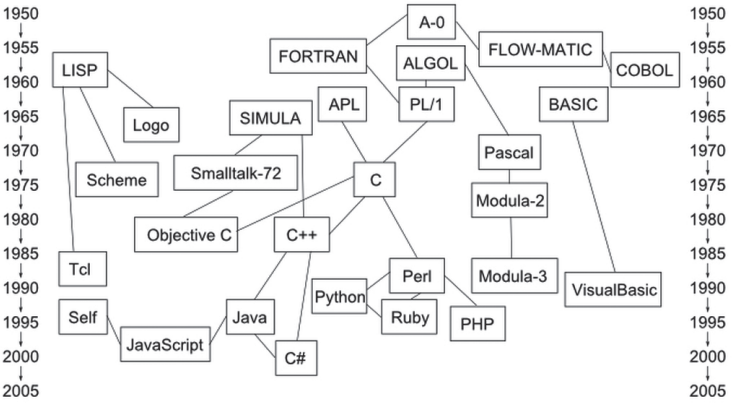

Figura 1.2. *Evolución de los lenguajes de programación*

1.1.2 PARADIGMAS DE PROGRAMACIÓN

Un **paradigma de programación** es una propuesta tecnológica que es adoptada por una comunidad de programadores cuyo núcleo central es incuestionable en cuanto a que únicamente trata de resolver uno o varios problemas claramente delimitados. Es un estilo de programación empleado. La resolución de estos problemas debe suponer consecuentemente un avance significativo en al menos un parámetro que afecte a la ingeniería de software. Tiene una estrecha relación con la formalización de determinados lenguajes en su momento de definición. Un paradigma de programación está delimitado en el tiempo en cuanto a aceptación y uso, ya que nuevos paradigmas aportan nuevas o mejores soluciones que la sustituyen parcial o totalmente.

El paradigma de programación que actualmente es el más usado es la orientación a objetos. El núcleo central de este paradigma es la unión de datos y procesamiento en una entidad llamada "objeto", relacionable a su vez con otras entidades "objeto".

Tradicionalmente, datos y procesamiento se han separado en áreas diferentes del diseño y la implementación de software. Esto provocó que grandes desarrollos tuvieran problemas de fiabilidad, mantenimiento, adaptación a los cambios y escalabilidad. Con la orientación a objetos y características, como el encapsulado, polimorfismo o la herencia, se permitió un avance significativo en el desarrollo de software a cualquier escala de producción.

La orientación a objeto parece estar ligada en sus orígenes con lenguajes como Lisp y Simula, aunque el primero que acuñó el título de programación orientada a objetos fue Smalltalk.

Tipos de paradigmas de programación más comunes:

- **Imperativo o por procedimientos**: es considerado el más común y está representado, por ejemplo, por C, BASIC o Pascal.
- **Funcional**: está representado por Scheme o Haskell. Este es un caso del paradigma declarativo.
- **Lógico**: está representado por Prolog. Este es otro caso del paradigma declarativo.
- **Declarativo**: por ejemplo, la programación funcional, la programación lógica, o la combinación lógico-funcional.
- **Orientado a objetos**: está representado por Smalltalk, un lenguaje completamente orientado a objetos.
- **Programación dinámica**: está definida como el proceso de romper problemas en partes pequeñas para analizarlos.

Si bien puede seleccionarse la forma pura de estos paradigmas al momento de programar, en la práctica es habitual que se mezclen, dando lugar a la programación multiparadigma.

Actualmente, el paradigma de programación más usado es el de la programación orientada a objetos.

1.2 PROGRAMACIÓN ORIENTADA A OBJETOS, PROGRAMACIÓN DIRIGIDA POR EVENTOS Y PROGRAMACIÓN BASADA EN COMPONENTES

1.2.1 PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos o **POO** (OOP según sus siglas en inglés) es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento. Su uso se popularizó a principios de la década de los años 1990. En la actualidad, existe una gran variedad de lenguajes de programación que soportan la orientación a objetos.

Los objetos son entidades que tienen un determinado estado, comportamiento (método) e identidad:

- El estado está compuesto de datos o informaciones; serán uno o varios atributos a los que se habrán asignado unos valores concretos (datos).
- El comportamiento está definido por los métodos o mensajes a los que sabe responder dicho objeto, es decir, qué operaciones se pueden realizar con él.
- La identidad es una propiedad de un objeto que lo diferencia del resto; dicho con otras palabras, es su identificador (concepto análogo al de identificador de una variable o una constante).

Figura 1.3. *Características de la programación orientada a objetos*

Un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos. A su vez, los objetos disponen de mecanismos de interacción llamados métodos, que favorecen la comunicación entre ellos. Esta comunicación favorece a su vez el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separa el estado y el comportamiento.

Los métodos (comportamiento) y atributos (estado) están estrechamente relacionados por la propiedad de conjunto. Esta propiedad destaca que una clase requiere de métodos para poder tratar los atributos con los que cuenta. El programador debe pensar indistintamente en ambos conceptos, sin separar ni darle mayor importancia a alguno de ellos. Hacerlo podría producir el hábito erróneo de crear clases contenedoras de información, por un lado, y clases con métodos que manejen a las primeras, por el otro. De esta manera se estaría realizando una programación estructurada camuflada en un lenguaje de programación orientado a objetos.

La POO difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones y, en segundo lugar, en las estructuras de datos que esos procedimientos manejan. En la programación estructurada solo se escriben funciones que procesan datos. Los programadores que emplean Programación Orientada a Objetos, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos.

1.2.2 PROGRAMACIÓN DIRIGIDA POR EVENTOS

La **programación dirigida por eventos** es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen.

Para entender la programación dirigida por eventos, podemos oponerla a lo que no es: mientras en la programación secuencial (o estructurada) es el programador el que define cuál va a ser el flujo del programa, en la programación dirigida por eventos será el propio usuario (o lo que sea que esté accionando el programa) el que dirija el flujo del programa. Aunque en la programación secuencial puede haber intervención de un agente externo al programa, estas intervenciones ocurrirán cuando el programador lo haya determinado, y no en cualquier momento como puede ser en el caso de la programación dirigida por eventos.

El creador de un programa dirigido por eventos debe definir los eventos que manejarán su programa y las acciones que se realizarán al producirse cada uno de ellos, lo que se conoce como el administrador de evento. Los eventos soportados estarán determinados por el lenguaje de programación utilizado, por el sistema operativo e incluso por eventos creados por el mismo programador.

En la programación dirigida por eventos, al comenzar la ejecución del programa se llevarán a cabo las inicializaciones y demás código inicial y, a continuación, el programa quedará bloqueado hasta que se produzca algún evento. Cuando alguno de los eventos esperados por el programa tenga lugar, el programa pasará a ejecutar el código del correspondiente administrador de evento. Por ejemplo, si el evento consiste en que el usuario ha hecho clic en el botón *Play* de un reproductor de películas, se ejecutará el código del administrador de evento, que será el que haga que la película se muestre por pantalla.

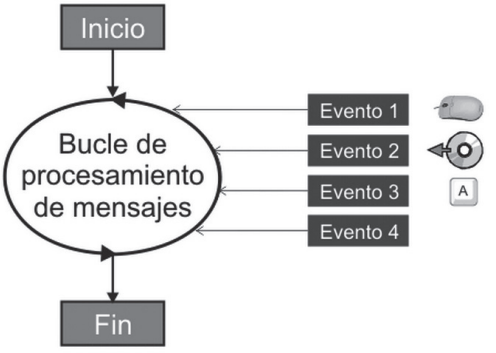

Figura 1.4. *Programación orientada a eventos*

Un ejemplo claro lo tenemos en los sistemas de programación Léxico 3 y Visual Basic, en los que a cada elemento del programa (objetos, controles, etcétera) se le asignan una serie de eventos que generará dicho elemento, como la pulsación de un botón del ratón sobre él o el redibujado del control.

Figura 1.5. *Léxico 3*

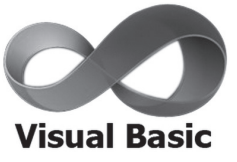

Figura 1.6. *Visual Basic*

La programación dirigida por eventos es la base de lo que llamamos interfaz de usuario, aunque puede emplearse también para desarrollar interfaces entre componentes de Software o módulos del núcleo.

En los primeros tiempos de la computación, los programas eran secuenciales, también llamados *Batch*. Un programa secuencial arranca, lee parámetros de entrada, procesa estos parámetros y produce un resultado, todo de manera lineal y sin intervención del usuario mientras se ejecuta.

Con la aparición y popularización de los PC, el software empezó a ser demandado para usos alejados de los clásicos académicos y empresariales, para los cuales era necesitado hasta entonces, y quedó patente que el paradigma clásico de programación no podía responder a las nuevas necesidades de interacción con el usuario que surgieron a raíz de este hecho.

1.2.3 PROGRAMACIÓN BASADA EN COMPONENTES

La **programación orientada a componentes** (que también es llamada basada en componentes) es una rama de la ingeniería del software, con énfasis en la descomposición de sistemas ya conformados, en componentes funcionales o lógicos, con interfaces bien definidas usadas para la comunicación entre componentes.

Se considera que el nivel de abstracción de los componentes es más alto que el de los objetos y, por tanto, no comparten un estado y se comunican intercambiando mensajes que contienen datos.

Un componente de software es un elemento de un sistema que ofrece un servicio predefinido, y es capaz de comunicarse con otros componentes.

Una definición más simple puede ser: un componente es un objeto escrito de acuerdo a unas especificaciones. No importa qué especificación sea esta, siempre y cuando el objeto se adhiera a la especificación. Solo cumpliendo correctamente con esa especificación es que el objeto se convierte en componente y adquiere características, como reusabilidad.

Cuando se necesita el acceso a un componente, o cuando este debe ser compartido entre distintas redes, se recurre a procesos como la serialización para entregar el componente a su destino.

La capacidad de ser reutilizado (*reusability*), es una característica importante de los componentes de software de alta calidad. Un componente debe ser diseñado e implementado, de tal forma que pueda ser reutilizado en muchos programas diferentes.

Requiere gran esfuerzo y atención escribir un componente que es realmente reutilizable. Para esto, el componente debe estar:

- Completamente documentado.
- Probado intensivamente:
- Diseñado pensando en que será usado de maneras imprevistas.

1.3 HERRAMIENTAS PROPIETARIAS Y LIBRES DE EDICIÓN DE INTERFACES

1.3.1 HERRAMIENTAS LIBRES

Software libre, con acceso a su código, generalmente gratuito:

1.3.1.1 Ventajas

- Existen aplicaciones para todas las plataformas (Linux, Windows, MacOs).
- El precio de las aplicaciones es mucho menor, la mayoría de las veces son gratuitas.
- Libertad de copia.
- Libertad de modificación y mejora.
- Libertad de uso con cualquier fin.
- Libertad de redistribución.
- Facilidad a la hora de traducir una aplicacion en varios idiomas
- Mayor seguridad y fiabilidad.
- El usuario no depende del autor del software

1.3.1.2 Inconvenientes

- Algunas aplicaciones (bajo Linux) pueden llegar a ser algo complicadas de instalar.
- Inexistencia de garantía por parte del autor.
- Interfaces gráficas menos amigables.
- Menor compatibilidad con el hardware.

1.3.2 HERRAMIENTAS PROPIETARIAS

Se trata de software privativo, es decir, no se permite el acceso al código. Hay software que aun siendo gratuito es de este tipo.

1.3.2.1 Ventajas

- Facilidad de adquisición e instalación.
- Existencia de programas diseñados específicamente para desarrollar una tarea.
- Las empresas que desarrollan este tipo de software son, por lo general, grandes y pueden dedicar muchos recursos en el desarrollo e investigación.
- Interfaces gráficas mejor diseñadas.
- Mayor compatibilidad con el hardware.

1.3.2.2 Inconvenientes

- No existen aplicaciones para todas las plataformas, estas versiones generalmente son para Windows y MacOS.
- No pueden ser copiadas a otros equipos (sin infringir la ley).
- No pueden ser modificadas ni adaptadas por el usuario final.
- Restricciones en el uso (marcadas por la licencia).
- Imposibilidad de redistribución.
- Por lo general, suelen ser menos seguras.
- El coste de las aplicaciones es mayor.
- El soporte de la aplicación es exclusivo del propietario.
- El usuario que adquiere software propietario depende al 100% de la empresa propietaria.

1.3.3 IDE

Para desarrollar software por su alta productividad se utilizan entornos de desarrollo o **IDE**. Un entorno de desarrollo (IDE) suele tener los siguientes componentes, aunque no necesariamente todos:

- Un editor de texto.
- Un compilador.
- Un intérprete.
- Un depurador.
- Un cliente.
- Posibilidad de ofrecer un sistema de control de versiones.
- Facilidad para ayuda en la construcción de interfaces gráficas de usuario.

Algunos de los entornos de desarrollo de interfaces gráficas:

1.3.3.1 Microsoft Visual Studio

Es un entorno de desarrollo integrado para sistemas operativos Windows. Soporta varios lenguajes de programación. tales como Visual C++, Visual C#, Visual J# y Visual Basic .NET, al igual que entornos de desarrollo web como ASP. NET. Aunque actualmente se han desarrollado las extensiones necesarias para muchos otros.

Es un producto comercial, aunque Microsoft tiene versiones *Express Edition* gratuitas pero no libres con ciertas limitaciones en la explotación de las aplicaciones a desarrollar.

Adicionalmente, Microsoft ha puesto gratuitamente a disposición de todo el mundo una versión reducida de SQL Server, o Express Edition, cuyas principales limitaciones son que no soporta bases de datos superiores a 4 GB de tamaño. Únicamente se ejecuta en un procesador y emplea 1 GB de RAM como máximo, y no cuenta con el Agente de SQL Server.

Actualmente está en su versión 2013, aunque ya está disponible de Preview de la versión 2015 (*http://www.visualstudio.com/*).

Figura 1.7. *Visual Studio*

ACTIVIDADES 1.1

- Descargue Visual Studio de la web *http://www.visualstudio.com/* e instálelo en su ordenador. Haga capturas de todo el proceso seguido.

1.3.3.2 Gambas

Es un lenguaje de programación libre derivado de BASIC (de ahí que Gambas quiere decir *Gambas Almost Means Basic*). Es distribuido con licencia GNU-GPL. Cabe destacar que presenta ciertas similitudes con Java ya que en la ejecución de cualquier aplicación, se requiere un conjunto de librerías interprete previamente instaladas (*Gambas Runtime*) que entiendan el *bytecode* de las aplicaciones desarrolladas y lo conviertan en código ejecutable por el computador. Por otro lado, es posible desarrollar grandes aplicaciones en poco tiempo.

Permite crear formularios con botones de comandos, cuadros de texto y muchos otros controles y enlazarlos a bases de datos como MySQL, Postgree o SQLite, además de facilitar la creación de aplicaciones muy diversas, como videojuegos (utilizando OpenGL), aplicaciones para dispositivos móviles (en desarrollo pero muy avanzado), aplicaciones de red (con manejo avanzado de protocolos HTTP, FTP, SMTP, DNS), entre otras.

Figura 1.8. *Gambas*

ACTIVIDADES 1.2

- Descargue Gambas de la web *http://gambas.sourceforge.net/es/main.html* e instálelo en su ordenador. Haga capturas de todo el proceso seguido.

1.3.3.3 Glade

Es una herramienta de desarrollo visual de interfaces gráficas mediante GTK/GNOME. Es independiente del lenguaje de programación y no generando código fuente, sino un archivo XML. GtkBuilder es un formato XML que Glade usa para almacenar los elementos de las interfaces diseñadas. Estos archivos pueden emplearse para construirla en tiempo de ejecución mediante el objeto GtkBuilder de GTK+.

Su conexión con lenguajes como Phyton o el IDE Anjuta (compilador C/C++) permite el desarrollo de interfaces en el mundo GNOME. Su sinónimo para KDE es QTCreator.

Figura 1.9. *Glade*

ACTIVIDADES 1.3

- Descargue Glade de la web *https://glade.gnome.org* e instálelo en su ordenador. Haga capturas de todo el proceso seguido.

1.3.3.4 Embarcadero Delphi

Antes conocido como CodeGear Delphi, Inprise Delphi y Borland Delphi, es un entorno de desarrollo de software diseñado para la programación de propósito general con énfasis en la programación visual. En Delphi se utiliza como lenguaje de programación una versión moderna de Pascal llamada Object Pascal.

Figura 1.10. *Embarcadero Delphi*

ACTIVIDADES 1.4

- Descargue Embarcadero Delphi de la web *https://www.embarcadero.com/es/products/delphi* e instálelo en su ordenador. Haga capturas de todo el proceso seguido.

1.3.3.5 NetBeans y Eclipse

Ambos son IDES para JAVA. El primero es de código abierto y admite más de un lenguaje de programación, como PHP y Phyton. El segundo, aunque gratuito, no soporta licencia GNU-GPL.

Figura 1.11. *NetBeans*

Figura 1.12. *Eclipse*

ACTIVIDADES 1.5

- Descargue Netbeans de la web *https://netbeans.org/index_es.html* e instálelo en su ordenador. Haga capturas de todo el proceso seguido.

ACTIVIDADES 1.6

- Descargue Eclipse de la web *https://eclipse.org/ide/* e instálelo en su ordenador. Haga capturas de todo el proceso seguido.

1.3.3.6 Oracle Database

Es un entorno que permite la creación de bases de datos, y con una herramienta Oracle Designer que permite crear las interfaces para acceder a ellas. Tiene una versión gratuita Lite.

Figura 1.13. *Oracle Database*

ACTIVIDADES 1.7

- Descargue Oracle Database de la web *https://www.oracle.com/database/index.html* e instálelo en su ordenador. Haga capturas de todo el proceso seguido.

1.3.3.7 Anjuta

De gran influencia en el mundo Linux. Es un (IDE) para programas en los lenguajes C, C++, Java, Python y Vala, en sistemas GNU/Linux y BSD. Su principal objetivo es trabajar con GTK y en el Gnome. Además, ofrece un gran número de características avanzadas de programación. Es software libre y de código abierto, disponible bajo la Licencia Pública General de GNU.

Figura 1.14. *Anjuta*

ACTIVIDADES 1.8

- Descargue Oracle Database de la web *https://www.oracle.com/database/index.html* e instálelo en su ordenador. Haga capturas de todo el proceso seguido.

1.3.3.8 Dreamweaver

Permite no solo el diseño de interfaces web en HTML, XML... sino el desarrollo de aplicaciones basadas, por ejemplo, en PHP. Soporta ASP.NET, JavaScript, CSS, ColdFusion...

Figura 1.15. *Dreamweaver*

ACTIVIDADES 1.9

- Descargue Dreamweaver de la web *http://www.adobe.com/es/products/dreamweaver.html* e instálelo en su ordenador. Haga capturas de todo el proceso seguido.

1.4 BIBLIOTECAS DE COMPONENTES DISPONIBLES PARA DIFERENTES SISTEMAS OPERATIVOS Y LENGUAJES DE PROGRAMACIÓN. CARACTERÍSTICAS

En informática, una **biblioteca** o **librería** (del inglés **library**) es un conjunto de implementaciones funcionales, codificadas en un lenguaje de programación, que ofrece una interfaz bien definida para la funcionalidad que se invoca.

A diferencia de un programa ejecutable, el comportamiento que implementa una biblioteca no espera ser utilizada de forma autónoma (un programa sí: tiene un punto de entrada principal), sino que su fin es ser utilizada por otros programas, independientes, y de forma simultánea. Por otra parte, el comportamiento de una biblioteca no tiene por qué diferenciarse en demasía del que pudiera especificarse en un programa. Es más, unas bibliotecas pueden requerir de otras para funcionar, pues el comportamiento que definen refina, o altera, el comportamiento de la biblioteca original; o bien la hace disponible para otra tecnología o lenguaje de programación.

Las bibliotecas pueden vincularse a un programa (o a otra biblioteca) en distintos puntos del desarrollo o la ejecución, según el tipo de vínculo que se quiera establecer.

La mayoría de los sistemas operativos modernos proporcionan bibliotecas que implementan los servicios del sistema. De esta manera, estos servicios se han convertido en una "materia prima" que cualquier aplicación moderna espera que el sistema operativo ofrezca. Como tal, la mayor parte del código utilizado por las aplicaciones modernas se ofrece en estas bibliotecas.

1.4.1 TIPOS

1.4.1.1 Las bibliotecas estáticas

Históricamente, las bibliotecas solo podían ser estáticas. Una biblioteca estática, también conocido como "archivo", es un fichero contenedor con varios archivos de código objeto empaquetados, que en el proceso de enlazado durante la compilación serán copiados y relocalizados (si es necesario) en el fichero ejecutable final, junto con el resto de ficheros de código objeto. Este proceso, y el archivo ejecutable, se conocen como una construcción estática de la aplicación objetivo. En este caso, la biblioteca actúa simplemente como un recipiente para ficheros de código objeto que no se diferencian (más que semánticamente) de los ficheros objeto intermedios producidos durante la etapa previa de compilación del programa. En la construcción estática de ficheros compilados se resuelven las direcciones de las subrutinas ensambladas en tiempo de compilación (más específicamente, en la etapa de enlazado), de modo que las referencias a subrutinas de la biblioteca se resuelven estáticamente, del mismo modo que las referencias a cualquier otra función del programa. Así, la dirección real, las referencias para saltos y otras llamadas a rutinas se almacenan en una dirección relativa o simbólica.

El enlazador resuelve todas las direcciones no resueltas convirtiéndolas en direcciones fijas o relocalizables (desde una base común) cargando todo el código (incluyendo las bibliotecas) en posiciones de memoria en tiempo de ejecución. Este proceso de enlazado puede durar incluso más tiempo que el proceso de compilación, y debe ser realizado cada vez que alguno de los módulos es recompilado.

Un enlazador puede trabajar sobre tipos específicos de ficheros objeto y, por lo tanto, requiere tipos específicos (compatibles) de bibliotecas. Los ficheros objeto recompilados en una biblioteca pueden distribuirse y utilizarse fácilmente. Un cliente, ya sea un programa u otra biblioteca, accede a una biblioteca objeto referenciando solo por su nombre. El proceso de enlazado resuelve las referencias buscando en las bibliotecas del orden dado. Por lo general, no se considera un error si un nombre puede encontrarse varias veces en un determinado conjunto de las bibliotecas.

1.4.1.2 Bibliotecas dinámicas

Las bibliotecas dinámicas, vinculadas dinámicamente (o de vínculos dinámicos) son ficheros que contienen código objeto construido de forma independiente a su ubicación, de tal modo que están preparadas para poder ser requeridas y cargadas en tiempo de ejecución por cualquier programa, en lugar de tener que ser enlazadas, previamente, en tiempo de compilación. Por tanto, han de estar disponibles como ficheros independientes al programa ejecutable (generalmente en directorios del sistema). En el proceso de enlazado (en tiempo de compilación) se genera un fichero ejecutable con anotaciones de qué bibliotecas dinámicas requiere (pero no de dónde encontrarlas), y funciones de "esbozo" que se encargan de delegar la llamada a la función al cargador dinámico (o *dynamic-loader*), en Linux *ld.so*. En el resto del programa, las llamadas a las funciones de la biblioteca se cambian por una llamada a la función de esbozo generada por el enlazador.

Por otra parte, cuando la aplicación que se ejecute requiera acceder a las rutinas almacenadas en una biblioteca dinámica, y ejecute la función de esbozo, el cargador de enlaces dinámicos podrá sustituir esta llamada por la función real de la biblioteca dinámica, cargándola en memoria si no lo estuviera ya, y mapeando las páginas de ésta en el espacio de memoria del proceso del programa.

En algunos sistemas operativos puede decidirse si una biblioteca ha de estar disponible inmediatamente o solamente cuando se haga referencia a una función de ella. Si se decide esto último, aparecerá un fenómeno denominado

"retraso de carga", derivado de tener que cargar de memoria secundaria la biblioteca, si no estuviera ya en memoria, y de ajustarla al espacio de direcciones del programa contra el que se vincula.

Las ventajas del enlace dinámico respecto al estático son que se permite la reutilización no solo de código, sino de espacio físico: un mismo fichero de biblioteca compartida puede ser utilizada por varios programas, sin que estos copien su contenido dentro de ellos. Esto puede llegar a ser bastante espacio, según el número de bibliotecas que requiera un programa. Además, puede reutilizarse memoria principal (RAM) para programas que utilicen la misma biblioteca (por ejemplo, puede ser necesario cargar las bibliotecas de Qt únicamente una vez para todos los programas que las utilicen).

Por otra parte, el mayor inconveniente es el aumento del tiempo de carga (debido a tener que buscar el fichero de la biblioteca, cargarlo y relocalizar las llamadas en el programa) y el aumento de una indirección a la hora de llamar a las funciones de la biblioteca.

El enlace dinámico, por su naturaleza, tiene tan solo las limitaciones establecidas por las licencias de software.

La tecnología que permite enlazar bibliotecas de forma dinámica es muy útil para la construcción de *plugins*, sobre todo cuando unas bibliotecas pueden ser sustituidas por otras con una interfaz similar, pero diferente funcionalidad. Se puede decir que un software tiene una "arquitectura de *plugin*" si utiliza bibliotecas con una funcionalidad básica con la intención de que puedan ser sustituidas. Sin embargo, el uso de las bibliotecas enlazadas dinámicamente en la arquitectura de una aplicación no significa necesariamente que puedan ser sustituidas.

El enlace dinámico se desarrolló originalmente en los sistemas operativos Multics a partir de 1964. Se trataba de una característica del MTS (*Michigan Terminal System*), construido a finales de los 60.4.

En distintos sistemas operativos toman distintos nombres, por ejemplo:

- En Microsoft Windows: DLL (*dynamic-link library*).
- En Linux: *shared-objects*.
- En Mac OS: "bibliotecas dinámicas" (*dylibs*).

1.4.2 RELOCALIZACIÓN

Uno de los problemas que el cargador debe gestionar es que la localización real de los datos de la biblioteca no puede conocerse hasta que el ejecutable y todas las bibliotecas dinámicas que se han enlazado han sido cargadas en memoria. Eso es debido a que las localizaciones en memoria dependen de qué bibliotecas dinámicas se han cargado. No es posible depender de la dirección absoluta de los datos en el ejecutable, ni incluso en la biblioteca, ya que podrían producirse conflictos entre las diferentes bibliotecas: si dos de ellas utilizaran las mismas direcciones o sus direcciones se solaparan, sería imposible utilizar ambas en el mismo programa.

Sin embargo, en la práctica, en muchos de los sistemas las bibliotecas no cambian frecuentemente. Por lo tanto, es posible calcular una dirección de carga probable para cada biblioteca compartida en el sistema antes de que sea utilizada, y almacenar esa información en bibliotecas y ejecutables. Si cada biblioteca que es cargada es tratada así, entonces cada una de ellas será cargada en direcciones predeterminadas, lo que acelera el proceso de enlace dinámico. Esta optimización se conoce como *Prebinding* en Mac OS X y *Prelinking* en Linux. Las desventajas de esta técnica son el tiempo requerido de precálculo de las direcciones cada vez que las bibliotecas compartidas cambian, la incapacidad de usar técnicas como la aleatorización de los espacios de direcciones, y el consumo de espacio virtual de direcciones (un problema que queda mitigado por el uso de arquitecturas de 64 bits, al menos en la actualidad).

Un antiguo método era examinar el programa en tiempo de carga. Una vez que todas las bibliotecas fueran cargadas, se reemplazan todas las referencias a datos en las bibliotecas, con punteros a localidades de memoria apropiados. En Windows 3.1 (y algunos sistemas embebidos como las calculadoras Texas Instruments), las referencias eran manejadas como listas ligadas, permitiendo la fácil enumeración y reemplazo. Ahora, la mayoría de las bibliotecas dinámicas ligan una tabla de símbolos con direcciones en blanco dentro del programa en tiempo de compilación. Todas las referencias a código o datos en la biblioteca pasan a través de esta tabla. En tiempo de carga, la tabla es modificada con la dirección de los datos/código por el *linker*. Este proceso es lento y afecta significativamente la velocidad de los programas que llaman continuamente a otros programas, tal como algunos *scripts* de *shell*.

La biblioteca contiene una tabla de saltos de todos los métodos que contiene, denominados puntos de entrada. Las llamadas dentro de la biblioteca "saltan a lo largo" de la tabla, buscando la ubicación del código en memoria y, a continuación, solicitándolo. Estas solicitudes suponen un sobreesfuerzo, pero el retardo es habitualmente tan pequeño que es despreciable.

1.4.2.1 Localización de bibliotecas en tiempo de ejecución

Los enlazadores/cargadores dinámicos tienen una funcionalidad muy amplia. Algunos dependen de rutas explícitas a las bibliotecas almacenadas en los ejecutables. Cualquier cambio en la nomenclatura o el diseño del sistema de ficheros hará que estos sistemas fallen. Habitualmente solo se almacena en el ejecutable el nombre de la biblioteca (no la ruta), siendo el sistema operativo el que proporciona el mecanismo para encontrar la biblioteca en el disco mediante ciertos algoritmos.

Una de las mayores desventajas del enlace dinámico es que el funcionamiento correcto de los ejecutables depende de una serie de bibliotecas almacenadas de forma aislada. Si la biblioteca es borrada, movida o renombrada, o si una versión incompatible de DLL es copiada en una ubicación que aparece antes en la ruta de búsqueda, el ejecutable no se podrá cargar. En Windows esto se conoce como *DLL hell* (en español "Infierno de las DLL").

1.4.2.2 Sistemas Unix

La mayor parte de los sistemas tipo Unix disponen de una "ruta de búsqueda" que especifica los directorios del sistema de archivos en los que buscar las bibliotecas dinámicas. En algunos sistemas, la ruta por defecto es especificada en un archivo de configuración; en otros, está prefijada (*hard coded*) en el cargador dinámico. Algunos formatos de fichero ejecutable pueden especificar directorios adicionales en los que buscar las bibliotecas de un determinado programa. Esto puede ser usualmente alterado por una variable de entorno, aunque es deshabilitado para programas que tengan *setuid* o *setgid*, de manera que el usuario no puede forzar a ese programa a ejecutar un código arbitrario. Es aconsejable que los desarrolladores de bibliotecas pongan sus bibliotecas dinámicas en directorios que se encuentren en la ruta de búsqueda por defecto. Por el contrario, esto puede hacer problemática la instalación de nuevas bibliotecas, pues hace que esos directorios crezcan mucho haciéndose complicada su gestión.

1.4.3 BIBLIOTECAS REMOTAS

Otra solución al problema de las bibliotecas es usar ejecutables completamente separados (a menudo una versión ligera) y llamarlos usando llamadas a procedimiento remoto sobre la red a otra computadora conocido como *remote procedure call* o RPC. Este enfoque maximiza la reutilización del sistema operativo: el código necesario para dar

soporte a la biblioteca es el mismo que el usado para proveer a la aplicación soporte y seguridad para cualquier otro programa. Adicionalmente, dichos sistemas no requerirán que la biblioteca esté grabada en la misma máquina, pudiendo redireccionar la petición por la red.

Sin embargo, tal enfoque implica que cada llamada a la biblioteca requerirá una gran cantidad de *overhead*. Las llamadas RPC son mucho más costosas que llamadas a procedimiento en la propia máquina. Este enfoque se usa comúnmente en las arquitecturas distribuidas que hacen un uso intensivo de las RPC, en los sistemas cliente-servidor y en aplicaciones como Enterprise JavaBeans.

1.4.4 BIBLIOTECA ESTÁNDAR DE C

La biblioteca estándar de C (también conocida como *libc*) es una recopilación de ficheros cabecera y bibliotecas con rutinas, estandarizadas por un comité de la Organización Internacional para la Estandarización (ISO), que implementan operaciones comunes, tales como las de entrada y salida o el manejo de cadenas. A diferencia de otros lenguajes como COBOL, Fortran, o PL/1, C no incluye palabras clave para estas tareas, por lo que prácticamente todo programa implementado en C se basa en la biblioteca estándar para funcionar.

El nombre y las características de cada función, el prototipo, así como la definición de algunos tipos de datos y macros, se encuentran en un fichero denominado "archivo de cabecera" (con extensión ".h"), pero la implementación real de las funciones están separadas en un archivo de la biblioteca. La denominación y el ámbito de las cabeceras se han convertido en comunes, pero la organización de las bibliotecas sigue siendo diversa, ya que estas suelen distribuirse con cada compilador. Dado que los compiladores de C a menudo ofrecen funcionalidades adicionales que no están especificadas en el ANSI C, la biblioteca de un compilador no siempre es compatible con el estándar ni con las bibliotecas de otros compiladores.

La biblioteca estándar de ANSI C consta de 24 ficheros cabecera que pueden ser incluidos en un proyecto de programación con una simple directiva. Cada cabecera contiene la declaración de una o más funciones, tipos de datos y macros.

En comparación con otros lenguajes de programación (como, por ejemplo, Java) la biblioteca estándar es muy pequeña, esta proporciona un conjunto básico de funciones matemáticas, de tratamiento de cadenas, conversiones de tipo y entrada/salida por consola o por ficheros. No se incluyen, ni un conjunto de tipos de datos contenedores básicos (listas, pilas, colas...), ni herramientas para crear una interfaz gráfica de usuario (GUI), ni operaciones para trabajar en red, ni otras funcionalidades que lenguajes como C++ o Java incorporan de manera estándar. La principal ventaja del reducido tamaño de la biblioteca estándar de C es que construir un entorno de trabajo en ANSI C es muy fácil y, en consecuencia, portar un programa en ANSI C de una plataforma a otra es relativamente sencillo.

Se han desarrollado muchas otras bibliotecas para proporcionar una funcionalidad equivalente a la de otros lenguajes de programación. Por ejemplo, el proyecto de desarrollo del entorno de escritorio de GNOME creó las bibliotecas *GTK+* y *GLib* con funcionalidades para desarrollar y trabajar con interfaces gráficas de usuario. La variedad de bibliotecas disponibles ha hecho que, a lo largo de la historia, haya quedado demostrada la superioridad de algunas de estas herramientas. El gran inconveniente es que a menudo no funcionan especialmente bien en conjunto, normalmente son los propios programadores familiarizados con las diferentes bibliotecas quienes consiguen sacarles el máximo partido, aunque diferentes partes de ellas puedan estar disponibles en cualquier plataforma.

Tabla 1.2

Librería	Contenido
<assert.h>	Contiene la macro *assert* (aserción), utilizada para detectar errores lógicos y otros tipos de fallos en la depuración de un programa.
<complex.h>	Conjunto de funciones para manipular números complejos (nuevo en C99).
<ctype.h>	Contiene funciones para clasificar caracteres según sus tipos o para convertir entre mayúsculas y minúsculas, independientemente del conjunto de caracteres (típicamente ASCII o alguna de sus extensiones).
<errno.h>	Para analizar los códigos de error devueltos por las funciones de biblioteca.
<fenv.h>	Para controlar entornos en coma flotante (nuevo en C99).
<float.h>	Contiene la definición de constantes que especifican ciertas propiedades de la biblioteca de coma flotante, como la diferencia mínima entre dos números en coma flotante (_EPSOLON), el número máximo de dígitos de precisión (_DIG), o el rango de valores que se pueden representar (_MIN, _MAX).
<inttypes.h>	Para operaciones de conversión con precisión entre tipos enteros (nuevo en C99).
<iso646.h>	Para utilizar los conjuntos de caracteres ISO 646 (nuevo en NA1).
<limits.h>	Contiene la definición de constantes que especifican ciertas propiedades de los tipos enteros, como rango de valores que se pueden representar (_MIN, _MAX).
<locale.h>	Para la función *setlocale()* y las constantes relacionadas. Se utiliza para seleccionar el entorno local apropiado (configuración regional).
<math.h>	Contiene las funciones matemáticas comunes.
<setjmp.h>	Declara las macros *setjmp* y *longjmp* para proporcionar saltos de flujo de control de programa no locales.
<signal.h>	Para controlar algunas situaciones excepcionales como la división por cero.
<stdarg.h>	Posibilita el acceso a una cantidad variable de argumentos pasados a una función.
<stdbool.h>	Para el tipo *booleano* (nuevo en C99).
<stdint.h>	Para definir varios tipos enteros (nuevo en C99).
<stddef.h>	Para definir varios tipos de macros de utilidad.
<stdio.h>	Proporciona el núcleo de las capacidades de entrada/salida del lenguaje C.
<stdlib.h>	Para realizar ciertas operaciones como conversión de tipos, generación de números pseudo-aleatorios, gestión de memoria dinámica, control de procesos, funciones de entorno, de ordenación y búsqueda.
<string.h>	Para manipulación de cadenas de caracteres.
<tgmath.h>	Contiene funcionalidades matemáticas de tipo genérico (*type-generic*) (nuevo en C99).
<time.h>	Para tratamiento y conversión entre formatos de fecha y hora.

<wchar.h>	Para manipular flujos de datos anchos y varias clases de cadenas de caracteres anchos (2 o más bytes por carácter), necesario para soportar caracteres de diferentes idiomas (nuevo en NA1).
<wctype.h>	Para clasificar caracteres anchos (nuevo en NA1).

- Las cabeceras <iso646.h>, <wchar.h> y <wctype.h> fueron añadidas con Normativa Addendum 1 (abreviado NA1), y fueron ratificadas por el estándar en 1995.
- Las cabeceras <complex.h>, <fenv.h>, <inttypes.h>, <stdbool.h>, <stdint.h> y <tgmath.h> fueron añadidas en 1999 con la revisión C99 del estándar.

1.4.5 BIBLIOTECA ESTÁNDAR DE C++

En C++, la biblioteca estándar es una colección de Clases y funciones, escritas en el núcleo del lenguaje. La biblioteca estándar proporciona varios contenedores genéricos, funciones para utilizar y manipular esos contenedores, funciones objeto, cadenas y flujos genéricos (incluyendo E/S interactiva y de archivos) y soporte para la mayoría de las características del lenguaje. La biblioteca estándar de C++ también incorpora la ISO C90 biblioteca estándar de C. Las características de la biblioteca estándar están declaradas en el espacio de nombres (*namespace*) *std*.

La Standard Template Library es un subconjunto de la biblioteca estándar de C++ que contiene los contenedores, algoritmos, iteradores, funciones objeto, etc; aunque algunas personas utilizan el término STL indistintamente con la biblioteca estándar de C++.

Los siguientes archivos contienen las declaraciones de la biblioteca estándar.

1.4.5.1 Contenedores

Tabla 1.3

Librería	Contenido
<bitset>	Provee la clase contenedora especializada *std::bitset*, un *array* de bits.
<deque>	Provee la plantilla clase contenedora *std::deque*, una cola doblemente enlazada.
<list>	Provee la plantilla clase contenedora *std::list*, una lista doblemente enlazada.
<map>	Provee las plantillas clases contenedoras *std::map* y *std::multimap*, un arreglo asociativo y un arreglo asociativo múltiple respectivamente.
<queue>	Provee la clase adaptadora contenedora *std::queue*, una cola de datos.
<set>	Provee las plantillas clases contenedoras *std::set* y *std::multiset*, contenedores asociativos ordenados.
<stack>	Provee la clase adaptadora contenedora *std::stack*, una pila de datos.
<vector>	Provee la plantilla clase contenedora *std::vector*, un arreglo dinámico.

1.4.5.2 General

Tabla 1.4

Librería	Contenido
`<algorithm>`	Provee la definición de muchos algoritmos.
`<functional>`	Provee varios objetos funcionales, diseñados para ser utilizados por los algoritmos estándares.
`<iterator>`	Provee clases y plantillas para trabajar con iteradores.
`<locale>`	Provee las clases y plantillas para trabajar con locales.
`<memory>`	Provee facilidades para el manejo de memoria en C++, incluyendo la clase plantilla *std::auto_ptr*.
`<stdexcept>`	Contiene las clases estándares de excepción, así como *std::logic_error* y *std::runtime_error*, ambas derivadas de *std::exception*.
`<utility>`	Provee la clase plantilla *std::pair*, para trabajar con pares (tuplas de dos miembros) de objetos.

1.4.5.3 Strings

Tabla1.5

Librería	Contenido
`<string>`	Provee las clases y plantillas estándares de C++ para trabajar con cadena de caracteres.

Flujos y Entrada/Salida

Tabla 1.6

Librería	Contenido
`<fstream>`	Provee facilidades para la entrada y salida basada en archivos.
`<ios>`	Provee varios tipos y funciones básicas para la operación de flujos de entrada y salida.
`<iostream>`	Provee los elementos fundamentales para la entrada y salida en C++.
`<iosfwd>`	Provee las declaraciones hacia adelante de varias clases plantillas de E/S.
`<iomanip>`	Provee facilidades para manipular el formateo de salida, así como la base utilizada cuando se formatean enteros y los valores "precisión" o "punto flotante".

<istream>	Provee la clase plantilla *std::istream* y otras clases para la entrada de datos.
<ostream>	Provee la clase plantilla *std::ostream* y otras clases para la salida de datos.
<sstream>	Provee la clase plantilla *std::sstream* y otras clases para la manipulación de cadena de caracteres.

1.4.5.4 Numéricas

Tabla 1.7

Librería	Contenido
<complex>	Provee la clase plantilla *std::complex* y funciones asociadas al trabajo con números complejos.
<numeric>	Provee algoritmos para el procesamiento numérico.
<valarray>	Provee la clase plantilla *std::valarray*, una clase arreglo optimizada para el procesamiento numérico.

1.4.5.5 Soporte del Lenguaje

Tabla 1.8

Librería	Contenido
<exception>	Provee varios tipos y funciones relacionadas al manejo de excepciones, incluyendo *std::exception*, la clase base para todas las excepciones arrojadas por la biblioteca estándar.
<limits>	Provee la clase plantilla *std::numeric_limits*, utilizada para describir las propiedades de los tipos numéricos fundamentales.
<new>	Provee los operadores *new* y *delete* y otras funciones y tipos que componen los elementos fundamentales para el manejo de memoria en C++.
<typeinfo>	Provee las facilidades para trabajar con información de tipo en tiempo de ejecución en C++.

1.4.6 BIBLIOTECA ESTÁNDAR DE JAVA

Las librerías estándar de Java incorporan un gran número de *packages* que facilitan la programación en muchas áreas distintas. Así, tenemos, por ejemplo, clases para programación en red (*java.net*), entrada/salida (*java.io*), estructuras de datos como vectores y colecciones (*java.util*), programación de *applets* (java.applet) o de interfaces gráficos (*java.awt*).

Aquí veremos solo algunas de las librerías más básicas. Para el resto, lo mejor es consultar la documentación sobre el API de Java que se distribuye junto con el SDK.

1.4.6.1 Operaciones matemáticas: java.lang.Math

La clase *java.lang.Math* encapsula las constantes y operaciones matemáticas más comunes. Todos los miembros y métodos de esta clase son de tipo *static*. Por ello, no es necesario instanciar ningún objeto de la clase *Math*, sino que los métodos se usan directamente con el nombre de la clase. Así, por ejemplo, podemos llamar en nuestro código a *Math.cos*(angulo) o a *Math.abs*(valor). Además, tenemos constantes predefinidas como *Math.PI* o *Math.E*.

1.4.6.2 Operaciones de entrada/salida: java.io

La jerarquía de clases definidas en *java.io* para realizar operaciones de entrada/salida es bastante amplia. En este apartado únicamente comentaremos las clases básicas. La referencia completa la proporciona, por supuesto, la documentación que incluye el Java SDK.

1.4.6.3 E/S de bytes y caracteres

Según si la operación está orientada a bytes o caracteres, tenemos dos jerarquías de clases distintas. Todas las clases que se ocupan de E/S de bytes reciben un nombre del tipo *xxxInputStream* (las de entrada) o *xxxOutputStream* (salida). Las que hacen lo propio con caracteres son las *xxxReader* y *xxxWriter*, respectivamente. Así, por ejemplo, la clase que sirve para leer bytes de un *array* es *ByteArrayInputStream*, y la clase empleada para escribir caracteres en un fichero, *FileWriter*.

Las clases que realizan operaciones de entrada (tanto de bytes como de caracteres) comparten el método *read*. Versiones sobrecargadas de read sirven para leer secuencias de bytes o caracteres y almacenarlos en un *array*. Lo mismo ocurre con las clases que implementan operaciones de salida y el método *write*.

Así, el siguiente fragmento de código serviría para copiar los contenidos de un fichero en otro:

```
FileReader in = new FileReader("datos.txt");
FileWriter out = new FileWriter("copia.txt");
int c;
while ((c = in.read()) != -1)
out.write(c);
in.close();
out.close();
```

En el código anterior, se utilizan las clases *FileReader* y *FileWriter* que, como es de suponer, implementan lectura y escritura de caracteres en un fichero, respectivamente.

1.4.6.4 E/S de más alto nivel

En el paquete *java.io* se definen una serie de clases que incorporan funcionalidades más complejas que la simple e/s de bytes o caracteres. Así, por ejemplo hay clases que incorporan un *buffer* para hacer más eficientes las operaciones, u otras que realizan lectura de tipos de datos como *int* o *double*.

A la hora de utilizar estas clases, la idea es instanciar primero un objeto que nos dé la funcionalidad básica (por ejemplo, un *FileInputStream*, lectura de bytes de un fichero) y utilizar este como base para instanciar la clase compleja. Por ejemplo:

```
FileInputStream f = new FileInputStream("datos");
DataInputStream d = new DataInputStream(f);
d.readInt();
d.readDouble();
d.readChar();
```

Como puede observarse, el constructor de la clase *DataInputStream* acepta como parámetro un objeto de la clase básica *FileInputStream*. Además, *DataInputStream* incorpora una serie de métodos para leer datos de tipo *int*, *double* o *char*, entre otros.

1.4.6.5 Entrada y salida estándar

La entrada y salida estándar en Java están representadas por los objetos *System.in* y *System.out*, respectivamente. El primero es un objeto de la clase *inputStream*. La salida estándar es un objeto de la clase *PrintStream*, que define los métodos *print* y *println* y que sirven para imprimir cualquier tipo de dato (*println* imprime además un retorno de carro final).

1.4.6.6 AWT (Abstract Window Toolkit)

Permite desarrollar interfaces de usuario gráficas. Es la librería básica aunque en la actualidad está siendo reemplazada por *Swing*.

Está compuesta por:

- Los Componentes (*java.awt.component*), como los *Buttons*, *Labels*...
- Los Contenedores (*java,awt.containers*), contienen componentes.
- Los gestores de posición (*java.awt.LayoutManager*), que posiciona los componentes dentro de los contenedores.
- Los eventos (*java.awt.AWTEvent*), que nos indican las acciones del usuario.

1.4.6.7 Javax.swing

Proporciona una serie de clases e interfaces que amplían la funcionalidad del anterior. Es la versión más moderna del API de Java. Están escritos en Java y son independientes de la plataforma.

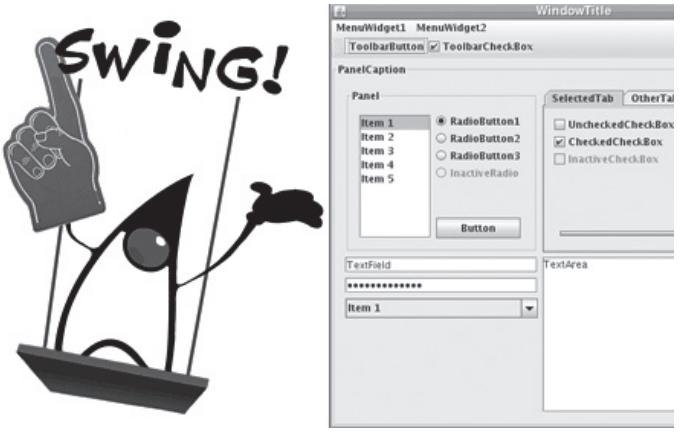

Figura 1.16. *Java Swing*

1.4.7 OPENGL (OPEN GRAPHICS LIBRARY)

Es una especificación estándar que define una API (*Application Programming Interface*, Interfaz de Programación de Aplicaciones) multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos. Fue desarrollada originalmente por Silicon Graphics Inc. (SGI) y se usa ampliamente en CAD, realidad virtual, representación científica, visualización de información simulación de vuelos y desarrollo de videojuegos, donde compite con Direct3D en plataformas Microsoft Windows.

El funcionamiento básico de OpenGL consiste en aceptar primitivas tales como puntos, líneas y polígonos, y convertirlas en píxeles. Este proceso es realizado por una *pipeline* gráfica conocida como "Máquina de estados de OpenGL". La mayor parte de los comandos de OpenGL bien emiten primitivas a la *pipeline* gráfica o bien configuran cómo la *pipeline* procesa dichas primitivas.

Figura 1.17. *OpenGL*

Una descripción somera del proceso en la *pipeline* gráfica podría ser:

- Evaluación, si procede, de las funciones polinomiales que definen ciertas entradas, como las superficies NURBS, aproximando curvas y la geometría de la superficie.
- Operaciones por vértices, transformándolos e iluminándolos según su material y recortando partes no visibles de la escena para producir un volumen de visión.
- Rasterización, o conversión de la información previa en píxeles. Los polígonos son representados con el color adecuado mediante algoritmos de interpolación.

- Operaciones por fragmentos o segmentos, como actualizaciones según valores venideros o ya almacenados de profundidad y de combinaciones de colores, entre otros.
- Por último, los fragmentos son volcados en el *Frame buffer* (dispositivo gráfico que representa los píxeles de la pantalla como ubicaciones en la memoria de acceso aleatorio o RAM gráfica).

1.4.8 DIRECTX

DirectX es una colección de API desarrolladas para facilitar las complejas tareas relacionadas con multimedia, especialmente programación de juegos y vídeo, en la plataforma Microsoft Windows.

Figura 1.18. *DirectX*

DirectX consta de las siguientes API:

- **Direct3D**: utilizado para el procesamiento y la programación de gráficos en tres dimensiones (una de las características más usadas de DirectX).
- **Direct Graphics**: para dibujar imágenes en dos dimensiones (planas), y para representar imágenes en tres dimensiones.
- **DirectInput**: para procesar datos del teclado, mouse, joystick y otros controles para juegos.
- **DirectPlay**: para comunicaciones en red.
- **DirectSound**: para la reproducción y grabación de sonidos de ondas.
- **DirectMusic**: para la reproducción de pistas musicales compuestas con DirectMusic Producer.
- **DirectShow**: para reproducir audio y vídeo con transparencia de red.
- **DirectSetup**: para la instalación de componentes DirectX.
- **DirectCompute**: lenguaje e instrucciones especiales para el manejo de cientos o miles de hilos de procesamiento, especial para procesadores de núcleos masivos.

1.4.9 GTK "GIMP TOOL KIT"

Es una biblioteca del equipo GTK+, la cual contiene los objetos y funciones para crear la interfaz gráfica de usuario. Maneja widgets como ventanas, botones, menús, etiquetas, deslizadores, pestañas, etc. GNOME utiliza estas librerías.

GTK+ se ha diseñado para permitir programar con lenguajes como C, C++, C#, Java, Ruby, Perl, PHP o Python.

Figura 1.19. *GTK*

GTK+ se basa en varias bibliotecas desarrolladas por el equipo de GTK+ y de GNOME:

- **GLib**. Biblioteca de bajo nivel, estructura básica de GTK+ y GNOME. Proporciona manejo de estructura de datos para C, portabilidad, interfaces para funcionalidades de tiempo de ejecución, como ciclos, hilos, carga dinámica o un sistema de objetos.
- **GTK**. Biblioteca la cual realmente contiene los objetos y funciones para crear la interfaz de usuario. Maneja *widgets* como ventanas, botones, menús, etiquetas, deslizadores, pestañas, etc.
- **GDK**. Biblioteca que actúa como intermediario entre gráficos de bajo nivel y gráficos de alto nivel.
- **ATK**. Biblioteca para crear interfaces con características de una gran accesibilidad muy importante para personas discapacitadas o minusválidas. Pueden usarse utilerías como lupas de aumento, lectores de pantalla, o entradas de datos alternativas al clásico teclado o ratón.
- **Pango**. Biblioteca para el diseño y renderizado de texto, hace hincapié especialmente en la internacionalización. Es el núcleo para manejar las fuentes y el texto de GTK+2.

1.4.10 QT

Es una biblioteca multiplataforma usada para desarrollar aplicaciones con interfaz gráfica de usuario, así como también para el desarrollo de programas sin interfaz gráfica, como herramientas para la línea de comandos y consolas para servidores.

Qt es utilizada en KDE, entorno de escritorio para sistemas como GNU/Linux o FreeBSD, entre otros. Qt utiliza el lenguaje de programación C++ de forma nativa, adicionalmente puede ser utilizado en otros lenguajes de programación a través de *bindings*. También es usada en sistemas informáticos empotrados para automoción, aeronavegación y aparatos domésticos, como frigoríficos.

***Figura 1.20.** QT*

Funciona en todas las principales plataformas, y tiene un amplio apoyo. El API de la biblioteca cuenta con métodos para acceder a bases de datos mediante SQL, así como uso de XML, gestión de hilos, soporte de red, una API multiplataforma unificada para la manipulación de archivos y una multitud de otros para el manejo de ficheros, además de estructuras de datos tradicionales.

1.5 ÁREA DE DISEÑO, PALETA DE COMPONENTES, EDITOR DE PROPIEDADES

Un entorno de desarrollo integrado, llamado también **IDE** (sigla en inglés de *Integrated Development Environment*), es un programa informático compuesto por un conjunto de herramientas de programación. Puede dedicarse en exclusiva a un solo lenguaje de programación o bien puede utilizarse para varios.

Un IDE es un entorno de programación que ha sido empaquetado como un programa de aplicación; es decir, que consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (GUI). Los IDE pueden ser aplicaciones por sí solas o pueden ser parte de aplicaciones existentes.

El entorno de desarrollo se divide, generalmente, en varias partes.

1.5.1 MENÚ PRINCIPAL, BARRA DE HERRAMIENTAS Y PALETA DE COMPONENTES

En la parte superior se coloca la ventana principal, que suele contener el menú principal, la barra de herramientas y, en ocasiones, la paleta de componentes.

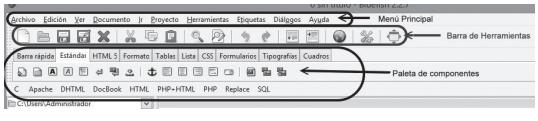

Figura 1.21. *Parte superior del IDE*

Hay IDE en los que la paleta de componentes aparece en el lado izquierdo en lugar de aparecer en la parte superior.

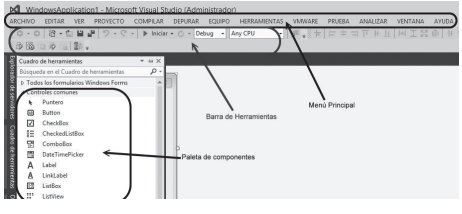

Figura 1.22. *Parte superior (y lateral) del IDE*

1.5.2 PROPIEDADES DE LOS OBJETOS. EXPLORADOR DE ARCHIVOS

Para cambiar las propiedades de los objetos que forman la aplicación y seleccionar los eventos a los que debe responder la aplicación. Esta sección suele estar en uno de los laterales. Si la paleta se encuentra en el lado izquierdo, normalmente estará en el lado derecho. En ocasiones, nos podemos encontrar un explorador de archivos para que la navegación por los diferentes ficheros del proyecto se produzca de forma sencilla y natural.

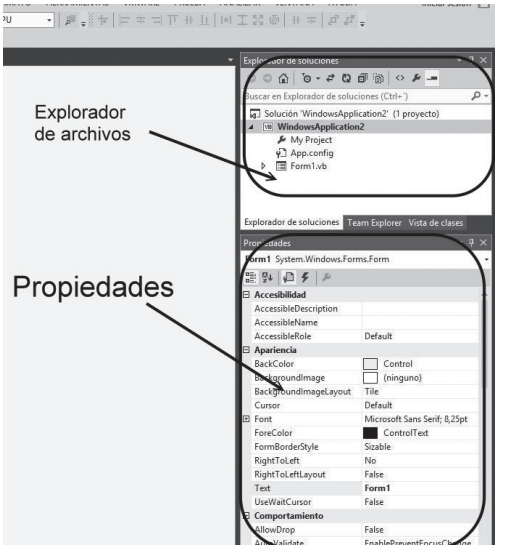

Figura 1.23. *Propiedades y explorador en Visual Studio*

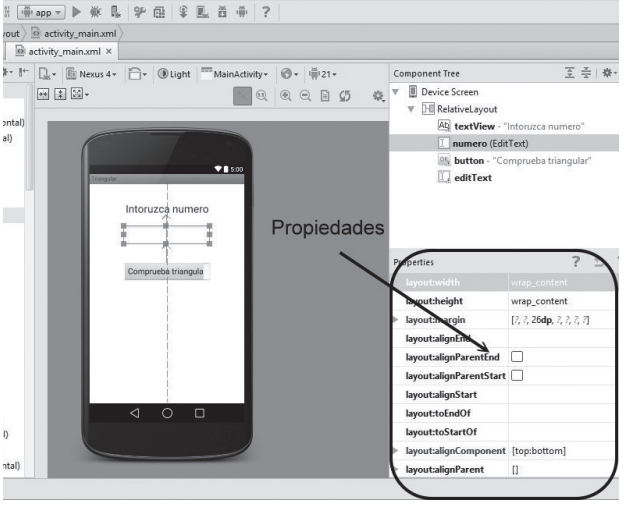

Figura 1.24. *Propiedades en Android Studio*

1.5.3 DISEÑADOR DE FORMULARIOS

Es una ventana cuadriculada (o no) sobre el que se disponen los componentes para diseñar las ventanas que formarán la aplicación.

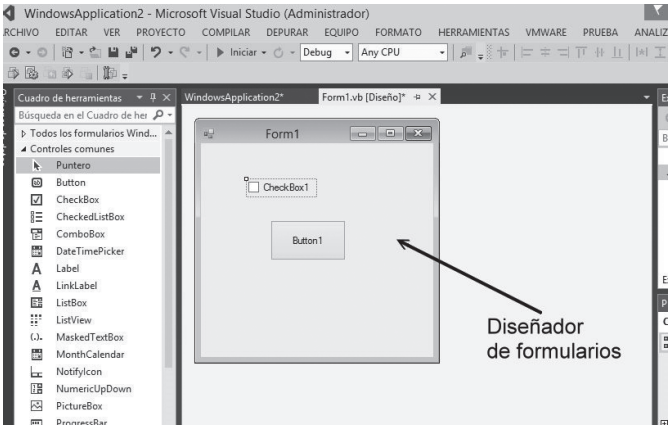

Figura 1.25. *Diseñador de formularios Visual Studio*

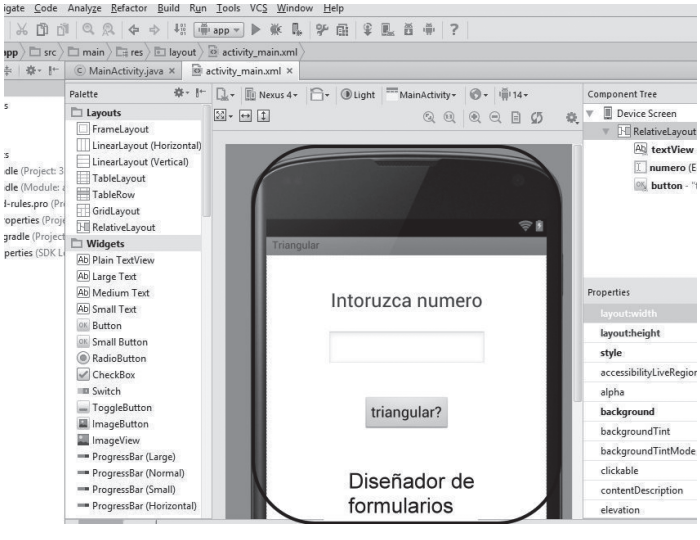

Figura 1.26. *Diseñador de formularios en Android Studio*

1.5.4 EDITOR DE CÓDIGO

Un típico editor de texto multiventana para ver y editar el código de la aplicación. Está perfectamente integrado con las propiedades y el diseñador de formularios.

http://elvex.ugr.es/decsai/builder/intro/2.html

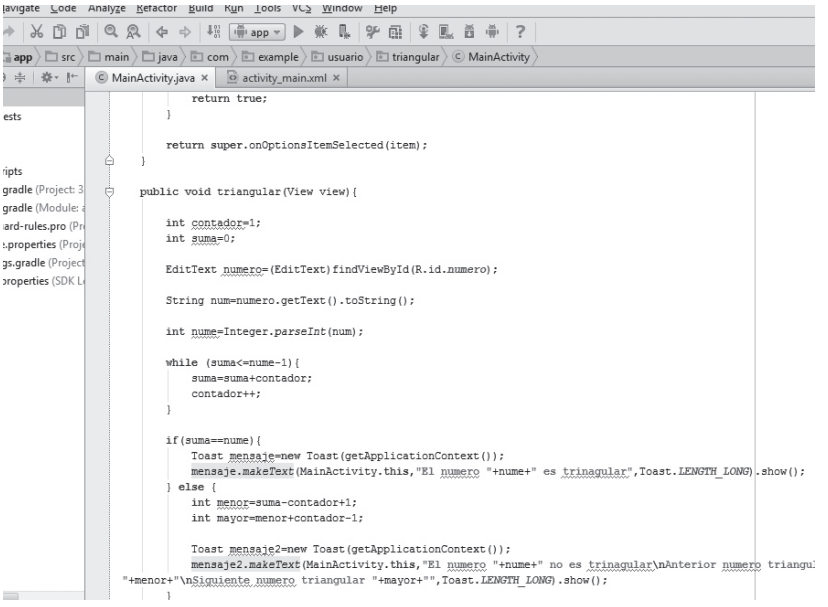

Figura 1.27. *Editor de texto*

1.6 CLASES, PROPIEDADES, MÉTODOS

Una **clase** es una plantilla para la creación de objetos de datos según un modelo predefinido. Las clases se utilizan para representar entidades o conceptos, como los sustantivos en el lenguaje. Cada clase es un modelo que define un conjunto de variables -el estado- y **métodos** apropiados para operar con dichos datos -el comportamiento-. Cada objeto creado a partir de la clase se denomina instancia de la clase.

Las clases son un pilar fundamental de la programación orientada a objetos. Permiten abstraer los datos y sus operaciones asociadas al modo de una caja negra. Los lenguajes de programación que soportan clases difieren sutilmente en su soporte para diversas características relacionadas con clases. La mayoría soportan diversas formas de herencia. Muchos lenguajes también soportan características para proporcionar encapsulación, como especificadores de acceso.

Una clase también puede tener una representación (metaobjeto) en tiempo de ejecución, que proporciona apoyo en tiempo de ejecución para la manipulación de los metadatos relacionados con la clase.

Las clases se componen de elementos, llamados genéricamente miembros, de varios tipos:

- **Campos de datos**: los campos de datos se utilizan para contener datos que reflejan el estado de la clase. Los datos pueden estar almacenados en variables, o estructuras más complejas, como *structs*, uniones e incluso otras clases. Habitualmente, las variables "miembro" son privadas al objeto (siguiendo las directrices de diseño

del Principio de ocultación) y su acceso se realiza mediante propiedades o métodos que realizan comprobaciones adicionales.

- **Métodos**: los métodos implementan la funcionalidad asociada al objeto. Los métodos son el equivalente a las funciones en programación estructurada. Se diferencian de ellos en que es posible acceder a las variables de la clase de forma implícita. Cuando se desea realizar una acción sobre un objeto, se dice que se le manda un mensaje invocando a un método que realizará la acción.
- Ciertos lenguajes permiten un tercer tipo de miembro: las **propiedades**. Las propiedades son un tipo especial de métodos. Debido a que suele ser común que las variables miembro sean privadas para controlar el acceso y mantener la coherencia, surge la necesidad de permitir consultar o modificar su valor mediante pares de métodos: *GetVariable* y *SetVariable*. Los lenguajes orientados a objetos más modernos (por ejemplo Java o C#) añaden la construcción de propiedad, que es una sintaxis simplificada para dichos métodos.

1.6.1 EJEMPLO DE UNA CLASE EN JAVA

En el siguiente ejemplo vamos a crear una clase en Java llamada *Taxi*:

```
//Esta clase representa un taxi. -- > Comentario general que puede incluir: cometido,
autor, versión, etc…
public class Taxi { //El nombre de la clase
    String ciudad; //Ciudad de cada objeto taxi
    String matricula; //Matrícula de cada objeto taxi
    String distrito; //Distrito asignado a cada objeto taxi
     int tipoMotor; //tipo de motor asignado a cada objeto taxi. 0 = desconocido, 1 =
gasolina, 2 = diesel

    //Constructor: cuando se cree un objeto taxi se ejecutará el código que incluyamos
en el constructor
    public Taxi () {
        ciudad = "México D.F.";
        matricula = "";
        distrito = "Desconocido";
        tipoMotor = 0;
    } //Cierre del constructor … el código continúa …

    //Método para establecer la matrícula de un taxi
    public void setMatricula (String valorMatricula) {
        matricula = valorMatricula; //La matrícula del objeto taxi adopta el valor que
contenga valorMatricula
    } //Cierre del método
    //Método para establecer el distrito de un taxi
    public void setDistrito (String valorDistrito) {
        distrito = "Distrito " + valorDistrito; //El distrito del objeto taxi adopta el
valor indicado
```

```
    } //Cierre del método
    public void setTipoMotor (int valorTipoMotor) {
        tipoMotor = valorTipoMotor; //El tipoMotor del objeto taxi adopta el valor que
contenga valorTipoMotor
    } //Cierre del método
    //Método para obtener la matrícula del objeto taxi
    public String getMatricula () { return matricula; } //Cierre del método
    //Método para obtener el distrito del objeto taxi
    public String getDistrito () { return distrito; } //Cierre del método
    //Método para obtener el tipo de motor del objeto taxi
    public int getTipoMotor () { return tipoMotor; } //Cierre del método
} //Cierre de la clase
```

Hemos creado una clase denominada *Taxi*. El espacio comprendido entre la apertura de la clase y su cierre, es decir, el espacio entre los símbolos { y } de la clase, se denomina **cuerpo de la clase**.

Todo objeto de tipo *Taxi* tendrá los mismos atributos: una matrícula (cadena de caracteres), un distrito (cadena de caracteres) y un tipo de motor (valor entero 0, 1 ó 2 representando desconocido, gasolina o diesel). Los atributos los definiremos normalmente después de la apertura de la clase, fuera de los constructores o métodos que puedan existir.

Hemos definido que cualquier objeto *Taxi* que se cree tendrá, inicialmente, estos atributos: como matrícula una cadena vacía; como distrito "Desconocido"; y como tipo de motor 0, que es el equivalente numérico de desconocido. La sintaxis que hemos utilizado para el constructor es *public nombreDeLaClase* { … }

Por otro lado, hemos establecido que todo objeto *Taxi* podrá realizar estas operaciones: recibir un valor de matrícula y quedar con esa matrícula asignada (*setMatricula*); recibir un valor de distrito y quedar con ese distrito asignado (*setDistrito*); recibir un valor de tipo de motor y quedar con ese valor asignado (*setTipoMotor*). Devolver su matrícula cuando se le pida (*getMatricula*); devolver su distrito cuando se le pida (*getDistrito*); devolver su tipo de motor cuando se le pida (*getTipoMotor*).

Para crear objetos *Taxi* debemos asignar a una variable el valor new *Taxi()*. De esta manera podemos crear 5 objetos *taxi1*, *taxi2*, *taxi3*, *taxi4* y *taxi5*. Cada objeto *Taxi* tiene tres atributos: *matricula*, *distrito* y *tipoMotor*. En total tendremos 5 *taxis* x 3 *atributos* = 15 atributos.

Un objeto es una instancia de una clase: por eso a los atributos que hemos definido se les denomina "variables de instancia", porque cada instancia es "portadora" de esos atributos. También es frecuente utilizar el término "campos de la clase" como equivalente. Cada clase tendrá sus campos específicos. Por ejemplo, si una clase representa una moneda sus campos pueden ser *país*, *nombreMoneda*, *valor*, *diámetro*, *grosor*. Si una clase representa una persona sus campos pueden ser *nombre*, *apellidos*, *DNI*, *peso* y *altura*.

ACTIVIDADES 1.10

- En este apartado se ha presentado un ejemplo de una clase en Java. Investigue en Internet y presente aquí una clase escrita en Visual Basic .NET.

1.7 COMPONENTES

1.7.1 CARACTERÍSTICAS Y CAMPO DE APLICACIÓN

La programación orientada a componentes (que también es llamada basada en componentes) es una rama de la ingeniería del software, con énfasis en la descomposición de sistemas ya conformados en componentes funcionales o lógicos, con interfaces bien definidas usadas para la comunicación entre componentes.

Se considera que el nivel de abstracción de los componentes es más alto que el de los objetos y, por lo tanto, no comparten un estado y se comunican intercambiando mensajes que contienen datos.

Un componente de software es un elemento de un sistema que ofrece un servicio predefinido, y es capaz de comunicarse con otros componentes.

Una definición más simple puede ser: un **componente** es un objeto escrito de acuerdo a unas especificaciones. No importa qué especificación sea esta, siempre y cuando el objeto se adhiera a la especificación. Solo cumpliendo correctamente con esa especificación es que el objeto se convierte en componente y adquiere características, como reusabilidad.

El objetivo de la Programación Orientada a Componentes (POC) es construir un mercado global de componentes software, donde los usuarios son los desarrolladores de las aplicaciones que necesitan reutilizar componentes ya hechos y testeados para construir sus aplicaciones de forma más rápida y robusta.

En general, puede verse como una extensión natural de la programación orientada a objetos dentro del ámbito de los sistemas de aplicación abiertos y distribuidos.

Las entidades básicas de la POC son los componentes, estos pueden interpretarse como cajas negras que encapsulan cierta funcionalidad y que son diseñadas sin saber quién los utilizará, ni cómo, ni cuándo. Los servicios de los componentes son conocidos mediante sus interfaces y requisitos.

La POC es un paradigma de programación que se centra en el diseño e implementación de componentes y, en particular, en los conceptos de encapsulación, polimorfismo, composición tardía y seguridad.

Cuando se necesita el acceso a un componente o cuando este debe ser compartido entre distintas redes, se recurre a procesos como la serialización para entregar el componente a su destino.

La capacidad de ser reutilizado (reusability), es una característica importante de los componentes de software de alta calidad. Un componente debe ser diseñado e implementado, de tal forma que pueda ser reutilizado en muchos programas diferentes.

Requiere gran esfuerzo y atención escribir un componente que es realmente reutilizable. Para esto, el componente debe estar:

- Completamente documentado.
- Probado intensivamente.
  - Debe ser robusto, comprobando la validez de las entradas.
  - Debe ser capaz de pasar mensajes de error apropiados.
- Diseñado pensando en que será usado de maneras imprevistas.

La POC es una disciplina muy joven y, por tanto, en la que los resultados obtenidos se centran más en la identificación de los problemas que en la resolución de los mismos. Algunos de los retos y problemas con los que se enfrenta son:

- **Clarividencia**: se refiere a la dificultad con la que se encuentra el diseñador de un componente al realizar su diseño, pues no conoce ni quién lo utilizará, ni cómo, ni en qué entorno, ni para qué aplicación. Este problema está muy ligado a la composición tardía y reusabilidad de los componentes.
- **Evolución de los componentes**: la gestión de la evolución es un problema serio, pues en los sistemas grandes han de poder coexistir varias versiones de un mismo componente.
- **Percepción del entorno**: posibilidad de un componente de descubrir tanto el tipo de entorno en donde se está ejecutando (de diseño o de ejecución), como los servicios y recursos disponibles en él.
- **Particularización**: cómo particularizar los servicios que ofrece un componente para adaptarlo a las necesidades y requisitos concretos de la aplicación, sin poder manipular su implementación.
- **Falta de soporte formal**: la dificultad que encuentran los métodos formales para trabajar con sus peculiaridades, como puede ser la composición tardía, el polimorfismo o la evolución de los componentes.
- **El problema de la clase base frágil (FBCP)**: este problema ocurre cuando la superclase de una clase sufre modificaciones. El FBCP existe a dos niveles: sintáctico y semántico. A nivel sintáctico ocurre cuando las modificaciones de la superclase son puramente a este nivel. A nivel semántico ocurre cuando lo que se altera es la implementación de los métodos de la superclase.
- **Asincronía y carreras de eventos**: problema que se presenta por los tiempos de comunicación en los sistemas abiertos (no se pueden despreciar retrasos). Es muy difícil garantizar el orden relativo en el que se distribuyen los eventos. El proceso de difusión de eventos es complicado cuando los emisores y receptores pueden cambiar con el tiempo.
- **Interoperabilidad**: los contratos de los componentes se reducen a la definición de sus interfaces a nivel sintáctico, y la interoperabilidad se reduce a la comprobación de los nombres y perfiles de los métodos. Sin embargo, es necesario ser capaces de buscar los servicios que se necesitan por algo más que sus nombres, y poder utilizar los métodos ofrecidos en una interfaz en el orden adecuado.

1.8 ENLACE DE COMPONENTES A ORÍGENES DE DATOS

Casi todas las aplicaciones y servicios necesitan almacenar y obtener acceso a un determinado tipo de datos. Por ejemplo, una aplicación comercial necesaria almacenar datos de productos, clientes y pedidos.

En la siguiente figura se muestra cómo la capa de datos lógicos de una aplicación consta de uno o varios almacenes de datos y describe una capa de componentes lógicos de acceso a datos.

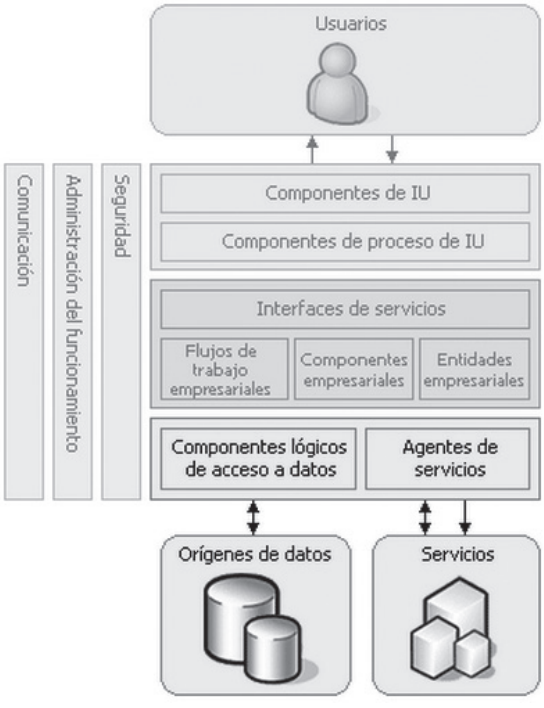

Figura 1.28. *Capa de acceso a datos y los componentes de lógica de acceso a datos*

La mayoría de las aplicaciones utilizan una base de datos relacional como almacén principal de los datos de la aplicación.

Cuando la aplicación recupera datos de la base de datos, puede hacerlo utilizando un formato de conjunto de datos *DataReader*. A continuación los datos se transfieren entre las capas y los distintos niveles de la aplicación y, finalmente, uno de los componentes los utilizará.

Independientemente del almacén de datos utilizado, la aplicación o el servicio utilizarán componentes lógicos de acceso a datos para obtener acceso a los datos. Estos componentes abstraen la semántica del almacén de datos subyacente y la tecnología de acceso a datos (como ADO.NET) y proporcionan una interfaz simple de programación para la recuperación y realización de operaciones con datos.

Los componentes lógicos de acceso a datos suelen implementar un patrón de diseño sin estado que separa el procesamiento empresarial de la lógica de acceso a datos. Cada uno de estos componentes suele proporcionar métodos para realizar operaciones Create, Read, Update y Delete.

Cuando la aplicación contiene varios componentes lógicos de acceso a datos, puede resultar útil utilizar un componente de ayuda de acceso a datos genéricos para administrar las conexiones de las bases de datos, ejecutar comandos y almacenar parámetros en caché, entre otros.

El componente de ayuda para el acceso a datos centraliza el desarrollo de API de acceso a datos y la configuración de la conexión a estos, permitiendo de esta forma la reducción de código duplicado.

Observe los siguientes puntos en la figura:

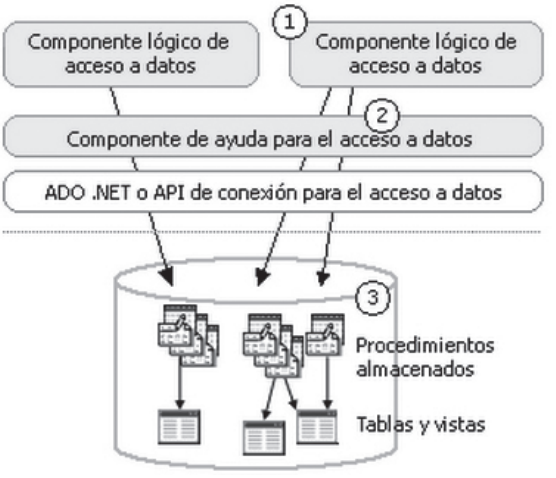

Figura 1.29. *Componentes lógicos de acceso a datos*

- Los componentes lógicos de acceso a datos exponen métodos para insertar, eliminar, actualizar y recuperar datos, incluyendo la provisión de funcionalidad de paginación al recuperar grandes cantidades de datos.
- Puede utilizar un componente de ayuda de acceso a datos para centralizar la administración de la conexión y todo el código relacionado con un origen de datos específico.
- Se recomienda implementar las consultas y operaciones de datos como procedimientos almacenados (si es compatible con el origen de datos) para mejorar el rendimiento y la facilidad de mantenimiento.

1.9 INTERFACES RELACIONADAS CON EL ENLACE DE DATOS

Generalmente, se pueden crear muchas estructuras de datos distintas para satisfacer las necesidades de enlace de la aplicación y de los datos con los que trabaja. Quizás desee crear sus propias clases que proporcionen o utilicen datos de formularios. Estos objetos pueden ofrecer diversos niveles de funcionalidad y complejidad, desde el enlace básico de datos hasta la compatibilidad en tiempo de diseño, la comprobación de errores, la notificación de cambios o, incluso, la compatibilidad con una reversión estructurada de las modificaciones realizadas en los propios datos.

Para ilustrar esto nos vamos a servir de ADO.NET. **ADO.NET** es un conjunto de clases que exponen servicios de acceso a datos para programadores de .NET Framework. ADO.NET ofrece abundancia de componentes para la creación de aplicaciones de uso compartido de datos distribuidas. Constituye una parte integral de .NET Framework y proporciona acceso a datos relacionales, XML y de aplicaciones. ADO.NET satisface diversas necesidades de desarrollo, como la creación de clientes de base de datos *front end* y objetos empresariales de nivel medio que utilizan aplicaciones, herramientas, lenguajes o exploradores de Internet.

1.9.1 INTRODUCCIÓN A LAS HERRAMIENTAS DE DISEÑO DE CONEXIONES ADO .NET

Para trasladar datos entre un almacén de datos y una aplicación, en primer lugar deberá tener una conexión con el almacén de datos.

Los proveedores de datos, las conexiones de datos, los comandos de datos y los lectores de datos son los componentes que constituyen un proveedor de datos de .NET Framework. Microsoft y otros fabricantes pueden poner a disposición otros proveedores de datos de .NET Framework que se pueden integrar en Visual Studio.

En ADO .NET, cree y administre conexiones utilizando objetos de conexión:

- **SqlConnection**: objeto que administra una conexión con un servidor SQL Server 7.0 o posterior. Está optimizado para utilizarse con SQL Server 7.0 o posterior, omitiendo (entre otras cosas) la capa OLE DB.
- **OleDbConnection**: objeto que administra una conexión con cualquier almacén de datos con acceso a través de OLE DB.
- **OdbcConnection**: objeto que administra una conexión a un origen de datos creado utilizando una cadena de conexión o un nombre de origen de datos (DSN) ODBC.
- **OracleConnection**: objeto que administra una conexión a bases de datos Oracle.

1.9.1.1 Cadenas de conexión

Todos los objetos de conexión exponen más o menos los mismos miembros. Sin embargo, los miembros específicos disponibles con un objeto OleDbConnection dado dependen del origen de datos al que se conecta; no todos los orígenes de datos admiten todos los miembros de la clase OleDbConnection.

La principal propiedad asociada a un objeto de conexión es la propiedad ConnectionString, que consiste en una cadena con pares de atributo/valor con la información necesaria para iniciar una sesión en una base de datos y apuntar a una base de datos específica. Una propiedad ConnectionString típica podría tener el siguiente aspecto:

```
Provider= SQLOLEDB.1;Data Source=MySQLServer; Initial Catalog=NORTHWIND; Integrated
Security=SSPI
```

Esta cadena de conexión determinada especifica que la conexión debería usar la seguridad integrada de Windows (autenticación de Windows NT). Una cadena de conexión puede incluir un identificador y una contraseña de usuario, pero esto no resulta aconsejable, ya que estos atributos se compilan en la aplicación y, por lo tanto, suponen una posible infracción de seguridad. Para obtener más información, vea *Permisos de acceso para aplicaciones Web* y *Modelo de seguridad*.

SEGURIDAD

El almacenamiento de los detalles de la cadena de conexión (como el nombre de servidor, el nombre de usuario y la contraseña) puede afectar a la seguridad de la aplicación. El uso de la *Seguridad integrada de Windows* es un modo más seguro de controlar el acceso a una base de datos. Para obtener más información, vea *Seguridad de bases de datos*.

Los pares atributo/valor que OLE DB utiliza con más frecuencia están representados también, por separado, por medio de propiedades individuales, como DataSource y Database. Cuando trabaje con un objeto de conexión, puede establecer la propiedad ConnectionString como una sola cadena o establecer propiedades individuales de conexión (si el origen de datos necesita valores de cadena de conexión que no estén representados por propiedades individuales, deberá establecer la propiedad ConnectionString). También puede establecer la propiedad ConnectionString en la ruta de acceso de un archivo Microsoft Data Link (.udl). Para obtener más información acerca de los archivos de vínculos de datos vea *Información general sobre la API de vínculos de datos*.

El objeto SQLConnection no admite ni permite especificar el atributo *Provider*.

1.9.1.2 Abrir y cerrar conexiones

Los dos métodos más importantes para las conexiones son *Open* y *Close*. El método *Open* utiliza la información de la propiedad ConnectionString para ponerse en contacto con el origen de datos y establecer una conexión abierta. El método *Close* cierra la conexión. Es esencial cerrar las conexiones, porque la mayoría de los orígenes de datos admiten solo un número limitado de conexiones abiertas y las conexiones abiertas consumen valiosos recursos del sistema.

Si está trabajando con adaptadores de datos o comandos de datos, no tendrá que abrir ni cerrar conexiones explícitamente. Cuando se llama a un método de estos objetos (por ejemplo, a los métodos *Fill* o *Update* del adaptador de datos), el método comprueba si ya está abierta la conexión. Si no es así, el adaptador abre la conexión, ejecuta su lógica y cierra de nuevo la conexión.

Los métodos como *Fill* solo abren y cierran la conexión automáticamente si no está ya abierta. Si la conexión está abierta, los métodos la utilizan pero no la cierran. Esto le proporciona la flexibilidad necesaria para abrir y cerrar comandos de datos. Puede hacer esto si tiene múltiples adaptadores de datos que compartan una conexión. En este caso, hacer que cada adaptador abra y cierre la conexión cuando se llame a su método *Fill* no resulta eficiente. En lugar de hacerlo así, puede abrir la conexión, llamar al método *Fill* de cada adaptador y, a continuación, cuando termine, cerrar la conexión.

1.9.1.3 Agrupar conexiones

A menudo, las aplicaciones tienen diferentes usuarios ejecutando el mismo tipo de acceso a la base de datos. Por ejemplo, en las aplicaciones Web ASP.NET, es posible que muchos usuarios estén consultando la misma base de datos para obtener los mismos datos. En esos casos, el rendimiento de la aplicación puede mejorar si se hace que la aplicación comparta o "agrupe" las conexiones con el origen de datos. La sobrecarga que supone hacer que cada usuario abra y cierre una conexión separada podría tener, de lo contrario, un efecto adverso sobre el rendimiento de la aplicación.

Si está utilizando la clase OleDbConnection, OdbcConnection u OracleConnection, el proveedor controlará automáticamente la agrupación de conexiones, así que no tendrá que preocuparse de administrarla personalmente.

Si está utilizando la clase SqlConnection, la agrupación de conexiones se administra de forma implícita, pero también dispone de opciones que le permiten administrar la agrupación personalmente. Para obtener más información, vea *Agrupar conexiones del proveedor de datos .Net Framework para SQL Server*.

1.9.1.4 Transacciones

Los objetos de conexión admiten transacciones con un método *BeginTransaction* que crea un objeto de transacción (por ejemplo, un objeto *SqlTransaction*). El objeto de transacción, a su vez, admite métodos que permiten confirmar o deshacer las transacciones.

Las transacciones se administran en el código. Para obtener más información, vea *Realizar transacciones*.

1.9.1.5 Propiedades de conexión configurables

En muchas aplicaciones, la información de conexión no se puede determinar en tiempo de diseño. Por ejemplo, en una aplicación que se vaya a distribuir a muchos clientes diferentes, es posible que no pueda determinar, en tiempo de diseño, la información de conexión (por ejemplo, el nombre de un servidor).

Por lo tanto, las cadenas de conexión se especifican a menudo como propiedades dinámicas. Puesto que las propiedades dinámicas se almacenan en un archivo de configuración (y no se compilan en los archivos binarios de aplicación) pueden modificarse sin tener que compilar de nuevo la aplicación.

Una estrategia típica consiste en especificar las propiedades de conexión en forma de propiedades dinámicas, proporcionar un medio (tal como un formulario Windows Forms o una página de formularios Web Forms) para que los usuarios especifiquen los detalles relevantes y, a continuación, actualizar el archivo de configuración. El mecanismo de propiedades dinámicas integrado en .NET Framework obtiene automáticamente los valores del archivo de configuración al leer la propiedad y actualiza el archivo cuando se actualiza el valor.

1.9.1.6 Información de conexión y seguridad

Dado que la apertura de una conexión implica obtener acceso a un recurso importante (una base de datos) a menudo surgen problemas de seguridad al configurar una conexión y trabajar con ella.

Cómo asegurar la aplicación y su acceso al origen de datos depende de la arquitectura del sistema. En una aplicación basada en Web, por ejemplo, los usuarios suelen obtener un acceso anónimo a Internet Information Services (IIS) y, por lo tanto, no proporcionan credenciales de seguridad. En este caso, la aplicación mantiene su propia información de inicio de sesión y la utiliza (en lugar de cualquier información específica del usuario) para abrir la conexión y tener acceso a la base de datos.

SEGURIDAD

El almacenamiento de los detalles de la cadena de conexión (como el nombre de servidor, el nombre de usuario y la contraseña) puede afectar a la seguridad de la aplicación. El uso de la Seguridad integrada de Windows es un modo más seguro de controlar el acceso a una base de datos. Para obtener más información, vea *Seguridad de bases de datos*.

Los archivos binarios y el archivo de configuración de la aplicación Web ASP .NET están asegurados frente al acceso Web por medio del modelo de seguridad inherente a ASP .NET, que impide el acceso a estos archivos a través de cualquier protocolo de Internet (HTTP, FTP, etc.). Para impedir el acceso al propio equipo servidor Web a través de otros métodos (por ejemplo, desde la red interna), utilice las funciones de seguridad de Windows.

En aplicaciones de intranet o de dos niveles, puede aprovechar la opción de seguridad integrada que proporcionan Windows, IIS y SQL Server. En este modelo, las credenciales de autenticación de un usuario para la red local se utilizan también para el acceso a los recursos de la base de datos y no se utiliza ninguna contraseña o nombre de usuario explícito en la cadena de conexión (habitualmente, los permisos se establecen en el equipo servidor de la base de datos por medio de grupos, de modo que no es necesario establecer permisos individuales para cada usuario que pueda tener acceso a la base de datos). En este modelo, no es necesario almacenar ninguna información de inicio de sesión para la conexión ni es necesario dar más pasos para proteger la información de la cadena de conexión.

1.9.1.7 Conexiones en tiempo de diseño en el Explorador de servidores

El Explorador de servidores proporciona un medio para crear conexiones en tiempo de diseño con orígenes de datos. Permite examinar los orígenes de datos disponibles, mostrar información acerca de las tablas, columnas y otros elementos que contienen, y modificar y crear elementos de la base de datos.

La aplicación no utiliza directamente las conexiones creadas de esta manera. Generalmente, la información que proporciona una conexión en tiempo de diseño se utiliza para establecer las propiedades de un nuevo objeto de conexión que se agrega a la aplicación.

Por ejemplo, podría utilizar en tiempo de diseño el Explorador de servidores para crear una conexión con SQL Server MyServer y la base de datos Northwind. Más tarde, al diseñar un formulario, podría examinar la base de datos Northwind, seleccionar columnas de una tabla y arrastrarlas al formulario. Esto crea un adaptador de datos para el formulario. También crea una nueva conexión específica para el formulario, al copiar información de cadena de conexión desde la conexión en tiempo de diseño a la nueva conexión. Cuando se ejecute la aplicación, se utilizará la nueva conexión para el acceso al origen de datos.

La información sobre las conexiones en tiempo de diseño de almacena en el equipo local, independientemente del proyecto o solución específicos. Por lo tanto, una vez establecida una conexión en tiempo de diseño mientras se trabaja en una aplicación, aparecerá en el Explorador de servidores cada vez que trabaje en Visual Studio (siempre que esté disponible el servidor al que apunta la conexión). Para obtener más información acerca del uso del Explorador de servidores y de la creación de conexiones en tiempo de diseño, vea Agregar nuevas conexiones de datos en el Explorador de servidores.

La conexión de datos que cree en el Explorador de servidores determina los objetos que se crean cuando se arrastran elementos de base de datos al diseñador. Por ejemplo, si se va a conectar a un servidor SQL Server en tiempo de diseño y la conexión de datos se configuró seleccionando el Proveedor OLE DB para SQL Server, si arrastra una tabla desde el Explorador de servidores hasta el diseñador creará un objeto SqlConnection. Si la aplicación requiere una conexión ODBC, asegúrese de que selecciona el Proveedor OLE DB para ODBC cuando cree la conexión de datos en el Explorador de servidores.

1.9.1.8 Herramientas de diseño de conexiones en Visual Studio

Habitualmente, no necesitará crear ni administrar directamente objetos de conexión en Visual Studio. Cuando utilice herramientas tales como el Asistente para el adaptador de datos, las herramientas le pedirán normalmente la información de conexión (es decir, la información de la cadena de conexión) y crearán automáticamente objetos de conexión en el formulario o componente con el que esté trabajando.

No obstante, si lo desea, puede agregar objetos de conexión a un formulario o componente y establecer sus propiedades. Esto es útil si no se está trabajando con conjuntos de datos (y, por lo tanto, tampoco con adaptadores de datos), sino simplemente leyendo datos. Si está utilizando transacciones, podría crear objetos de conexión personalmente.

1.9.2 INTERFACES DISEÑADAS PARA QUE CONSUMIDORES DEL ORIGEN DE DATOS LAS UTILICEN E INTERFACES DISEÑADAS PARA QUE LAS UTILICEN LOS CREADORES DE COMPONENTES

En las secciones siguientes se describen dos grupos de objetos de interfaz. El primer grupo muestra interfaces que se implementan en orígenes de datos por autores de orígenes de datos. Estas interfaces están diseñadas para que consumidores del origen de datos las utilicen, que la mayoría de los casos son controles o componentes de formularios *Windows Forms*.

El segundo grupo muestra interfaces diseñadas para que las utilicen los autores de componentes. Los autores de componentes utilizan estas interfaces cuando crean componentes que admiten que el motor de enlace de datos de formularios *Windows Forms* utilice el enlace de datos. Estas interfaces pueden implementarse en las clases asociadas al formulario para habilitar el enlace de datos; cada caso presenta una clase que implementa una interfaz que habilita la interacción con los datos. Las herramientas con experiencia en diseño de datos de desarrollo rápido de aplicaciones (RAD) de Visual Studio ya aprovechan esta funcionalidad.

1.9.2.1 Interfaces diseñadas para que consumidores del origen de datos las utilicen

Las interfaces siguientes están diseñadas para ser utilizadas por controles de formularios Windows Forms:

1.9.2.1.1 La interfaz IList

Una clase que implementa la interfaz IList podría ser Array, ArrayList o CollectionBase. Estas son listas de elementos indizadas de tipo Object. Estas listas deben contener tipos homogéneos porque el primer elemento del índice determina el tipo. IList solo debe estar disponible para el enlace en tiempo de ejecución.

1.9.2.1.2 La interfaz IBindingList

Una clase que implementa la interfaz IBindingList proporciona un nivel mucho mayor de funcionalidad de enlace de datos. Esta implementación ofrece posibilidades de ordenación básicas y notificación de cambios, ambas para cuando cambian los elementos de lista (por ejemplo, el tercer elemento de una lista de clientes tiene un cambio en el campo *Dirección*), así como cuando cambia la lista en sí (por ejemplo, aumenta o disminuye el número de elementos de la lista). Es importante la notificación de cambios si tiene previsto tener varios controles enlazados a los mismos datos y desea que los cambios de datos realizados en uno de los controles se propaguen a los demás controles enlazados.

1.9.2.1.3 La interfaz IBindingListView

Una clase que implementa la interfaz IBindingListView proporciona todas las funciones de una implementación de IBindingList, así como funciones de filtrado y ordenación avanzada. Esta implementación proporciona el filtrado basado en cadena y la ordenación multicolumna con pares de descriptor de propiedad y dirección.

1.9.2.1.4 La interfaz IEditableObject

Una clase que implementa la interfaz IEditableObject permite que un objeto controle cuándo se hacen permanentes los cambios realizados en el objeto. Esta implementación ofrece los métodos BeginEdit, EndEdit y CancelEdit, que le permiten revertir los cambios realizados en el objeto. A continuación figura una breve descripción del funcionamiento de los métodos BeginEdit, EndEdit y CancelEdit y cómo funcionan entre ellos para permitir la posibilidad de revertir los cambios realizados en los datos:

El método BeginEdit señala el inicio de una modificación de un objeto. Un objeto que implemente esta interfaz necesitará almacenar las actualizaciones que se produzcan después de la llamada al método BeginEdit de modo que puedan descartarse las actualizaciones si se llama al método CancelEdit. En el enlace de datos de formularios Windows Forms, puede llamar varias veces al método BeginEdit dentro del ámbito de una transacción de edición única (por ejemplo, BeginEdit, BeginEdit, EndEdit). Las implementaciones de IEditableObject deberían mantener el seguimiento que comprueba si se ha llamado ya al método BeginEdit y omitir las llamadas subsiguientes a BeginEdit. Dado que es posible llamar varias veces a este método, es importante que las llamadas subsiguientes no sean destructivas; es decir, las llamadas subsiguientes a BeginEdit no pueden destruir las actualizaciones realizadas ni modificar los datos guardados en la primera llamada a BeginEdit.

1.9.2.1.5 La interfaz ICancelAddNew

Una clase que implementa la interfaz ICancelAddNew normalmente implementa la interfaz IBindingList y permite revertir las agregaciones realizadas en el origen de datos con el método AddNew. Si el origen de datos implementa la interfaz IBindingList, establezca también que implemente la interfaz ICancelAddNew.

1.9.2.1.6 La interfaz IDataErrorInfo

Una clase que implementa la interfaz IDataErrorInfo permite a los objetos ofrecer la información de error personalizada a los controles enlazados:

La propiedad *Error* devuelve un texto de mensaje de error general (por ejemplo, "Se produjo un error").

La propiedad *Item* devuelve una cadena con el mensaje de error específico de la columna (por ejemplo, "El valor de la columna *State* no es válido").

1.9.2.1.7 La interfaz IEnumerable

ASP.NET utiliza normalmente una clase que implementa la interfaz IEnumerable. Solo está disponible la compatibilidad de formularios Windows Forms con esta interfaz a través del componente BindingSource.

1.9.2.1.8 La interfaz ITypedList

Una clase de colecciones que implementa la interfaz ITypedList proporciona la posibilidad de controlar el orden y el conjunto de propiedades expuestas al control enlazado.

1.9.2.1.9 La interfaz ICustomTypeDescriptor

Una clase que implementa la interfaz ICustomTypeDescriptor proporciona información dinámica sobre sí misma. Esta interfaz es similar a ITypedList pero la utilizan objetos en lugar de listas. DataRowView utiliza esta interfaz para proyectar el esquema de las filas subyacentes. La clase CustomTypeDescriptor proporciona una implementación sencilla de ICustomTypeDescriptor.

1.9.2.1.10 La interfaz IListSource

Una clase que implementa la interfaz IListSource habilita el enlace basado en lista en objetos que no son lista. El método *GetList* de IListSource se usa para devolver una lista enlazable de un objeto que no hereda de IList. La clase *DataSet* utiliza IListSource.

1.9.2.1.11 La interfaz IRaiseItemChangedEvents

Una clase que implementa la interfaz IRaiseItemChangedEvents es una lista enlazable que también implementa la interfaz IBindingList. Esta interfaz se utiliza para indicar si su tipo provoca eventos ListChanged de tipo ItemChanged a través de la propiedad RaisesItemChangedEvents.

1.9.2.1.12 La interfaz ISupportInitialize

Un componente que implementa la interfaz ISupportInitialize aprovecha las ventajas de las optimizaciones por lotes para establecer las propiedades e inicializar propiedades codependientes. ISupportInitialize contiene dos métodos:

- BeginInit señala que comienza la inicialización del objeto.
- EndInit señala que finaliza la inicialización del objeto.

1.9.2.1.13 La interfaz ISupportInitializeNotification

Un componente que implementa la interfaz ISupportInitializeNotification también implementa la interfaz ISupportInitialize. Esta interfaz permite notificar a otros componentes ISupportInitialize que la inicialización está completa. La interfaz ISupportInitializeNotification contiene dos miembros:

1.9.2.1.14 La interfaz INotifyPropertyChanged

Una clase que implementa esta interfaz es un tipo que provoca un evento cuando cualquiera de sus valores de propiedad cambia. Esta interfaz está diseñada para reemplazar al modelo de tener un evento de cambio para cada propiedad de un control. Cuando se utiliza en una clase *BindingList<T>*, un objeto comercial debe implementar la interfaz INotifyPropertyChanged y BindingList`1 convertirá los eventos PropertyChanged a eventos ListChanged de tipo ItemChanged.

1.9.3 INTERFACES DISEÑADAS PARA QUE LAS UTILICEN LOS CREADORES DE COMPONENTES

Las interfaces siguientes están diseñadas para que las utilice el motor de enlace de datos de formularios Windows Forms:

1.9.3.1 La interfaz IBindableComponent

Una clase que implementa esta interfaz es un componente que no sea un control que admite el enlace de datos. Esta clase devuelve los enlaces de datos y el contexto de enlace del componente a través de las propiedades DataBindings y BindingContext de esta interfaz.

1.9.3.2 La interfaz ICurrencyManagerProvider

Una clase que implementa la interfaz ICurrencyManagerProvider es un componente que proporciona su propio CurrencyManager para administrar los enlaces asociados a este componente determinado. La propiedad CurrencyManager proporciona el acceso al CurrencyManager personalizado.

1.10 CUADROS DE DIÁLOGO: DIÁLOGOS MODALES Y NO MODALES

Un cuadro de diálogo es un tipo de ventana que permite comunicación simple entre el usuario y el sistema informático.

El tipo cuadro de diálogo más simple únicamente informa al usuario, es decir, que muestran un texto (y eventualmente objetos gráficos) y ofrece la opción de cerrar el cuadro. Un ejemplo es un cuadro de error.

Luego existen cuadros de pregunta o confirmación, que además de mostrar información ofrecen alternativas al usuario. La más sencilla es una opción binaria como aceptar/cancelar o permitir/impedir.

Existen versiones más complejas con más opciones. Por ejemplo, si el usuario intenta cerrar un editor de texto y el documento abierto tiene cambios sin guardar, un cuadro de diálogo completo podría mostrar cuatro opciones: «cerrar sin guardar», «guardar y salir», «cancelar el cierre y seguir editando» y «guardar con otro nombre» (ésta última con una caja de texto donde ingresar el nombre alternativo).

Los cuadros de diálogo se los clasifica en **modales** y **no modales**, según si impiden o permiten que el usuario continúe usando el programa ignorando el cuadro. Los cuadros modales se suelen usar para mostrar información crítica y ante eventos peligrosos y acciones irreversibles.

En ocasiones, se usan cuadro de diálogo para paliar la ausencia de funcionalidad de revertir acciones. Los expertos en usabilidad afirman que es un mecanismo pobre, ya que desconcierta al usuario ante el cambio brusco en el funcionamiento del programa, y motiva a ignorar la información del cuadro.

ACTIVIDADES 1.11

- Busque en Internet y realice dos capturas, una será de un cuadro de diálogo modal y otra será de un cuadro de diálogo no modal.

1.11 EVENTOS. ESCUCHADORES

La programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen.

Para entender la programación dirigida por eventos, podemos oponerla a lo que no es: mientras en la programación secuencial (o estructurada) es el programador el que define cuál va a ser el flujo del programa, en la programación dirigida por eventos será el propio usuario -o lo que sea que esté accionando el programa- el que dirija el flujo del programa. Aunque en la programación secuencial puede haber intervención de un agente externo al programa, estas intervenciones ocurrirán cuando el programador lo haya determinado, y no en cualquier momento, como puede ser en el caso de la programación dirigida por eventos.

El creador de un programa dirigido por eventos debe definir los eventos que manejarán su programa y las acciones que se realizarán al producirse cada uno de ellos, lo que se conoce como el administrador de evento. Los eventos soportados estarán determinados por el lenguaje de programación utilizado, por el sistema operativo e incluso por eventos creados por el mismo programador.

En la programación dirigida por eventos, al comenzar la ejecución del programa se llevarán a cabo las inicializaciones y demás código inicial y a continuación el programa quedará bloqueado hasta que se produzca algún evento. Cuando alguno de los eventos esperados por el programa tenga lugar, el programa pasará a ejecutar el código del correspondiente administrador de evento. Por ejemplo, si el evento consiste en que el usuario ha hecho clic en el

botón de *play* de un reproductor de películas, se ejecutará el código del administrador de evento, que será el que haga que la película se muestre por pantalla.

Un ejemplo claro lo tenemos en los sistemas de programación Léxico y Visual Basic, en los que a cada elemento del programa (objetos, controles, etc.) se le asignan una serie de eventos que generará dicho elemento, como la pulsación de un botón del ratón sobre él o el redibujado del control.

La programación dirigida por eventos es la base de lo que llamamos interfaz de usuario, aunque puede emplearse también para desarrollar interfaces entre componentes de Software o módulos del núcleo.

En los primeros tiempos de la computación, los programas eran secuenciales, también llamados *Batch*. Un programa secuencial arranca, lee parámetros de entrada, procesa estos parámetros y produce un resultado, todo de manera lineal y sin intervención del usuario mientras se ejecuta.

Con la aparición y popularización de los PC, el software empezó a ser demandado para usos alejados de los clásicos académicos y empresariales para los cuales era necesitado hasta entonces, y quedó patente que el paradigma clásico de programación no podía responder a las nuevas necesidades de interacción con el usuario que surgieron a raíz de este hecho.

En contraposición al modelo clásico, la programación orientada a eventos permite interactuar con el usuario en cualquier momento de la ejecución. Esto se consigue debido a que los programas creados bajo esta arquitectura se componen por un bucle exterior permanente encargado de recoger los eventos, y distintos procesos que se encargan de tratarlos. Habitualmente, este bucle externo permanece oculto al programador que simplemente se encarga de tratar los eventos, aunque en algunos lenguajes será necesaria su construcción.

Ejemplo de programa orientado a eventos en pseudocodigo:

```
while (true){
      switch (event){
          case mousse_button_down:
          case mouse_click:
          case keypressed:
          case Else:
      }
  }
```

1.11.1 PROBLEMÁTICA

La programación orientada a eventos supone una complicación añadida con respecto a otros paradigmas de programación, debido a que el flujo de ejecución del software escapa al control del programador. En cierta manera podríamos decir que en la programación clásica el flujo estaba en poder del programador y era este quien decidía el orden de ejecución de los procesos, mientras que en programación orientada a eventos, es el usuario el que controla el flujo y decide.

Pongamos como ejemplo de la problemática existente, un menú con dos botones, botón 1 y botón 2. Cuando el usuario pulsa botón 1, el programa se encarga de recoger ciertos parámetros que están almacenados en un fichero y

calcular algunas variables. Cuando el usuario pulsa el botón 2, se le muestran al usuario por pantalla dichas variables. Es sencillo darse cuenta de que la naturaleza indeterminada de las acciones del usuario y las características de este paradigma pueden fácilmente desembocar en el error fatal de que se pulse el botón 2 sin previamente haber sido pulsado el botón 1. Aunque esto no pasa si se tienen en cuenta las propiedades de dichos botones, haciendo inaccesible la pulsación sobre el botón 2 hasta que previamente se haya pulsado el botón 1.

1.11.2 GUI / INTERFAZ GRÁFICA DE USUARIO

Con la evolución de los lenguajes orientados a eventos, la interacción del software con el usuario ha mejorado enormemente permitiendo la aparición de interfaces que, aparte de ser la vía de comunicación del programa con el usuario, son la propia apariencia del mismo. Estas interfaces, también llamadas GUI (*Graphical User Interface*), han sido la herramienta imprescindible para acercar la informática a los usuarios, permitiendo en muchos casos, a principiantes utilizar de manera intuitiva y sin necesidad de grandes conocimientos, el software que ha colaborado a mejorar la productividad en muchas tareas.

Uno de los periféricos que ha cobrado mayor importancia tras la aparición de los programas orientados a eventos ha sido el ratón, gracias también en parte a la aparición de los sistemas operativos modernos con sus interfaces gráficas. Estas suelen dirigir directamente al controlador interior que va entrelazado al algoritmo.

1.11.3 HERRAMIENTAS VISUALES DE DESARROLLO

Con el paso del tiempo, han ido apareciendo una nueva generación de herramientas que incluyen código que automatiza parte de las tareas más comunes en la detección y tratamiento de eventos.

Destacan particularmente los entornos de programación visual que conjugan una herramienta de diseño gráfica para la Interfaz Grafica de Usuario (GUI) y un lenguaje de alto nivel. Entre estas herramientas se encuentra Visual Basic, un lenguaje altamente apreciado por principiantes debido a la facilidad para desarrollar software en poco tiempo y con pocos conocimientos, y denostado por tantos otros debido a su falta de eficiencia.

1.11.4 CREACIÓN DE ESCUCHADORES DE EVENTOS

Un escuchador de eventos (*event listener*) es un mecanismo que nos permite reaccionar de forma asíncrona ante ciertas circunstancias que ocurren en clases diferentes a la nuestra. Es un sistema de comunicación muy habitual en Java; seguro que lo habrás utilizado para detectar si un botón ha sido pulsado, para saber que cambia la localización del dispositivo o el valor de un sensor. Para usar un escuchador de eventos hemos de seguir tres pasos:

- Implementar la interfaz del escuchador.
- Registrar el escuchador en el objeto que genera el evento indicándole el objeto que los recogerá.
- Implementar los métodos *callback* correspondientes.

1.12 EDICIÓN Y ANÁLISIS DEL CÓDIGO GENERADO POR LA HERRAMIENTA DE DISEÑO

Cuando desarrollamos una interfaz mediante el uso de una herramienta grafica de diseño, esta herramienta realmente está generando código del lenguaje de programación sobre el que se base dicha herramienta y, por tanto, podremos visualizar y analizar el código generado por dicha aplicación.

Como hemos indicado, dependiendo del tipo de aplicación que utilicemos el código en lenguaje de programación será diferente, como hemos indicado anteriormente, los IDE más destacados son:

1.12.1 MICROSOFT VISUAL STUDIO

La plataforma .NET es la propuesta de Microsoft para competir con la plataforma Java. Mientras que Java se caracteriza por la máxima "*write once, run anywhere*", la plataforma .NET de Microsoft está diseñada para que se puedan desarrollar componentes software utilizando casi cualquier lenguaje de programación, de forma que lo que escribamos en un lenguaje pueda utilizarse desde cualquier otro de la manera más transparente posible (utilizando servicios web como *middleware*). Esto es, en vez de estar limitados a un único lenguaje de programación, permitimos cualquier lenguaje de programación, siempre y cuando se adhiera a unas normas comunes establecidas para la plataforma .NET en su conjunto. De hecho, existen compiladores de múltiples lenguajes para la plataforma .NET: Visual Basic .NET, C#, Managed C++, Oberon, Component Pascal, Eiffel, Smalltalk, Cobol, Fortran, Scheme, Mercury, Mondrian/Haskell, Perl, Python, SML.NET...

1.12.2 GAMBAS

Genera código BASIC, similar a Visual Basic, aunque Gambas no es un clon de este último.

1.12.3 GLADE

Genera código XML.

1.12.4 EMBARCADERO DELPHI

Genera código Object Pascal.

1.12.5 NETBEANS Y ECLIPSE

- Netbeans: da soporte a las siguientes tecnologías, entre otras: Java, PHP, Groovy, C/C++, HTML5…
- Eclipse: da soporte a las siguientes tecnologías, entre otras: Java, C/C++, Python y lenguajes de *script* no tipados como PHP o Javascript.

1.12.6 ANJUTA

Soporta los siguientes lenguajes de programación: C, C++, Java, Python y Vala.

1.12.7 DREAMWEAVER

Lenguajes soportados: HTML, XHTML, CSS, JavaScript, ColdFusion Markup Language (CFML), VBScript (para ASP), C# y Visual Basic (para ASP.NET), JSP y PHP.

RESUMEN DEL CAPÍTULO

En este primer capítulo se han tratado varios temas, entre los que destacan los siguientes apartados:

- ✓ Librerías de componentes disponibles para diferentes sistemas operativos y diversos lenguajes de programación: características.
- ✓ Herramientas propietarias y libres de edición de interfaces.
- ✓ Componentes de la interfaz visual: características y campo de aplicación. Localización y alineación.
- ✓ Unión de componentes a los orígenes de los datos.
- ✓ Asociación de acciones a eventos.
- ✓ Diálogos modales e non modales.
- ✓ Edición do código generado por la herramienta de diseño.
- ✓ Clases, propiedades e métodos.
- ✓ Eventos: escuchadores.

TEST DE CONOCIMIENTOS

1 De los siguientes ¿Cuáles son tipos de lenguajes de programación existentes?
- a) Lenguajes de bajo nivel.
- b) Lenguajes de alto nivel.
- c) Ambas son ciertas.
- d) No existen diferentes tipos de lenguajes de programación, todos los lenguajes pertenecen al mismo tipo.

2 ¿Cuántas Generaciones existen en la evolución de los lenguajes de programación?
- a) 3: Primera, segunda y tercera.
- b) 1: Estamos en la primera generación aún.
- c) 4: Primera, segunda, tercera y cuarta.
- d) 5: Primera, segunda, tercera, cuarta y quinta.
- e) Todas son falsas.

3 Una propuesta tecnológica que es adaptada por un comunidad de programadores cuyo núcleo central es incuestionable en cuanto a que únicamente trata de resolver uno o varios problemas claramente determinados, se llama:
- a) Algoritmo.
- b) Paradigma de programación.
- c) Propuesta de programación.
- d) Lenguaje de programación.
- e) Todas son ciertas.
- f) Todas son falsas.

4 La Programación Orientada a Objetos (POO):
- a) Es un paradigma de programación.
- b) Está basada en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.
- c) Su uso se popularizo a principios de los años 90.
- d) Existen múltiples lenguajes de programación que la soportan.
- e) Todas son ciertas.
- f) Todas son falsas.

5 La Programación Dirigida por Eventos:
- a) Es un paradigma de programación.
- b) Tanto las estructura como la ejecución de los programas van determinados por sucesos que ocurren en el sistema.
- c) Un ejemplo de esta programación es Microsoft Visual Basic.
- d) Es la base de la Interfaz de usuario (UI).
- e) Todas son ciertas.
- f) Todas son falsas.

6 La Programación basada en componentes:
- a) Es un paradigma de programación.
- b) Tanto las estructura como la ejecución de los programas van determinados por sucesos que ocurren en el sistema.
- c) Un ejemplo de esta programación es Léxico.
- d) Es la base de la Interfaz de usuario (UI).
- e) Todas son ciertas.
- f) Todas son falsas.

7 Gambas es un lenguaje de programación libre derivado de:
- a) Fortran.
- b) Basic.
- c) C++.
- d) C#.
- e) Java.
- f) Gambas no es un lenguaje de programación.

8 Glade es un lenguaje de programación libre derivado de:

- **a)** Fortran.
- **b)** Basic.
- **c)** C++.
- **d)** C#.
- **e)** Java.
- **f)** Glade no es un lenguaje de programación.

9 En informática, una Biblioteca:

- **a)** Es conjunto de implementaciones funcionales.
- **b)** Está codificada en un lenguaje de programación.
- **c)** Ofrece una interfaz bien definida para la funcionalidad que se crea.
- **d)** Todas son ciertas
- **e)** Todas con falsas.

10 En Java existen unas bibliotecas que permiten desarrollar interfaces de usuario gráficas, estas bibliotecas se conocen como:

- **a)** AWT.
- **b)** SWING.
- **c)** DirectX.
- **d)** OpenGL.
- **e)** A y b son ciertas.
- **f)** A y c son ciertas.
- **g)** A y d son ciertas.

11 Es una colección de API desarrolladas para facilitar las complejas tareas relacionadas con multimedia, especialmente programación de juegos y vídeo, en la plataforma Microsoft Windows:

- **a)** AWT.
- **b)** SWING.
- **c)** DirectX.
- **d)** OpenGL.
- **e)** A y b son ciertas.
- **f)** A y c son ciertas.
- **g)** A y d son ciertas.

12 Es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples.

- **a)** AWT.
- **b)** SWING.
- **c)** DirectX.
- **d)** OpenGL.
- **e)** A y b son ciertas.
- **f)** A y c son ciertas.
- **g)** A y d son ciertas.

13 Es una ventana cuadriculada (o no) sobre el que se disponen los componentes para diseñar las ventanas que formarán la aplicación:

- **a)** Explorador de archivos.
- **b)** Menú principal.
- **c)** Barra de herramientas.
- **d)** Diseñados de formularios.
- **e)** Todas son ciertas.

14 En ADO .NET puede crear y administrar conexiones utilizando objetos de conexión:

- **a)** SQLConnection.
- **b)** OleDbConnection.
- **c)** OdbcConnection.
- **d)** OracleConnection.
- **e)** Todas son ciertas.

15 Un cuadro de diálogo modal:

- **a)** Impide que el usuario continúe con la ejecución de la aplicación ignorando el cuadro.
- **b)** Se suele utilizar para mostrar información crítica.
- **c)** Se suele utilizar para advertir eventos peligrosos y acciones irreversibles.
- **d)** Todas son ciertas.
- **e)** Todas son falsas.

16 Un cuadro de diálogo no modal:

a) Impide que el usuario continúe con la ejecución de la aplicación ignorando el cuadro.

b) Se suele utilizar para mostrar información crítica.

c) Se suele utilizar para advertir eventos peligrosos y acciones irreversibles.

d) Todas son ciertas.

e) Todas son falsas.

17 Para usar un escuchador de eventos hemos de:

a) Implementar la interfaz del escuchador.

b) Registrar el escuchador en el objeto que genera el evento indicándole el objeto que lo recogerá.

c) Implementar los métodos *callback* correspondientes.

d) Todas son ciertas.

e) Todas son falsas.

18 Aplicación que genera código XML es:

a) Gambas.

b) Glade.

c) Embarcadero Delphi.

d) Microsoft Visual Studio.

e) Todas generan código XML.