PROGRAMACION ORIENTADA A OBJETOS

Luis Joyanes Aguilar
Director del Departamento de
Lenguajes y Sistemas Informáticos e Ingeniería de Software
Facultad de Informática
Universidad Pontificia de Salamanca *Campus* Madrid

McGraw-Hill

MADRID • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MEXICO
NUEVA YORK • PANAMA • SAN JUAN • SANTAFE DE BOGOTA • SANTIAGO • SAO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILAN • MONTREAL • NUEVA DELHI • PARIS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST LOUIS • TOKIO • TORONTO

PROGRAMACION ORIENTADA A OBJETOS

Edificio Valrealty, 1ª planta
Basauri, 17
28023 Aravaca (Madrid)

ISBN: 84-481-0590-7
Depósito legal: M. 30 121-1996

Editor: José Domínguez Alconchel
Diseño de cubierta: Juan García
Compuesto e impreso en Fernández Ciudad, S. L.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

CONTENIDO

Parte III
DISEÑO ORIENTADO A OBJETOS

Parte IV
EL LENGUAJE C++: SINTAXIS, CONSTRUCCION Y PUESTA A PUNTO DE PROGRAMAS

PROLOGO

LA ORIENTACION A OBJETOS

Las tecnologías orientadas a objetos se han convertido en la década de los noventa en uno de los motores clave de la industria del software. Sin embargo, las tecnologías de objetos no es, como algunos innovadores pregonan, una novísima tecnología, sino que, muy al contrario, es una vieja y madura tecnología que se remonta a los años sesenta. De hecho, Simula, uno de los lenguajes orientados a objetos más antiguos, fue desarrollado en 1967.

El desarrollo de programas orientados a objetos es un enfoque diferente del mundo informático. Implica la creación de modelos del mundo real y la construcción de programas informáticos basados en esos modelos. El proceso completo de programación comienza con la construcción de un modelo del suceso (evento) real. El resultado final del proceso es un programa de computadora que contiene características que representan algunos de los objetos del mundo real que son parte del suceso.

El principio básico de la programación orientada a objetos es que un sistema de software se ve como una secuencia de «transformaciones» en un conjunto de objetos. El término **objeto** tiene el mismo significado que un nombre o una frase nominal. Es una persona, un lugar o una cosa. Ejemplos de objetos del mundo real son: persona, tabla, computadora, avión, vuelo de avión, diccionario, ciudad o la capa de ozono. La mayoría de los objetos del mundo real tienen **atributos** (características que los describen). Por ejemplo, los atributos de una persona incluyen el nombre, la edad, el sexo, la fecha de nacimiento, la dirección, etc. Los objetos tienen atributos, y ellos, a su vez, comportamiento. El **comportamiento** (*behavior*) es el conjunto de cosas que puede hacer un objeto; por ejemplo, una persona puede estudiar, caminar, trabajar, etc. En síntesis, se puede decir que los objetos *conocen* cosas y *hacen* cosas. Las cosas que un objeto conoce son sus atributos; las cosas que puede hacer un objeto son su comportamiento.

Los principios en que se apoyan las tecnologías orientadas a objetos son:

- Objetos como instancia de una clase
- Métodos
- Mensajes.

Y las características que ayudan a definir un objeto son:

- Encapsulamiento.
- Modularidad
- Abstracción.
- Polimorfismo

Las clases se organizan para modelar el mundo real en las siguientes relaciones:

- Herencia (generalización/especialización).
- Agregación.
- Asociación.
- Uso

TIPOS ABSTRACTOS DE DATOS Y CLASES

Una **clase** es una caracterización abstracta de un conjunto de objetos; todos los objetos similares pertenecen a una clase determinada. Por ejemplo, un conjunto de objetos tales como cuadrados, triángulos, círculos, líneas, etc., pertenecen a una clase *figura*. De modo más formal, una clase define *variables* (datos) y *métodos* (operaciones) comunes a un conjunto de objetos. En realidad, una clase es un prototipo o generador de un conjunto de objetos.

Una clase bien diseñada especifica un **tipo abstracto de dato (TAD)**. Un tipo de dato es abstracto si las operaciones de alto nivel adecuadas a los tipos de datos están aisladas de los detalles de la implementación asociados con el tipo de datos. Así, por ejemplo, si diseñamos una clase *círculo* que convierte a un círculo en un tipo abstracto de dato, la clase nos proporciona métodos (funciones) tales como dibujar, mover, ampliar, contraer, borrar, etc. Se pueden utilizar estos métodos para manipular objetos *círculo* de todas las formas esperadas. Los métodos son todo lo que se necesita conocer sobre la clase *círculo*. Una estructura de datos fundamental de un círculo puede ser un array, un registro, una cadena de caracteres, etc. Los detalles de la representación interna de un círculo se pueden ignorar mientras se crean, amplían o mueven círculos. Un círculo como tipo abstracto de dato se centra exclusivamente en operaciones (métodos) apropiadas a los círculos; un *círculo* como tipo abstracto de dato ignora totalmente la representación interna de un círculo.

La **clase** es el bloque de construcción fundamental de un lenguaje de programación orientada a objetos. Una clase es un tipo abstracto de datos junto con un conjunto de transformaciones permitidas de dicho tipo abstracto de datos; puede definir también su interfaz a otras clases o funciones, descubriendo para ello que parte de su descripción interna de datos o conjunto de transformaciones permitidas pueden hacerse públicos. La regla por defecto es que *nada* de una clase es pública, a menos que se declare explícitamente por el desarrollador de software que definió la clase.

Aunque no es completamente una terminología estándar en POO, el término objeto se utiliza normalmente para representar una instancia de una *clase*. Otra visión de la clase es que un tiempo de ejecución tiene un estado, un comportamiento y una identidad.

El entorno orientado a objetos oculta los detalles de implementación de un objeto. Es la propiedad conocida como **ocultación de la información**. La parte que no está oculta de un objeto es su **interfaz público**, que consta de los mensajes que se pueden enviar al objeto. Los mensajes representan operaciones de alto nivel, tales como *dibujar* un círculo. El término **encapsulamiento** se utiliza también para enfatizar un aspecto específico de un tipo abstracto de datos. Un TAD combina métodos (*operaciones*) y representación interna (*implementación*).

Un objeto es una *instancia* —*ejemplar* o *caso*— de una clase que encapsula operaciones y representación. Este encapsulamiento contrasta con la separación tradicional de operaciones (funciones) y representación (datos). La clase en C++ y el paquete en Ada-95 soportan el encapsulamiento.

ANALISIS Y DISEÑO ORIENTADO A OBJETOS

El problema fundamental que debe asumir un equipo de desarrollo de software es convertir el mundo real en un programa informático. En esencia, la tarea clave de la programación es describir las tareas de especificación del programa que resuelve el problema dado.

Un problema de programación se describe normalmente con un conjunto de **especificaciones** (detalles que constituyen el problema real). Las especificaciones son parte de lo que se denomina **análisis orientado a objetos (AOO)**, que responde en realidad a la pregunta «¿Qué hace?» Durante la fase de análisis se piensa en las especificaciones en términos intuitivos y con independencia del lenguaje y de la máquina. La etapa crítica de esta actividad es deducir los tipos de objetos del mundo real que están implicados y obtener los atributos de estos objetos determinando su comportamiento e interacciones.

La siguiente fase del proceso de desarrollo de software es el **diseño orientado a objetos (DOO)**, que responde a la pregunta «¿Cómo lo hace?» Durante esta fase se comienza a crear un modelo de computadora basado en el análisis que realice la tarea específica concreta. En esta etapa se piensa en objetos del mundo real que pueden ser representados como objetos del mundo informático. Se deben especificar los objetos con mayor precisión especificando en detalle lo que los objetos conocen y lo que pueden hacer, y describe con prudencia sus interacciones. Durante la fase de diseño se pueden encontrar atributos útiles adicionales y comportamiento de los objetos que no aparecieron en la fase de análisis o no estaban definidos con claridad.

La diferencia entre AOO y DOO no es clara, y es difícil definir la transición entre ambas etapas. De hecho, ninguna de las metodologías de OO clásicas, como Yourdon/Coad, Booch o Rumbaugh (OMT) proporcionan reglas precisas para pasar de una etapa a otra. De hecho, las fases AOO y DOO no representan un proceso estricto de dos etapas, y a veces se funden en una sola. Normalmente, ocurrirá que el modelo inicial que se selecciona no es el apro-

piado, y se necesita retroceder y volver a reiterar el proceso sucesivamente Se pueden descubrir especificaciones adicionales que no se conocían al comenzar su trabajo inicial y encontrar que los atributos o comportamiento de un objeto sean diferentes de lo que se decidió en la etapa de análisis. De cualquier forma, el mejor medio para practicar desarrollo de software orientado a objetos es realizar el análisis y diseño de ejemplos de todo tipo Por esta causa, en el libro se incluyen numerosos ejemplos que tratan de ayudar al lector a familiarizarse con la POO.

La fase de diseño conduce a la fase de implementación, que consiste en traducir dicho diseño en un código real en un lenguaje de programación OO La fase de codificación del proceso de desarrollo OO se llama **programación orientada a objetos (POO).**

El proceso de desarrollo orientado a objetos supone, en síntesis, la construcción de un modelo del mundo real que se pueda traducir posteriormente en un código real escrito en un lenguaje de programación OO. En realidad, las tres fases, análisis, diseño y programación, interactúan entre sí. Las decisiones de programación pueden cambiar algunos aspectos del modelo o pueden refinar realmente algunas decisiones anteriores.

Los objetos pueden cambiar, o incluso modificarse o deducirse de otros objetos; atributos y comportamiento se pueden también modificar o añadir a cada objeto. En resumen, el análisis, diseño y programación no constituyen un proceso único de tres etapas para la resolución de un problema, sino que todas las etapas interactúan entre sí para resolver los problemas del mundo real. Sin embargo, como regla general, el análisis se debe hacer antes del diseño, y éste se ha de hacer antes de la programación o codificación.

NOTACIONES ORIENTADAS A OBJETOS

El mejor sistema para modelar el mundo real con objetos de un modo práctico es disponer de una notación gráfica consistente y eficiente. Cada metodología de análisis y diseño orientado a objetos posee su propia notación.

Nuestra experiencia en estos cinco últimos años impartiendo cursos de AOO y DOO a estudiantes de pregrado, postgrado y profesionales nos ha llevado a seleccionar las notaciones que personalmente hemos comprobado que son las más idóneas, tanto desde el punto de vista pedagógico como profesional. Pensando en un aprendizaje rápido y gradual, hemos seleccionado tres metodologías de las más populares:

- Coad/Yourdon.
- Booch'93
- OMT (Rumbaugh *et al*)

Junto con otras dos notaciones, que si bien no son tan conocidas, a nosotros nos han resultado de gran valor y podemos considerarlas excelentes para el aprendizaje de objetos. Son las notaciones de Raimund K. Ege y David Taylor, que hemos incluido en el texto y con ejemplos inspirados en sus textos

base, que se recogen en el momento oportuno y en la bibliografía, y que recomendamos como lecturas notables y excelentes, así como referencia obligada de todo buen estudioso de las tecnologías de objetos

PROGRAMACION ORIENTADA A OBJETOS

La programación orientada a objetos es una extensión natural de la actual tecnología de programación, y representa un enfoque nuevo y distinto al tradicional Al igual que cualquier otro programa, el diseño de un programa orientado a objetos tiene lugar durante la fase de diseño del ciclo de vida de desarrollo de software. El diseño de un programa OO es único en el sentido de que se organiza en función de los objetos que manipulará De hecho, probablemente la parte más difícil de la creación de software orientado a objetos es identificar las clases necesarias y el modo en que interactúan entre sí

Desgraciadamente, no hay reglas fáciles para determinar las clases de un programa dado. La identificación de clases puede ser tanto arte como ciencia El proceso es algo impreciso, y por esta causa han surgido numerosos métodos que proporcionan reglas para la identificación de clases y las relaciones que existen entre ellas; estos métodos son los citados anteriormente.

EL LENGUAJE C++

C++ todavía no es un lenguaje estándar, aunque ya se encuentra en la fase final de estandarización. C++ es sin duda el lenguaje del futuro, y marca las pautas de desarrollo para nuevos lenguajes, como es el caso de **Java,** el lenguaje de programación orientado a objetos para desarrollo en Internet, o Ada-95, el lenguaje de desarrollo para sistemas en tiempo real también orientado a objetos

Las características comunes más importantes a las nuevas versiones de C++ son:

- C++ es esencialmente un superconjunto de C ANSI.
- C++ tiene las mismas características de tipificación que C ANSI para propiedades no OO.
- Los compiladores de C++ aceptan normalmente código escrito en la versión original de K&R (Kernighan y Ritchie) Generalmente, los compiladores de C++ proporcionan mensajes de error o advertencia cuando el código C no tiene prototipos
- Desde el punto de vista específico de sintaxis, algunas características de C han sido mejoradas notablemente:

  1 Las funciones de entrada/salida `printf` y `scanf` se utilizan raramente en C++, y en su lugar se emplean `cin` y `cout`, que realizan un trabajo mejor y más eficiente

2. Las constantes `#define` y las macros han sido sustituidas por el calificador `const` y las funciones `inline`
3. Identificadores de tipos en tiempo de ejecución.
4. Espacios de nombre

Las principales diferencias entre los diversos compiladores de C++ son, además del precio, entornos integrados de desarrollo (editores, depuradores, etc.), velocidad de compilación, velocidad del código ejecutable, sistema en tiempo de ejecución, calidad de mensajes de error e interoperabilidad de código con otro software, tales como sistemas operativos, sistemas de ventana, enlazadores u otros programas de aplicación.

Otras diferencias incluyen soporte para manejadores de excepciones y plantillas (*templates*) La mayoría de los compiladores actuales proporcionan soporte para ambas propiedades Los manejadores (*handlers*) de excepciones son construcciones que permiten a los programas recuperar su control ante errores en tiempo de ejecución no previstos Las plantillas permitirán a las clases ser definidas mediante tipos genéricos de datos

HISTORIA DEL LENGUAJE C++

Al principio de los ochenta, Bjarne Stroustrup diseñó una extensión del lenguaje C al que llamó *C con clases*, debido a que su característica fundamental era añadirle clases a C El concepto de clase procedía de Simula 67 y servía para capturar el comportamiento del mundo real a la vez que oculta los detalles de su implementación

En 1983-84, *C con clases* fue rediseñado, extendido e implementado en un compilador. El lenguaje se denominó C++ y fue descrito por Stroustrup en *Data Abstraction in C* en el *Technical Journal* (vol. 63, núm. 8, octubre 1984), de AT&T Bell Laboratories. La primera versión comercial de C++ estuvo disponible en 1985 y se documentó en el libro de Bjarne Stroustrup *The C++ Programming Language*, editado por Addison-Wesley en 1986.

El nombre de C++ fue elegido como variante del lenguaje de programación C. Dado que era una extensión de C, se decidió elegir C++ debido a que el operador ++ significa «añadir uno a la variable» y por consiguiente el lenguaje C++ se supondría que era una versión inmediatamente superior o siguiente a C.

En realidad, Stroustrup creó C++ con dos objetivos principales: (1) hacer compatible C++ con el C ordinario, y (2) ampliar C con construcciones de POO basadas en la construcción *clase* de Simula 67. El lenguaje en su forma actual ha sido descrito en 1990 por Stroustrup y Ellis en el *Annotated C++ Reference Manual* (el ARM)[1], que sirve como documento base para la estandarización de la versión 3.0, actualmente en fase de estandarización por el comité ANSI.

Como C++ es una extensión del C estándar, la mayoría de los programas C se pueden compilar utilizando un compilador C++.

La versión actual estandarizada por ANSI —la citada actualización 3.0— es la que soportan la mayoría de los fabricantes mundiales: Borland, Microsoft, Watcom, AT&T, etc., en sus últimas actualizaciones tales como 4.5/5 de Borland, Visual C++ 4, la 10 de Watcom, etc

[1] MARGARET ELLIS y BJARNE STROUSTRUP: *The Annotated C++ Reference Manual* Reading, Mass, Addison-Wesley, 1990 Existe versión en español con el título *C++ Manual de referencia con anotaciones* editado por Addison-Wesley/Díaz de Santos en 1994

OBJETIVOS DEL LIBRO

Programación Orientada a Objetos Conceptos, modelado, diseño y codificación en C++, es una obra, como su nombre indica, esencialmente de **objetos.** Trata fundamentalmente de enseñar a programar con tecnologías de objetos, pero al contrario que otras obras —incluso algunas nuestras— centradas exclusivamente en un lenguaje de programación orientada a objetos —casi siempre C++—, hemos pensado que sería muy interesante —sin abandonar el estudio de C++— dar mayor importancia a los conceptos teóricos y prácticos fundamentales del mundo de objetos, que sólo son tratados por libros específicos de las metodologías de AOO/DOO e incluso ingeniería de software OO y otros de carácter avanzado.

Desde finales de los ochenta, en que decidimos adentrarnos en las emergentes tecnologías de objetos, se ha producido un cambio radical en el mundo de la ingeniería de software. Los objetos que los años 1988 a 1990 se centraban en experiencias sobre el primitivo lenguaje C++ y Smalltalk e incipientes trabajos en Eiffel, comenzaron a pasar en los años 1990 a 1992 al campo profesional, y así nacieron las metodologías de análisis y diseño orientadas a objetos de primera generación y que han facilitado la transición al nuevo paradigma. Las metodologías pioneras se deben a Shlaer y Mellor, que, junto con Rebeca WirkBrocks, se consideran como creadores del modelado de objetos; Yourdon/Coad, autores consolidados de metodologías estructuradas que se pasaron a metodologías de objetos, y lanzan en esos años dos excelentes libros sobre análisis y diseño OO; Grady Booch, uno de los pioneros del mundo de ingeniería de software en tiempo real con Ada, que aprovecha su experiencia para lanzar una metodología de diseño OO que se ha hecho muy popular y cuya última edición se publicó en 1993; Rumbaugh *et al*, autores de la metodología OMT, seguramente la metodología más utilizada en el mundo del software en estos últimos años.

La siguiente frontera se produce en 1995, cuando se publica el borrador 0.8 del método unificado diseñado por la unión en una misma empresa (Rational), de Grady Booch y James Rumbaugh, que han contado también con la colaboración de Ivar Jacobson, creador del *use case* —caso de uso—, concepto teórico fundamental que se ha impuesto en todos los buenos desarrollos de OO de los últimos años. Desde el punto de vista del lenguaje se está estandarizando C++ con la versión 3.0 de AT&T y se ha estandarizado Ada con la

versión Ada-95, y se han lanzado al mercado otros lenguajes de objetos híbridos: Object Pascal, Object COBOL, Delphi, Visual BASIC 4 —con ciertas características de objetos—, Visual Object, etc. Los lenguajes puros clásicos como Eiffel y Smalltalk luchan por hacerse un hueco en el mercado profesional, saliendo de los laboratorios de investigación y universitarios hacia el mundo profesional. Internet, el fenómeno social y tecnológico de la década de los noventa y del futuro siglo XXI, ha traído el advenimiento de Java un lenguaje OO evolucionado de C++ que la empresa Sum lanzó el año pasado y que promete convertirse en un duro competidor de C++.

Teniendo presente el estado del arte de la ingeniería de software orientada a objetos y nuestra experiencia personal en el Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software de la Universidad Pontificia de Salamanca en Madrid, donde hemos impartido numerosos cursos de programación, análisis y diseño orientados a objetos, así como otros cursos, seminarios y conferencias en otras universidades españolas y latinoamericanas y en empresas informáticas multinacionales y escuelas de la administración pública española, hemos considerado los numerosos consejos, sugerencias y críticas de alumnos y colegas, y como resultado hemos escrito un contenido para esta obra que contiene los conceptos vitales de las tecnologías de objetos que confiamos permitan una progresión y aprendizaje rápido y eficiente por parte del lector en el mundo de la programación orientada a objetos. Para cumplir estos objetivos, pensamos que sería muy interesante mezclar con el máximo de prudencia conceptos fundamentales tales como:

- *Tipos abstractos de datos.*
- Clases y objetos.
- Relaciones de objetos: generalización/especialización.
- Herencia
- Modelado de objetos.
- Diseño orientado a objetos.
- Fundamentos de reutilización de software con objetos.
- Bibliotecas de clases.

Así pues, la obra considera los conceptos teórico-prácticos importantes de la **programación orientada a objetos**, junto con los métodos correspondientes de codificación en C++. El contenido del libro se ha diseñado de modo que pueda permitir al lector ya iniciado en objetos y/o C++, y al no iniciado adentrarse en el mundo de los objetos de un modo gradual y con la mayor eficacia posible. Para ello se ha pensado que el mejor método podría ser enseñar al lector las técnicas de modelado del mundo real mediante objetos, de modo que cuando se tuviera el modelo idóneo se pudiera pasar fácilmente a la codificación de un programa que resolviera el problema en cuestión. Para conseguir estos objetivos hemos creído conveniente hacer uso no sólo de los conceptos teóricos ya mencionados, sino recurrir a una herramienta gráfica que facilite al lector realizar el análisis y diseño previo a la programación que redunde en el mayor grado de eficiencia por parte del programador.

CONTENIDO DEL LIBRO

Este libro se apoya fundamentalmente en el emergente paradigma de la orientación a objetos y trata de enseñarle sus conceptos básicos, así como las técnicas de programación de dicho paradigma. Supone que el lector tiene experiencia anterior en programación en algún lenguaje, tal como BASIC, C o Pascal; también supone que el lector tiene experiencia en editar, compilar, enlazar y ejecutar programas en su computadora. De cualquier forma, pensando en los lectores que no conocen C ni C++, hemos incluido un apéndice que contiene una guía de referencia del lenguaje C++, junto con una parte completa (Parte IV) que incluye tres capítulos que pretenden enseñar al lector la transición de C a C++, o simplemente el lenguaje C++, caso de no conocer C/C++, así como reglas prácticas para poner a punto programas en C++, con una amplia relación de errores típicos cometidos en programas.

El libro consta de cuatro partes que contienen quince capítulos, todos ellos con una estructura muy similar: *teoría y ejemplos prácticos* desarrollados en la versión 3.0 de C++ de ANSI, de modo que prácticamente podrá utilizar con cualquier compilador de los comercializados en la actualidad de las empresas Borland, Microsoft, Watcom, etc; un *resumen* del capítulo y *ejercicios propuestos* al lector, de modo que pueda practicar los conceptos aprendidos en el capítulo correspondiente.

La Parte I, «El mundo de la orientación a objetos», describe los conceptos fundamentales de objetos, relaciones, modelado y **lenguajes de programación orientada a objetos (LPOO)**. El Capítulo 1 ofrece una visión general del desarrollo del software, con una revisión de los conceptos clave del mismo, que abarcan desde los factores de calidad del software a la *reutilización* de software, pasando por conceptos clave, como abstracción de datos, encapsulamiento, jerarquía y polimorfismo, entre otros. El Capítulo 2 es una revisión del importante concepto de modularidad y su pieza clave los tipos abstractos de datos. Se han utilizado como herramientas de programación los lenguajes Modula-2, Ada, Turbo Pascal, C y C++. El Capítulo 3 es una descripción exhaustiva de los conceptos fundamentales de la programación orientada a objetos (clases, objetos, lenguajes, herencia, sobrecarga, ligadura, objetos compuestos y reutilización). El Capítulo 4 describe los lenguajes de POO, los clasifica y realiza una síntesis de las propiedades orientadas a objetos de los lenguajes seleccionados, en este caso Ada, Eiffel y Smalltalk. El Capítulo 5 trata el importante concepto de *modelado*. El proceso de desarrollo de un sistema de software comienza con la construcción de un modelo del mundo real. Este modelo captura normalmente las características más significativas del problema, y para ello se apoya en el concepto de relaciones entre clases. Las diferentes relaciones junto con la importante propiedad de la herencia y sus tipos, se describen también en el Capítulo 5.

La Parte II incluye los Capítulos 6 al 11 y explica los fundamentos de la **programación orientada a objetos (POO)** con C++. El Capítulo 6 describe cómo declarar y construir clases, diseños y reglas prácticas para la construcción de clases. El Capítulo 7 examina las clases abstractas y la propiedad de la herencia, junto con la sintaxis para su implementación y el problema de la

herencia repetida; el capítulo se termina con una aplicación práctica. El Capítulo 8 presenta la propiedad de polimorfismo, junto con el concepto de ligadura. El Capítulo 9 describe las **plantillas** (*templates*) y muestra el concepto de **genericidad**; examina la sintaxis para declarar plantillas de funciones y de clases, así como la definición de sus funciones miembro y el modo de *instanciar* las clases. El Capítulo 10 presenta el concepto de **excepción**, junto con su manejo o manipulación (errores en tiempo de ejecución), y examina el método empleado por C++ para lanzar y capturar excepciones; se describe la sintaxis de C++ para implementar estas operaciones y muestra cómo manejar excepciones. El Capítulo 11 describe la reutilización de software con C++ y los diferentes métodos empleados para ello. Asimismo, se describen las bibliotecas y contenedores de clases.

La Parte III incluye el transcendental Capítulo 12, que describe los principios para el desarrollo orientado a objetos y especialmente su diseño. Se describen las notaciones gráficas de las metodologías Booch, Yourdon/Coad y Rumbaugh (OMT), junto con las reglas prácticas para la implementación con C++ de las diferentes relaciones entre clases; se incluye una pequeña aplicación orientada a objetos. La Parte IV «El lenguaje C++: Sintaxis, construcción y puesta apunto de programas» contiene los Capítulos 13 a 15. El Capítulo 13 describe las características más sobresalientes de C++ que lo diferencian de C en el sentido de mejorarlo y ampliarlo. El Capítulo 14 explica un sistema práctico para construir programas en C/C++ junto con el concepto de programas multiarchivos y el sistema para construir archivos proyecto. La puesta a punto de programas en C++ se explica en el Capítulo 14; en este capítulo se dan reglas prácticas para depurar programas con una extensa enumeración de errores típicos en el desarrollo de programas.

Aunque el lenguaje base del texto es C++ se pretende que el lector pueda codificar fácilmente los conceptos fundamentales de objetos y comenzar el aprendizaje de la POO con otros lenguajes. Por esta causa los Apéndices A a E incluyen guías de referencias de sintaxis de los lenguajes C++, Delphi, Turbo/Borland Pascal, Ada-95 y Java —el nuevo lenguaje orientado a objetos de *Internet*— El Apéndice F explica un concepto importante y específico de C++: *la sobrecarga de operadores*. La serie de apéndices se completa con el Apéndice G, que contiene síntesis de las notaciones gráficas de las metodologías de análisis y diseño orientadas a objetos más populares y usadas en el libro. Estas notaciones se han extraído de las fuentes originales de las metodologías utilizadas: Booch'93, OMT (Rumbaugh *et al*) y Coad/Yourdon. Por último, se incluye un glosario de términos de objetos que facilitan la comprensión del lector. La bibliografía contiene los libros consultados por el autor en la escritura de la obra, incluyendo los libros base de las metodologías de AOO y DOO utilizadas en el libro.

AGRADECIMIENTOS

Muchas son las personas que me han prestado ayuda de una u otra forma en la elaboración de esta obra y a las que debo mi agradecimiento más sincero.

En particular, deseo expresar mi reconocimiento a mis colegas del Departamento de Lenguajes y Sistemas Informáticos de la Facultad y Escuela Universitaria de Informática de la Universidad Pontificia de Salamanca, en el *campus* de Madrid, que han impartido e imparten conmigo ***Programación Orientada a Objetos*** dentro de la asignatura **Metodología de la Programación**, así como **Análisis y Diseño Orientados a Objetos**, cuyas sugerencias, consejos y críticas han permitido los apuntes originales de la asignatura en este libro que hoy ve la luz. Estos profesores son: Ignacio Zahonero, Antonio Reus, Paloma Centenera, Rosa Hernández, Rafael Ojeda, Fernando Davara, M.ª Luisa Díez, Daniel García y Luis Doreste. De un modo especial han contribuido a esta obra las profesoras Paloma Centenera, María Luisa Díez y Rosa Hernández, que han volcado su experiencia docente personal en la revisión de las galeradas de esta obra; tras su lectura han detectado erratas y sobre todo me han dado consejos, sugerencias y aportaciones personales que han permitido mejorar la versión original de esta obra. Gracias, amigas y colegas, por vuestra ayuda. También el profesor Ignacio Zahonero ha leído parte de las galeradas y me ha sugerido ideas de mejora. Aunque no han intervenido directamente en la obra hay numerosísimas personas que han contribuido eficientemente en la redacción final de esta obra, sin su colaboración esta obra no estaría ahora en la calle. Son todos mis alumnos de los últimos cinco años a los que he impartido cursos, seminarios y conferencias sobre tecnologías orientadas a objetos (análisis, diseño, programación y bases de datos) en universidades españolas y latinoamericanas, así como en centros de formación de empresas multinacionales y de la administración española. Ellos me han alentado en todo momento a difundir las tecnologías de objetos y de ellos he recibido todo tipo de críticas, consejos y sugerencias que he volcado en muchos casos en el libro. Por último, deseo expresar de modo muy especial mi agradecimiento a Jorge Piernavieja, mi antiguo editor —y, sin embargo, *amigo*— que inició la edición de esta obra y que por sus nuevas responsabilidades profesionales no la ha podido terminar, pero su consejo y aliento al igual que tantas otras veces nunca me faltaron. A mi nuevo editor y también amigo Pepe Domínguez que ha terminado la obra; mi agradecimiento por su paciencia y comprensión. Al lector que ha confiado en esta obra en la esperanza de que le sea lo más útil y eficaz posible en su formación en programación orientada a objetos.

En Carchelejo (Andalucía-España), verano de 1996
El autor

PARTE I

EL MUNDO DE LA ORIENTACION A OBJETOS: CONCEPTOS, RELACIONES, MODELADO Y LENGUAJES DE PROGRAMACION

CAPITULO 1

EL DESARROLLO DE SOFTWARE

CONTENIDO

La década de los noventa será, sin lugar a dudas, la década de la programación orientada a objetos. Como Rentsch predijo, *«la programación orientada a objetos será en los ochenta lo que la programación estructurada fue en la década de los setenta»*. En la actualidad la programación orientada a objetos se ha hecho enormemente popular. Escritores y diseñadores de software, junto a compañías importantes en el campo del software, se dedican de modo continuo a producir compiladores de lenguajes, sistemas operativos, bases de datos, etc., orientados a objetos.

¿Qué es la programación orientada a objetos? ¿Por qué es tan popular? La programación orientada a objetos es algo más que una colección de lenguajes de programación, tales como Smalltalk, Object Pascal, C++, etc. Se podría decir que este tipo de programación es un nuevo medio de pensar sobre lo que significa computar (computadorizar), es decir, cómo se puede estructurar información en un computador.

1.1. LA COMPLEJIDAD INHERENTE AL SOFTWARE

Como Brooks sugiere, «la complejidad del software es una propiedad esencial, no accidental». Esta complejidad inherente al software, como dice Booch, se deriva de cuatro elementos: la complejidad del dominio del problema, la dificultad de gestionar el proceso de desarrollo, la posible flexibilidad a través del software y los problemas de caracterización del comportamiento de sistemas discretos.

1.1.1. La complejidad del dominio del problema

Los problemas que se intentan resolver con software implican normalmente elementos de ineludible complejidad, en los que se encuentran una gran cantidad de requisitos, en muchas ocasiones contradictorios. Esta complejidad se produce por las difíciles interacciones entre los usuarios de un sistema y sus desarrolladores: los usuarios encuentran generalmente muy difícil dar precisión sobre sus necesidades de forma que los desarrolladores puedan comprender. En casos extremos, los usuarios pueden tener sólo ideas vagas de lo que se desea en un sistema software.

Por otra parte, los usuarios y desarrolladores tienen diferentes perspectivas de la naturaleza del problema y hacen suposiciones diferentes sobre la naturaleza de la solución. El medio común de expresar los requisitos hoy día es utilizar un gran volumen de textos, en ocasiones acompañados por esquemas y dibujos. Tales documentos son difíciles de comprender, están abiertos a diferentes interpretaciones y con frecuencia contienen elementos que son diseños en lugar de requisitos esenciales.

Otra complicación frecuente es que los requisitos de un sistema software cambian durante su desarrollo. Esto supone que un sistema grande tiende a evolucionar con el tiempo y el mantenimiento del software en ocasiones es un término que no siempre está bien acuñado.

Para ser más preciso, existen diferentes términos a definir: el mantenimiento busca errores; la evolución responde a cambios de requisitos, y la conservación, cuando se utilizan medios para mantener piezas de software en funcionamiento. Desgraciadamente, la realidad sugiere que un porcentaje alto de los recursos de desarrollo de software se gastan en la conservación del software.

1.1.2. La dificultad de gestionar el proceso de desarrollo

El tamaño de un programa no es una gran virtud en un sistema de software. Sin embargo, la escritura de un gran programa requiere la escritura de grandes cantidades de nuevo software y la reutilización del software existente. Recordemos que hace dos o tres décadas los programas en lenguaje emsamblador se construían a base de centenares de líneas. Sin embargo, hoy es usual encontrar sistemas en funcionamiento cuyo tamaño se mide en centenares de millares, o

incluso millones de líneas de código. Esta característica se facilita descomponiendo nuestra implementación en centenares y a veces millones de módulos independientes. Esta cantidad de trabajo exige el uso de un equipo de desarrolladores, aunque se trata por todos los medios de que este equipo sea lo más pequeño posible. Ahora bien, a medida que haya más desarrolladores, se producen comunicaciones entre ellos más complejas, e incluso con coordinación difícil entre ellos, particularmente si el equipo está disperso geográficamente, como suele ser el caso de proyectos grandes.

La rotura de una aplicación en entidades y relaciones que son significativas a los usuarios es un análisis convencional y técnicas de diseño. Con la programación orientada a objetos, este proceso de descomposición se extiende a la fase de implementación. Es más fácil diseñar e implementar aplicaciones orientadas a objetos, ya que los objetos en el dominio de la aplicación se corresponden directamente con los objetos en el dominio del software.

1.1.3. La flexibilidad a través del software

El software ofrece flexibilidad, de modo que es posible para un desarrollador expresar prácticamente cualquier clase de abstracción.

Los sistemas orientados a objetos proporcionan el rendimiento, la flexibilidad y funcionalidad requerida para implementaciones prácticas. La programación se puede hacer con extensiones de lenguajes comerciales, tales como Object-Pascal (Turbo Pascal, Borland Pascal, Mac Pascal, etc.) y C++, que incorporan a sus típicas propiedades las propiedades orientadas a objetos, y lenguajes OO puros, como Smalltalk y Eiffel. Por otra parte, las ayudas de programación actual mejoran la capacidad del programador para administrar y modificar sistemas mientras se desarrollan.

La programación orientada a objetos expande también la variedad de aplicaciones que se pueden programar, debido a que se liberan las restricciones de los tipos de datos predefinidos.

La programación orientada a objetos acomoda estructuras de datos heterogéneos y complejos. Se pueden añadir nuevos tipos de datos sin modificar código existente.

1.2. LA CRISIS DEL SOFTWARE

En 1968 una conferencia sobre software, patrocinada por la OTAN, asumió los términos *ingeniería del software* y *crisis del software*. Con estos términos se quería expresar que el software era caro, poco fiable y escaso.

Las metodologías y técnicas estructurales que han reinado en la década de los setenta y ochenta no han eliminado el problema, y de hecho la crisis del software continúa hoy en día. Pese a las muchas herramientas y métodos utilizados, los problemas del diseño descendentes permanecen igual, posiblemente debido a que la complejidad del problema ha crecido considerablemente.

Entre las diferentes fases del ciclo de vida del software (Fig. 1.1), el mantenimiento, aunque en tiempos fue despreciada su importancia, se considera actualmente como uno de los problemas más rigurosos en el desarrollo del software.

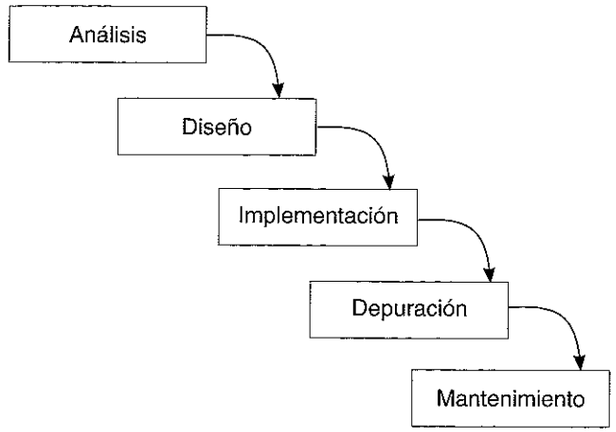


Figura 1.1. Ciclo de vida del software.

Muchos investigadores sugieren que los costes de software requieren más de la mitad de los costes y recursos globales en el desarrollo de software.

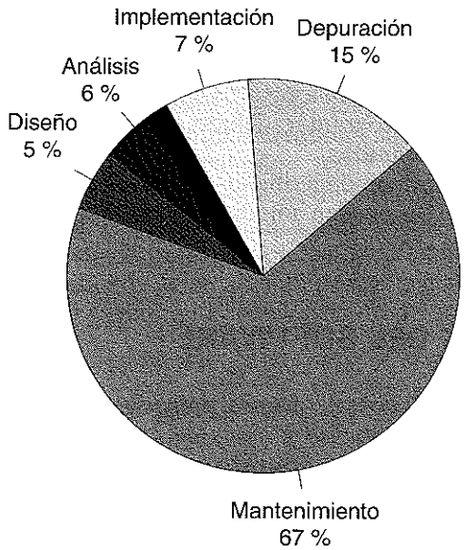


Figura 1.2. Costes de las diferentes fases del ciclo de vida de un proyecto software.

Los cambios realizados en la evolución de un programa son el punto débil de los métodos tradicionales de desarrollo de software, siendo paradójicamente uno de los puntos fuertes de los métodos de desarrollo de software orientado a objetos.

En 1986, Fredrick P. Brooks[1], en un famoso artículo, apuntaba que en los últimos diez años no se había producido ningún progreso significativo en el desarrollo de software, y analizaba críticamente todas las tecnologías más prometedoras. Aunque él confesaba que tenía más confianza en la programación orientada a objetos que en cualquier otra tecnología, mantenía dudas sobre sus ventajas efectivas.

Recientemente, las propuestas de *reusabilidad* o *reutilización*, ***«reusability»***, *de componentes software*, se consideran como bloques iniciales para la construcción del programa, de modo similar a la construcción de cualquier objeto complejo (tal como un automóvil) que se construye ensamblando sus partes.

En respuesta al artículo de Brooks, Brad Cox[2], el inventor de Objective-C, publicó un artículo en el que esencialmente rebatía las tesis de Brooks:

> Existe una bala de plata. Es un arma tremendamente potente, impulsada por vastas fuerzas económicas a la que nuevos obstáculos técnicos sólo pueden resistir brevemente.
>
> La bala de plata es un *cambio cultural* en lugar de un cambio tecnológico. Es un nuevo paradigma; una revolución industrial basada en partes reutilizables e intercambiables que modificarán el universo del software, de igual modo que la revolución industrial cambió la fabricación.

Por consiguiente, la POO (Programación Orientada a Objetos) no sólo son nuevos lenguajes de programación, sino un nuevo modo de *pensar* y *diseñar* aplicaciones que pueden ayudar a resolver problemas que afectan al desarrollo del software. Sin embargo, el lenguaje debe ser capaz de soportar el nuevo paradigma, siendo por consiguiente una parte esencial de esta revolución.

1.3. FACTORES EN LA CALIDAD DEL SOFTWARE

La construcción de software requiere el cumplimiento de numerosas características. Entre ellas se destacan las siguientes:

Eficiencia

La eficiencia del software es su capacidad para hacer un buen uso de los recursos que manipula.

---

[1] Brooks, Frederik P. Jr.: «No Silver Bullet», *Computer*, 10-19, abril 1986.
[2] Cox, Brad J.: «There is a Silver Bullet», *Byte*, 209-218, octubre 1987.

Transportabilidad (portabilidad)

La transportabilidad o portabilidad es la facilidad con la que un software puede ser transportado sobre diferentes sistemas físicos o lógicos.

Verificabilidad

La verificabilidad es facilidad de verificación de un software; es su capacidad para soportar los procedimientos de validación y de aceptar juegos de test o ensayo de programas.

Integridad

La integridad es la capacidad de un software para proteger sus propios componentes contra los procesos que no tengan derecho de acceso.

Fácil de utilizar

Un software es fácil de utilizar si se puede comunicar con él de manera cómoda.

Corrección

Capacidad de los productos software de realizar exactamente las tareas definidas por su especificación

Robustez

Capacidad de los productos software de funcionar incluso en situaciones anormales.

Extensibilidad

Facilidad que tienen los productos de adaptarse a cambios en su especificación Existen dos principios fundamentales para conseguir esto:

- diseño simple;
- descentralización.

Reutilización

Capacidad de los productos para ser reutilizados, en su totalidad o en parte, en nuevas aplicaciones.

Compatibilidad

Facilidad de los productos para ser combinados con otros.

1.3.1. Razones fundamentales que están influyendo en la importancia de la POO

Algunas de las causas que están influyendo considerablemente en el notable desarrollo de las técnicas orientadas a objetos son:

- La OO (Orientación a Objetos) es especialmente adecuada para realizar determinadas aplicaciones, sobre todo realización de prototipos y simulación de programas.
- Los mecanismos de encapsulación de POO soportan un alto grado de reutilización de código, que se incrementa por sus mecanismos de herencia.
- En el entorno de las bases de datos, la OO se adjunta bien a los modelos semánticos de datos para solucionar las limitaciones de los modelos tradicionales, incluido el modelo relacional.
- Aumento espectacular de LPOO (Lenguajes de Programación Orientados a Objetos).
- Interfaces de usuario gráficos (por iconos) y visuales. Los interfaces de usuario de una aplicación manipulan la entrada y salida del usuario. Por consiguiente, su función principal es la comunicación con el usuario final. La entrada al sistema se puede controlar a través de líneas de órdenes (enfoque utilizado por DOS y UNIX), o alternativamente el usuario puede interactuar con el sistema, con construcciones de programación visuales, tales como iconos de menús, Windows, Macintosh, etc

Estas razones hacen fundamentalmente que dentro de las tendencias actuales de la ingeniería de software, el marco de POO se revela como el más adecuado para la elaboración del diseño y desarrollo de aplicaciones

Este marco se caracteriza por la utilización del diseño modular OO y la reutilización del software. Los TAD (Tipo Abstracto de Dato) han aumentado la capacidad para definir nuevos tipos (clases) de objetos, cuyo significado se definirá abstractamente, sin necesidad de especificar los detalles de implementación, tales como la estructura de datos a utilizar para la representación de los objetos definidos

Los objetos pasan a ser los elementos fundamentales en este nuevo marco, en detrimento de los subprogramas que lo han sido en los marcos tradicionales.

1.4. PROGRAMACION Y ABSTRACCION

Para comprender mejor el significado de la revolución que suponen las tecnologías orientas a objetos, se va a examinar uno de los elementos fundamentales: *programación por abstracción*

En general, un programa no es más que una descripción abstracta de un procedimiento o fenómeno que existe o sucede en el mundo real. Frecuentemente, un programa imita un comportamiento o acción humana; otras veces *simula* (es decir, lo reproduce) un fenómeno físico.

Sin embargo, la relación entre abstracción y lenguaje de programación es doble: por un lado se utiliza el lenguaje de programación para escribir un programa que es una abstracción del mundo real; por otro lado se utiliza el lenguaje de programación para describir de un modo abstracto el comportamiento físico de la computadora que se está utilizando (por ejemplo utilizando números decimales en lugar de números binarios, variables en lugar de celdas de memoria direccionadas explícitamente, etc.)

En la década de los cincuenta, el único mecanismo de abstracción era el lenguaje ensamblador y de máquina, que ofrecía la posibilidad de utilizar nombres simbólicos para representar celdas de memoria. Posteriormente, los lenguajes de alto nivel ofrecieron un nuevo nivel de abstracción

El *arte de la programación* es el método por el que se describirá a una computadora (mediante un lenguaje de programación) un fenómeno, una acción, un comportamiento o una idea

1.5. EL PAPEL (EL ROL) DE LA ABSTRACCION

Los programadores han tenido que luchar con el problema de la complejidad durante mucho tiempo desde el nacimiento de la informática. Para comprender lo mejor posible la importancia de las técnicas orientadas a objetos, revisemos cuáles han sido los diferentes mecanismos utilizados por los programadores para controlar la complejidad. Entre todos ellos destaca la *abstracción*. Como describe Wulft: «Los humanos hemos desarrollado una técnica excepcionalmente potente para tratar la complejidad: abstraernos de ella. Incapaces de dominar en su totalidad los objetos complejos, se ignora los detalles no esenciales, tratando en su lugar con el modelo ideal del objeto y centrándonos en el estudio de sus aspectos esenciales»

En esencia, *la abstracción es la capacidad para encapsular y aislar la información del diseño y ejecución*. En otro sentido, las técnicas orientadas a objetos se ve como resultado de una larga progresión histórica que comienza en los procedimientos y sigue en los módulos, tipos abstractos de datos y objetos.

1.5.1. La abstracción como proceso natural mental

Las personas normalmente comprenden el mundo construyendo modelos mentales de partes del mismo; tratan de comprender cosas con las que pueden interactuar: un modelo mental es una vista simplificada de cómo funciona de modo que se pueda interactuar contra ella. En esencia, este proceso de construcción de modelos es lo mismo que el diseño de software, aunque el desarrollo de software es único: el diseño de software produce el modelo que puede ser manipulado por una computadora

Sin embargo, los modelos mentales deben ser más sencillos que el sistema al cual imitan, o en caso contrario serán inútiles. Por ejemplo, consideremos un mapa como un modelo de su territorio. A fin de ser útil, el mapa debe ser más sencillo que el territorio que modela. Un mapa nos ayuda, ya que abstrae sólo

aquellas características del territorio que deseamos modelar. Un mapa de carreteras modela cómo conducir mejor de una posición a otra. Un mapa topográfico modela el contorno de un territorio, quizá para planear un sistema de largos paseos o caminatas.

De igual forma que un mapa debe ser más pequeño significativamente que su territorio e incluye sólo información seleccionada cuidadosamente, así los modelos mentales abstraen esas características de un sistema requerido para nuestra comprensión, mientras ignoran características irrelevantes. Este proceso de *abstracción* es psicológicamente necesario y natural: la abstracción es crucial para comprender este complejo mundo.

La abstracción es esencial para el funcionamiento de una mente humana normal y es una herramienta muy potente para tratar la complejidad. Considerar, por ejemplo, el ejercicio mental de memorizar números. Un total de siete dígitos se puede memorizar con más o menos facilidad. Sin embargo, si se agrupan y se denominan números de teléfono, los dígitos individuales se relegan en sus detalles de más bajo nivel, creándose un nivel abstracto y más alto, en el que los siete números se organizan en una única entidad. Utilizando este mecanismo se pueden memorizar algunos números de teléfonos, de modo que la agrupación de diferentes entidades conceptuales es un mecanismo potente al servicio de la abstracción.

1.5.2. Historia de la abstracción del software

La abstracción es la clave para diseñar buen software. En los primeros días de la informática, los programadores enviaban instrucciones binarias a una computadora, manipulando directamente interrupciones en sus paneles frontales. Los nemotécnicos del lenguaje ensamblador eran abstracciones diseñadas para evitar que los programadores tuvieran que recordar las secuencias de bits que componen las instrucciones de un programa. El siguiente nivel de abstracción se consigue agrupando instrucciones primitivas para formar macroinstrucciones.

Por ejemplo, un conjunto se puede definir por abstracción como una colección no ordenada de elementos en el que no existen duplicados. Utilizando esta definición, se pueden especificar si sus elementos se almacenan en un *array*, una lista enlazada o cualquier otra estructura de datos. Un conjunto de instrucciones realizadas por un usuario se pueden invocar por una macroinstrucción; una macroinstrucción instruye a la máquina para que realice muchas cosas. Tras los lenguajes de programación ensambladores aparecieron los lenguajes de programación de alto nivel, que supusieron un nuevo nivel de abstracción. Los lenguajes de programación de alto nivel permitieron a los programadores distanciarse de las interioridades arquitectónicas específicas de una máquina dada. Cada instrucción en un lenguaje de alto nivel puede invocar varias instrucciones máquina, dependiendo de la máquina específica donde se compila el programa. Esta abstracción permitía a los programadores escribir software para propósito genérico, sin preocuparse sobre que máquina corre el programa.

Secuencias de sentencias de lenguajes de alto nivel se pueden agrupar en procedimientos y se invocan por una sentencia. La programación estructurada

alienta el uso de abstracciones de control, tales como bucles o sentencias **if-then**, que se han incorporado en lenguajes de alto nivel Estas sentencias de control permitieron a los programadores abstraer las condiciones comunes para cambiar la secuencia de ejecución.

El proceso de abstracción fue evolucionando desde la aparición de los primeros lenguajes de programación. El método más idóneo para controlar la complejidad fue aumentar los niveles de abstracción. En esencia, la *abstracción* supone la capacidad de encapsular y aislar la información del diseño y ejecución En un determinado sentido, las técnicas orientadas a objetos pueden verse como un producto natural de una larga progresión histórica, que va desde las estructuras de control, pasando por los procedimientos, los módulos, los tipos abstractos de datos y los objetos.

En las siguientes secciones describiremos los mecanismos de abstracción que han conducido al desarrollo profundo de los objetos: procedimientos, módulos, tipos abstractos de datos y objetos

1.5.3. Procedimientos

Los procedimientos y funciones fueron uno de los primeros mecanismos de abstracción que se utilizaron ampliamente en lenguajes de programación Los procedimientos permitían tareas que se ejecutaban rápidamente, o eran ejecutadas sólo con ligeras variaciones, que se reunían en una entidad y se reutilizaban, en lugar de duplicar el código varias veces Por otra parte, el procedimiento proporcionó la primera posibilidad de *ocultación de información*. Un programador podía escribir un procedimiento o conjunto de procedimientos que se utilizaban por otros programadores. Estos otros programadores no necesitaban conocer con exactitud los detalles de la implementación; sólo necesitaban el interfaz necesario. Sin embargo, los procedimientos no resolvían todos los problemas En particular, no era un mecanismo efectivo para ocultar la información y para resolver el problema que se producía al trabajar múltiples programadores con nombres idénticos.

Para ilustrar el problema, consideremos un programador que debe escribir un conjunto de rutinas para implementar una pila. Siguiendo los criterios clásicos de diseño de software, nuestro programador establece en primer lugar el interfaz visible a su trabajo, es decir cuatro rutinas: `meter`, `sacar`, `pilavacía` y `pilallena`. A continuación implementa los datos mediante arrays, listas enlazadas, etc. Naturalmente, los datos contenidos en la pila no se pueden hacer locales a cualquiera de las cuatro rutinas, ya que se deben compartir por todos. Sin embargo, si las únicas elecciones posibles son variables locales o globales, entonces la pila se debe mantener en variables globales: por el contrario, al ser las variables globales, no existe un método para limitar la accesibilidad o visibilidad de dichas variables. Por ejemplo, si la pila se representa mediante un array denominado `datospila`, este dato debe ser conocido por otros programadores, que puedan desear crear variables utilizando el mismo nombre pero relativo a las referidas rutinas De modo similar, las rutinas citadas están reservadas y no se pueden utilizar en otras partes del programa para

otros propósitos. En Pascal existe el ámbito local y global. Cualquier ámbito que permite acceso a los cuatro procedimientos debe permitir también el acceso a sus datos comunes. Para resolver este problema se ha desarrollado un mecanismo de estructuración diferente

1.5.4. Módulos

Un módulo es una técnica que proporciona la posibilidad de dividir sus datos y procedimientos en una *parte privada* —sólo accesible dentro del módulo— y *parte pública* —accesible fuera del módulo— Los tipos, datos (variables) y procedimientos se pueden definir en cualquier parte.

El criterio a seguir en la construcción de un módulo es que si no se necesita algún tipo de información, no se debe tener acceso a ella. Este criterio es *la ocultación de información*.

Los módulos resuelven algunos problemas, pero no todos los problemas del desarrollo de software. Por ejemplo, los módulos permitirán a nuestros programadores ocultar los detalles de la implementación de su pila, pero ¿qué sucede si otros usuarios desean tener dos o más pilas? Supongamos que un programador ha desarrollado un tipo de dato `Complejo` (representación de un número complejo) y ha definido las operaciones aritméticas sobre números complejos —suma, resta, multiplicación y división; asimismo ha definido rutinas para convertir números convencionales a complejos. Se presenta un problema: sólo puede manipular un número complejo El sistema de números complejos no será útil con esta restricción, pero es la situación en que se encuentra el programador con módulos simples

Los módulos proporcionan un método efectivo de ocultación de la información, pero no permiten realizar *instanciación*, que es la capacidad de hacer múltiples copias de las zonas de datos

1.5.5. Tipos abstractos de datos

Un *tipo abstracto de datos* (TAD) es un tipo de dato definido por el programador que se puede manipular de un modo similar a los tipos de datos definidos por el sistema. Al igual que los tipos definidos por el sistema, un tipo *de dato abstracto* corresponde a un conjunto (puede ser de tamaño indefinido) de valores legales de datos y un número de operaciones primitivas que se pueden realizar sobre esos valores. Los usuarios pueden crear variables con valores que están en el rango de valores legales y pueda operar sobre esos valores utilizando las operaciones definidas. Por ejemplo, en el caso de la pila ya citada se puede definir dicha pila como un tipo abstracto de datos y las operaciones sobre la pila como las únicas operaciones legales que están permitidas para ser realizadas sobre instancias de la pila.

Los módulos se utilizan frecuentemente como una técnica de implementación para tipos abstractos de datos, y el tipo abstracto de datos es un concepto más teórico. Para construir un tipo abstracto de datos se debe poder:

1. Exponer una definición del tipo
2. Hacer disponible un conjunto de operaciones que se puedan utilizar para manipular instancias de ese tipo
3. Proteger los datos asociados con el tipo de modo que sólo se pueda actuar sobre ellas con las rutinas proporcionadas.
4. Hacer instancias múltiples del tipo

Los módulos son mecanismos de ocultación de información y no cumplen básicamente más que los apartados 2 y 3. Los tipos abstractos de datos se implementan con *módulos* en Modula-2 y *paquetes* en CLU o Ada.

1.5.6. Objetos

Un objeto es sencillamente un tipo abstracto de datos al que se añaden importantes innovaciones en compartición de código y reutilización. Los mecanismos básicos de orientación a objetos son: *objetos, mensajes y métodos, clases e instancias y herencia*.

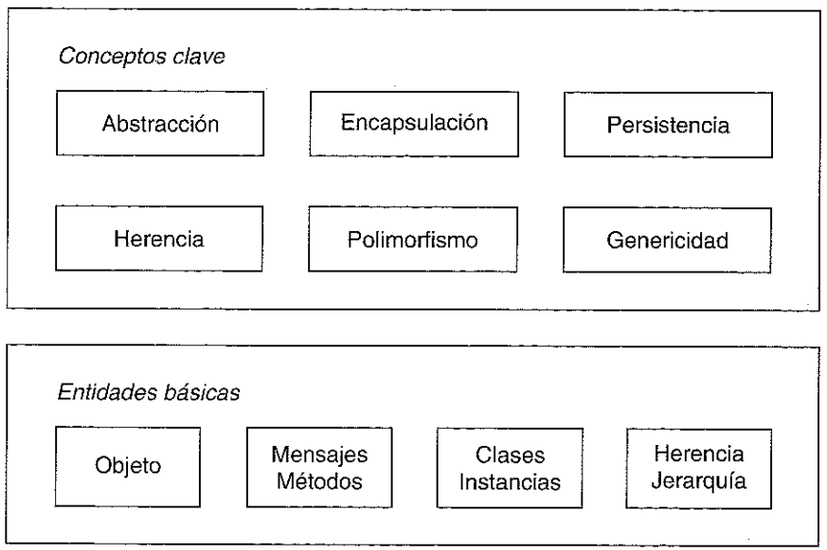


Figura 1.3. Principios básicos de la orientación a objetos.

Una idea fundamental es la comunicación de los objetos a través de *paso de mensajes*. Además de esta idea, se añaden los mecanismos de *herencia y polimorfismo*. La herencia permite diferentes tipos de datos para compartir el mismo código, permitiendo una reducción en el tamaño del código y un incremento en la funcionalidad. El polimorfismo permite que un mismo mensaje pueda actuar sobre objetos diferentes y comportarse de modo distinto.

La *persistencia* se refiere a la permanencia de un objeto, esto es, la cantidad de tiempo para el cual se asigna espacio y permanece accesible en la memoria del computador.

1.6. UN NUEVO PARADIGMA DE PROGRAMACION

La *programación orientada a objetos* **(POO)*** se suele conocer como un nuevo *paradigma* de programación. Otros paradigmas conocidos son: *el paradigma de la programación imperativa* (con lenguajes tales como Pascal o C), *el paradigma de la programación lógica* **(PROLOG)** y *el paradigma de la programación funcional* **(Lisp)**. El significado de *paradigma*[3] (*paradigma* en latín; *paradeigma* en griego) en su origen significaba un ejemplo ilustrativo, en particular enunciado modelo que mostraba todas las inflexiones de una palabra. En el libro *The Structure of Scientific Revolutions*, el historiador Thomas Kuhn[4] describía un paradigma como un conjunto de teorías, estándar y métodos que juntos representan un medio de organización del conocimiento: es decir, un medio de visualizar el mundo. En este sentido, la programación orientada a objetos es un nuevo paradigma. La orientación a objetos fuerza a reconsiderar nuestro pensamiento sobre la computación, sobre lo que significa realizar computación y sobre cómo se estructura la información dentro del computador[5].

Jenkins y Glasgow observan que «la mayoría de los programadores trabajan en un lenguaje y utilizan sólo un estilo de programación. Ellos programan en un paradigma forzado por el lenguaje que utilizan. Con frecuencia, no se enfrentan a métodos alternativos de resolución de un problema, y por consiguiente tienen dificultad en ver la ventaja de elegir un estilo más apropiado al problema a manejar». Bobrow y Stefik definen un estilo de programación como «un medio de organización de programas sobre la base de algún modelo conceptual de programación y un lenguaje apropiado para hacer programas en un estilo claro». Sugieren que existen cuatro clases de estilos de programación:

- Orientados a procedimientos — *Algoritmos*
- Orientados a objetos — *Clases y objetos*
- Orientados a lógica — *Expresado en cálculo de predicados*
- Orientados a reglas — *Reglas* `if-then`

No existe ningún estilo de programación idóneo para todas las clases de programación. La orientación a objetos se acopla a la simulación de situaciones del mundo real.

En POO, las entidades centrales son los **objetos**, que son tipos de datos que encapsulan con el mismo nombre estructuras de datos y las operaciones o algoritmos que manipulan esos datos.

---

[3] Un ejemplo que sirve como modelo o patrón: *Dictionary of Science and Technology*. Academic Press, 1992.

[4] KUHN, Thomas S.: *The Structure of Scientific Revolution* 2ª ed., University of Chicago Press. Chicago, 1970.

[5] Object-Oriented Programming **(OOP)**

1.7. ORIENTACION A OBJETOS

La orientación a objetos puede describirse como el *conjunto de disciplinas (ingeniería) que desarrollan y modelizan software que facilitan la construcción de sistemas complejos a partir de componentes*.

El atractivo intuitivo de la orientación a objetos es que proporciona conceptos y herramientas con las cuales se modela y representa el mundo real tan fielmente como sea posible. Las ventajas de la orientación a objetos son muchas en programación y modelación de datos. Como apuntaban Ledbetter y Cox (1985):

> La programación orientada a objetos permite una representación más directa del modelo de mundo real en el código. El resultado es que la transformación radical normal de los requisitos del sistema (definido en términos de usuario) a la especificación del sistema (definido en términos de computador) se reduce considerablemente.

La Figura 1.4 ilustra el problema. Utilizando técnicas convencionales, el código generado para un problema de mundo real consta de una primera codificación del problema y a continuación la transformación del problema en términos de un lenguaje de computador Von Newmann. Las disciplinas y técnicas orientadas a objetos manipulan la transformación automáticamente, de modo que el volumen de código codifica el problema y la transformación se minimiza. De hecho, cuando se compara con estilos de programación convencionales (*procedimentales por procedimientos*), las reducciones de código van desde un 40 por 100 hasta un orden de magnitud elevado cuando se adopta un estilo de programación orientado a objetos.

Los conceptos y herramientas orientados a objetos son tecnologías que permiten que los problemas del mundo real sean expresados de modo fácil y

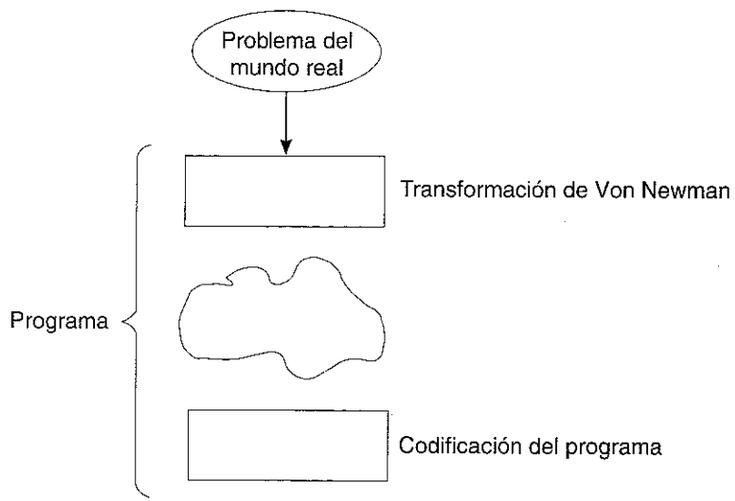


Figura 1.4. Construcción de software.

natural. Las técnicas orientadas a objetos proporcionan mejoras y metodologías para construir sistemas de software complejos a partir de unidades de software modularizado y reutilizable.

Se necesita un nuevo enfoque para construir software en la actualidad. Este nuevo enfoque debe ser capaz de manipular tanto sistemas grandes como pequeños y debe crear sistemas fiables que sean flexibles, mantenibles y capaces de evolucionar para cumplir las necesidades de cambio.

La tecnología orientada a objetos puede cubrir estos cambios y algunos otros más en el futuro.

La orientación a objetos trata de cumplir las necesidades de los usuarios finales, así como las propias de los desarrolladores de productos software. Estas tareas se realizan mediante la modelización del mundo real. El soporte fundamental es el *modelo objeto*. Los cuatro elementos (propiedades) más importantes de este modelo[6] son:

- Abstracción.
- Encapsulación.
- Modularidad.
- Jerarquía.

Como sugiere Booch, si alguno de estos elementos no existe, se dice que el modelo no es orientado a objetos.

1.7.1. Abstracción

La *abstracción* es uno de los medios más importantes, mediante el cual nos enfrentamos con la complejidad inherente al software. La abstracción es la propiedad que permite representar las características esenciales de un objeto, sin preocuparse de las restantes características (no esenciales).

Una abstracción se centra en la vista externa de un objeto, de modo que sirva para separar el comportamiento esencial de un objeto de su implementación. Definir una abstracción significa describir una entidad del mundo real, no importa lo compleja que pueda ser, y a continuación utilizar esta descripción en un programa.

El elemento clave de la programación orientada a objetos es la clase. Una **clase** se puede definir como una descripción abstracta de un grupo de objetos, cada uno de los cuales se diferencia por su *estado* específico y por la posibilidad de realizar una serie de *operaciones*. Por ejemplo, una pluma estilográfica es un objeto que tiene un estado (llena de tinta o vacía) y sobre la cual se pueden realizar algunas operaciones (por ejemplo escribir, poner o quitar el capuchón, llenar de tinta si está vacía).

La idea de escribir programas definiendo una serie de abstracciones no es nueva, pero el uso de clases para gestionar dichas abstracciones en lenguajes de programación ha facilitado considerablemente su aplicación.

---

[6] BOOCH, Grady: *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1994.

1.7.2. Encapsulación

La *encapsulación* o *encapsulamiento* es la propiedad que permite asegurar que el contenido de la información de un objeto está oculta al mundo exterior: el objeto A no conoce lo que hace el objeto B, y viceversa. La encapsulación (también se conoce como *ocultación de la información*), en esencia, es el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales.

La encapsulación permite la división de un programa en módulos. Estos módulos se implementan mediante clases, de forma que una clase representa la encapsulación de una abstracción. En la práctica, esto significa que cada clase debe tener dos partes: un interfaz y una implementación. El *interfaz* de una clase captura sólo su vista externa y la *implementación* contiene la representación de la abstracción, así como los mecanismos que realizan el comportamiento deseado.

1.7.3. Modularidad

La *modularidad* es la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas *módulos*), cada una las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.

La modularización, como indica Liskov, consiste en dividir un programa en módulos que se puedan compilar por separado, pero que tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan la modularidad de diversas formas. Por ejemplo, en C++ los módulos son archivos compilados por separado. La práctica usual (se verá en el Capítulo 2) es situar los interfaces de los módulos en archivos con nombres con extensión `.h` (*archivos de cabecera*) y las implementaciones de los módulos se sitúan en archivos con nombres con extensión `.cpp`.

En Ada, el módulo se define como *paquete* (`package`). Un paquete tiene dos partes: la *especificación* del paquete y el *cuerpo* del paquete; también se pueden compilar por separado.

Como se verá en el Capítulo 2, la *modularidad* es la propiedad de un sistema que permite su descomposición en un conjunto de módulos cohesivos y débilmente acoplados.

1.7.4. Jerarquía

La jerarquía es una propiedad que permite una ordenación de las abstracciones. Las dos jerarquías más importantes de un sistema complejo son: estructura de clases (jerarquía **«es-un»** (*is-a*): generalización/especialización) y estructura de objetos (jerarquía **«parte-de»** (*part-of*): agregación).

> No se debe confundir clases y objetos de la misma clase: un coche rojo y un coche azul no son objetos de clases diferentes, sino objetos de la misma clase con un atributo diferente.

Las jerarquías de generalización/especialización se conocen como *herencia*. Básicamente, la herencia define una relación entre clases, en donde una clase comparte la estructura o comportamiento definido en una o más clases (*herencia simple* y *herencia múltiple*, respectivamente).

La agregación es el concepto que permite el agrupamiento físico de estructuras relacionadas lógicamente. Así, un camión se compone de ruedas, motor, sistema de transmisión y chasis; en consecuencia, camión es una *agregación*, y ruedas, motor, transmisión y chasis son agregados de camión.

1.7.5. Polimorfismo

La quinta propiedad significativa de los lenguajes de programación orientados a objetos es el *polimorfismo*. Esta propiedad no suele ser considerada como fundamental en los diferentes modelos de objetos propuestos, pero, dada su importancia, no tiene sentido considerar un *objeto modelo* que no soporte esta propiedad.

Polimorfismo es la propiedad que indica, literalmente, la posibilidad de que una entidad tome *muchas formas*. En términos prácticos, el polimorfismo permite referirse a objetos de clases diferentes mediante el mismo elemento de programa y realizar la misma operación de diferentes formas, según sea el objeto que se referencia en ese momento.

Por ejemplo, cuando se describe la clase *mamíferos* se puede observar que la operación *comer* es una operación fundamental en la vida de los mamíferos, de modo que cada tipo de mamífero debe poder realizar la operación o función *comer*. Por otra parte, una vaca o una cabra que pastan en un campo, un niño que se come un bombón o caramelo y un león que devora a otro animal, son diferentes formas que utilizan los distintos mamíferos para realizar la misma función (*comer*).

El polimorfismo implica la posibilidad de tomar un objeto de un tipo (mamífero, por ejemplo) e indicarle que ejecute *comer*; esta acción se ejecutará de diferente forma, según sea el objeto mamífero sobre el que se aplica.

Clases, herencia y polimorfismo son aspectos claves en la programación orientada a objetos, y se reconocen a estos elementos como esenciales en la *orientación a objetos*. El polimorfismo adquiere su máxima expresión en la *derivación* o *extensión* de clases, es decir, cuando se obtiene una clase a partir de una clase ya existente, mediante la propiedad de derivación de clases o herencia. Así, por ejemplo, si se dispone de una figura que represente figuras genéricas, se puede enviar cualquier mensaje, tanto a un tipo derivado (elipse, círculo, cuadrado, etc.) como al tipo base. Por ejemplo, una clase *figura* puede aceptar los mensajes *dibujar*, *borrar* y *mover*. Cualquier tipo derivado de una *figura* es un tipo de figura y puede recibir el *mismo* mensaje. Cuando se envía un mensaje,

por ejemplo dibujar, esta tarea será distinta según que la clase sea un triángulo, un cuadrado o una elipse. Esta propiedad es el polimorfismo, que permite que una misma función se comporte de diferente forma según sea la clase sobre la que se aplica. La función *dibujar* se aplica igualmente a un círculo, a un cuadrado o a un triángulo, y el objeto ejecutará el código apropiado dependiendo del tipo específico.

El polimorfismo requiere *ligadura tardía* o *postergada* (también llamada *dinámica*), y esto sólo se puede producir en lenguajes de programación orientados a objetos. Los lenguajes no orientados a objetos soportan *ligadura temprana* o *anterior*; esto significa que el compilador genera una llamada a un nombre específico de función y el enlazador (*linker*) resuelve la llamada a la dirección absoluta del código que se ha de ejecutar. En POO, el programa no puede determinar la dirección del código hasta el momento de la ejecución; para resolver este concepto, los lenguajes orientados a objetos utilizan el concepto de *ligadura tardía*. Cuando se envía un mensaje a un objeto, el código que se llama no se determina hasta el momento de la ejecución. El compilador asegura que la función existe y realiza verificación de tipos de los argumentos y del valor de retorno, pero no conoce el código exacto a ejecutar.

Para realizar la ligadura tardía, el compilador inserta un segmento especial de código en lugar de la llamada absoluta. Este código calcula la dirección del cuerpo de la función para ejecutar en tiempo de ejecución utilizando información almacenada en el propio objeto. Por consiguiente, cada objeto se puede comportar de modo diferente de acuerdo al contenido de ese puntero. Cuando se envía un mensaje a un objeto, éste sabe qué ha de hacer con ese mensaje.

1.7.6. Otras propiedades

El modelo objeto ideal no sólo tiene las propiedades anteriormente citadas al principio del apartado, sino que es conveniente que soporte, además, estas otras propiedades:

- *Concurrencia* (multitarea).
- *Persistencia*.
- *Genericidad*
- *Manejo de excepciones*

Muchos lenguajes soportan todas estas propiedades y otros sólo algunas de ellas. Así, por ejemplo, Ada soporta concurrencia y Ada y C++ soportan genericidad y manejo de excepciones. La persistencia o propiedad de que las variables —y por extensión a los objetos— existan entre las invocaciones de un programa es posiblemente la propiedad menos implantada en los LPOO[7], aunque ya es posible considerar la persistencia en lenguajes tales como Smalltalk y C+, lo que facilitará el advenimiento de las bases de datos orientadas a objetos, como así está sucediendo en esta segunda mitad de la década de los noventa.

[7] Lenguaje de Programación Orientado a Objetos.

1.8. REUTILIZACION DE SOFTWARE

Cuando se construye un automóvil, un edificio o un dispositivo electrónico, se ensamblan una serie de piezas independientes, de modo que estos componentes se reutilicen, en vez de fabricarlos cada vez que se necesita construir un automóvil o un edificio. En la construcción de software, esta pregunta es continua. ¿Por qué no se utilizan programas ya construidos para formar programas más grandes? Es decir, si en electrónica los computadores y sus periféricos se forman esencialmente con el ensamblado de circuitos integrados, ¿existe algún método que permita realizar grandes programas a partir de la utilización de otros programas ya realizados? ¿Es posible reutilizar estos componentes de software?

Las técnicas orientadas a objetos proporcionan un mecanismo para construir *componentes de software* reutilizables que posteriormente puedan ser interconectados entre sí y formar grandes proyectos de software[8].

En los sistemas de programación tradicionales, y en particular en los basados en lenguajes de programación estructuradas (tales como FORTRAN, C, etc.), existen las bibliotecas de funciones, que contienen funciones (o procedimientos, según el lenguaje) que pueden ser incorporados en diferentes programas. En sistemas orientados a objetos se pueden construir componentes de software reutilizables, al estilo de las bibliotecas de funciones, normalmente denominados *bibliotecas de software o paquetes de software reutilizables*. Ejemplos de componentes reutilizables comercialmente disponibles son: Turbo Visión de Turbo Pascal OLE 2.0 de C++, jerarquía de clases Smalltalk, clases MacApp para desarrollo de interfaces gráficos de usuario en Object Pascal, disponibles en Apple, la colección de clases de Objective-C, etc.

En el futuro inmediato, los ingenieros de software dispondrán de catálogos de paquetes de software reutilizable, al igual que sucede con los catálogos de circuitos integrados electrónicos, como les ocurre a los ingenieros de hardware.

Las técnicas orientadas a objetos ofrecen una alternativa de escribir el mismo programa una y otra vez. El programador orientado a objetos modifica una funcionalidad del programa sustituyendo elementos antiguos u objetos por nuevos objetos, o bien conectando simplemente nuevos objetos en la aplicación.

La reutilización de código en reprogramación tradicional se puede realizar copiando y editando, mientras que en programación orientada a objetos se puede reutilizar el código, creando automáticamente una subclase y anulando alguno de sus métodos.

Muchos lenguajes orientados a objetos fomentan la reutilización mediante el uso de bibliotecas robustas de clases preconstruidas, así como otras herramientas, como *hojeadores («browser»)*, para localizar clases de interés y depuradores interactivos para ayudar al programador.

[8] BRAD COX, en su ya clásico libro *Object-Oriented Programming An Evolutionary Approach* [Cox, Novobilski, 91], acuñó el término *chip de software* **(Software-IC)**, o *componentes de software* para definir las clases de objetos como componentes de software reutilizables. Existe versión en español de Addison-Wesley/Díaz de Santos, 1993, con el título *Programación orientada a objetos Un enfoque evolutivo*.

1.9. LENGUAJES DE PROGRAMACION ORIENTADOS A OBJETOS

El primer lenguaje de programación que introdujo el concepto de *clase* fue Simula-67, como entidad que contenía datos y las operaciones que manipulaban los datos. Asimismo, introdujo también el concepto de herencia

El siguiente lenguaje orientado a objetos, y seguramente el más popular desde un enfoque conceptual exclusivamente de objetos, es Smalltalk, cuya primera versión comercial se desarrolló en 1976, y en 1980 se popularizó con la aparición de Smalltalk-80. Posteriormente se ha popularizado gracias al desarrollo de Smalltalk/V de la casa Digital Y, que recientemente se ha implementado bajo entorno Windows. El lenguaje se caracteriza por soportar todas las propiedades fundamentales de la orientación a objetos, dentro de un entorno integrado de desarrollo, con interfaz interactivo de usuario basado en menús.

Entre los lenguajes orientados a objetos que se han desarrollado a partir de los ochenta destacan extensiones de lenguajes tradicionales tales como C++ y Objective-C (extensiones de C), Modula-2 y Object Pascal (extensión de Pascal) y recientemente Object Cobol, que a lo largo de 1994 han aparecido sus primeras versiones comerciales, y Java, el nuevo lenguaje para programación en Internet.

Otro lenguaje orientado a objetos puros es Eiffel, creado por Bertrand Meyer y que soporta todas las propiedades fundamentales de objetos. Hasta ahora no ha adquirido popularidad más que en ambientes universitarios y de investigación. Sin embargo, la prevista aparición para el año 1995 de la versión 3, que correrá bajo Windows, seguramente aumentará su difusión

Ada ha sido también un lenguaje —en este caso *basado en objetos*— que soporta la mayoría de las propiedades orientadas a objetos. Sin embargo, la nueva versión Ada-95 ya soporta herencia y polimorfismo.

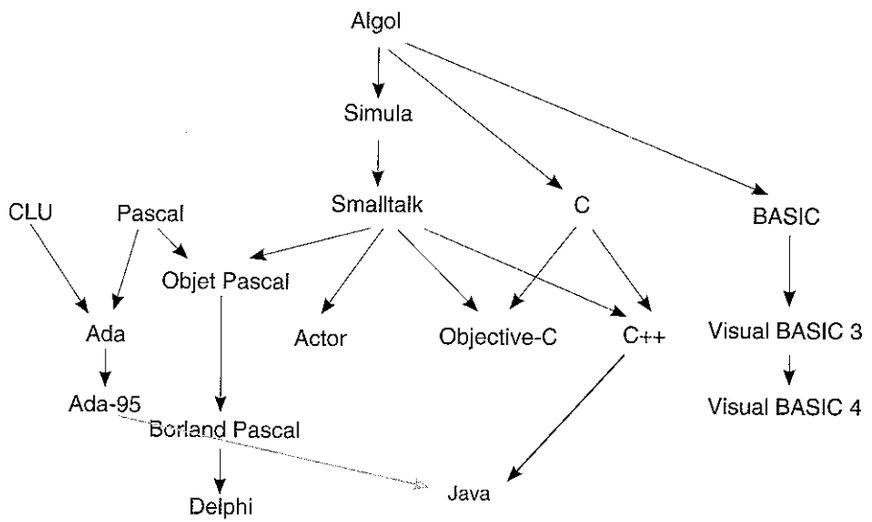


Figura 1.5. Evolución de los lenguajes orientados a objetos.

En los últimos años han aparecido lenguajes con soporte de objetos que cada vez se están popularizando más: Clipper 5-2, Visual BASIC, etc.

De cualquier forma, existe un *rey* actual en los lenguajes orientados a objetos: C++. La normalización por ANSI y AT&T de la versión 3.0 y las numerosas versiones de diferentes fabricantes, tales como Borland C++ 4.0/4.5, Turbo C++ 3.0/3.1 y 4.5, Microsoft C/C++ 7.0, Visual C++ 1.5/2, Symantec 6.0/7.0, etcétera, hacen que en la segunda mitad de los noventa será el lenguaje orientado a objetos más popular y utilizado en el mundo de la programación.

La evolución de los lenguajes orientados a objetos se han mostrado en la Figura 1.5, en la que se aprecia el tronco común a todos los lenguajes modernos Algol y las tres líneas fundamentales: enfoque en Pascal (Ada, Object Pascal), enfoque puro de orientación a objetos (Simula/Smalltalk/Eiffel) y enfoque en C (Objective-C, C++, Java)

1.9.1. Clasificación de los lenguajes orientados a objetos

Existen varias clasificaciones de lenguajes de programación orientados a objetos, atendiendo a criterios de construcción o características específicas de los mismos. Una clasificación ampliamente aceptada y difundida es la dada por Wegner y que se ilustra en la Figura 1.6[9].

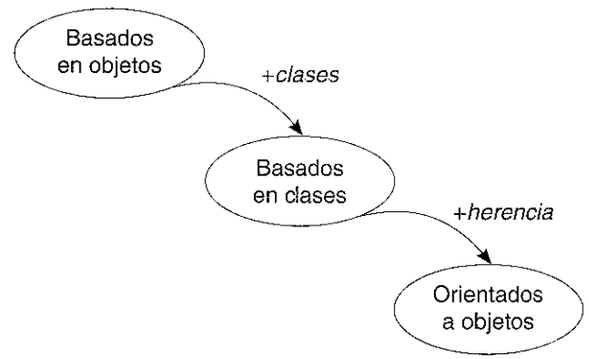


Figura 1.6. Clasificación de lenguajes OO de Wegner.

La clasificación de Wegner divide los lenguajes en tres categorías:

1. Lenguajes **basados en objetos** que soportan objetos. Es decir, disponen de componentes caracterizados por un conjunto de operaciones (comportamiento) y un estado
2. Lenguajes **basados en clases** que implican objetos y clases. Es decir, disponen de componentes tipo clase con operaciones y estado común.

[9] WEGNER, Peter [1987]: *Dimensions of Object-Based Languajes Design*. Número especial de *SIGPLAN Notices*.

Una clase de un objeto se construye con un *«interfaz»* que especifica las operaciones posibles y un «cuerpo» que implementa dichas operaciones.

3. Lenguajes **orientados a objetos** que además de objetos y clases ofrecen mecanismos de herencia entre clases. Esto es, la posibilidad de derivar operaciones y atributos de una clase (superclase) a sus subclases.

La definición anterior, pese a su antigüedad, sigue teniendo vigencia. Existen otras clasificaciones similares, pero con la inclusión de la propiedad de *polimorfismo* en la categoría 3, como requisito para ser lenguaje orientado a objetos.

De cualquier forma, hoy en día es posible ampliar esa clasificación de acuerdo a criterios puramente técnicos y hacer una nueva clasificación de la categoría 3:

3.1 *Lenguajes orientados a objetos* **puros**. Soportan en su totalidad el paradigma de orientación a objetos:

Smalltalk Eiffel Simula

3.2 *Lenguajes orientados a objetos* **híbridos**. Soportan en su totalidad el paradigma de orientación a objetos sobre un núcleo de lenguaje híbrido:

C++ (*extensión* de C: Borland C++, Microsoft C++, Turbo C++, Visual C++, Symantec,Watcom.)
Objective-C (*extensión* de C).
Object COBOL (*extensión* de COBOL).
Object Pascal (*extensión de Pascal*: Turbo/Borland Pascal).
Visual Object (*extensión* de Clipper).
Delphi (*extensión* de Turbo Pascal 7.0).
Java (*extensión* de C++ y Ada-95).

Tabla 1.1. Criterios de Meyer en lenguajes OO y basados en objetos.

	Criterios	Ada-83	Ada-95	C++	Eiffel	Smalltalk	Java
1.	Modularización	Sí	Sí	Sí	Sí	Sí	Sí
2	Tipos abstractos de datos	Sí	Sí	Sí	Sí	Sí	Sí
3	Gestión automática de memoria	Sí	Sí	*En parte*	Sí	Sí	Sí
4	Sólo clases	Sí	Sí	*En parte*	Sí	Sí	Sí
5	Herencia	No	Sí	Sí	Sí	Sí	Sí
6	Polimorfismo (y ligadura dinámica)	No	Sí	Sí	Sí	Sí	Sí
7.	Herencia múltiple y repetida	No	No	Sí	Sí	No	No

De cualquier forma, Meyer, creador del lenguaje Eiffel, proporciona unos criterios para considerar la «bondad»[10] de un lenguaje orientado a objetos,

[10] MEYER, Bertrand: *Object Oriented Software Construction* Englewood Cliffs NJ, Prentice-Hall, 1988

cuyos complementos configuran, *de hecho*, una nueva clasificación. En este sentido, los criterios recogidos por este autor son los siguientes:

1. La modularización de los sistemas ha de realizarse mediante estructuras de datos apropiadas.
2. Los objetos se describen como la implementación de tipos abstractos de datos
3. La memoria se gestiona (administra) automáticamente
4. Existe una correspondencia entre tipos de datos no elementales y clases
5. Las clases se pueden definir como extensiones o restricciones de otras clases ya existentes mediante herencia
6. Soportan polimorfismo y ligadura dinámica
7. Existe herencia múltiple y repetida.

De acuerdo con los criterios de Meyer, recogemos en la Tabla 1.1 el cumplimiento de dichos criterios en los lenguajes OO y basados en objetos más populares.

1.10. DESARROLLO TRADICIONAL FRENTE A ORIENTADO A OBJETOS

El sistema tradicional del desarrollo del software para un determinado sistema es la subdivisión del mismo en módulos, a la cual deben aplicarse criterios específicos de descomposición, los cuales se incluyen en metodologías de diseño. Estos módulos se refieren a la fase de construcción de un programa, que en el modelo clásico sigue a la definición de los requisitos (fase de análisis), que se muestra en la Figura 3.1.

El modelo clásico del ciclo de vida del software no es el único modelo posible, dado que es posible desarrollar código de un modo evolutivo, por refinamiento y prototipos sucesivos. Existen numerosos lenguajes de programación y metodologías que se han desarrollado en paralelo a los mismos, aunque normalmente con independencia de ellos.

En esta sección nos centraremos en la metodología más utilizada, denominada *desarrollo estructurado*, que se apoya esencialmente en el *diseño descendente* y en la *programación estructurada*.

La *programación estructurada* es un estilo disciplinado de programación según los lenguajes procedimentales (por procedimientos), tales como FORTRAN, BASIC, COBOL y recientemente C y C++.

Las metodologías *diseño descendente* (o descomposición funcional) se centran en operaciones y tienden a descuidar la importancia de las esrtructuras de datos. Se basan en la célebre ecuación de Wirth:

Datos + Algoritmos = Programas

La idea clave del diseño descendente es romper un programa grande en tareas más pequeñas, más manejables. Si una de estas tareas es demasiado grande, se divide en tareas más pequeñas. Se continúa con este proceso hasta

que el programa se compartimentaliza en módulos más pequeños y que se programan fácilmente. Los subprogramas facilitan el enfoque estructurado, y en el caso de lenguajes como C, estas unidades de programas, llamadas *funciones*, representan las citadas tareas o módulos individuales. Las técnicas de programación estructuradas reflejan, en esencia, un modo de resolver un programa en términos de las acciones que realiza.

Para comprender mejor las relaciones entre los algoritmos (*funciones*) y los datos, consideremos una comparación con el lenguaje natural (por ejemplo español o inglés), que se compone de muchos elementos pero que reflejará poca expresividad si sólo se utilizan nombres y verbos. Una metodología que se basa *sólo en datos* o *sólo en procedimientos* es similar a un lenguaje (idónea) en el que sólo se utilizan nombres o verbos. Sólo enlazando nombres o verbos correctos (siguiendo las reglas semánticas), las expresiones tomarán formas inteligibles y su proceso será más fácil.

Las metodologías tradicionales se vuelven poco prácticas cuando han de aplicarse a proyectos de gran tamaño. El diseño orientado a objetos se apoya en lenguajes orientados a objetos que se sustentan fundamentalmente en los tipos de datos y operaciones que se pueden realizar sobre los tipos de datos. Los datos no fluyen abiertamente en un sistema, como ocurre en las técnicas estructuradas, sino que están protegidos de modificaciones accidentales. En programación orientada a objetos, los *mensajes* (en vez de los datos) se mueven por el sistema. En lugar del enfoque funcional (invocar una función con unos datos), en un lenguaje orientado a objetos, «se envía un mensaje a un objeto».

De acuerdo con Meyer, el diseño orientado a objetos es el método que conduce a arquitecturas de software basadas en objetos que cada sistema o subsistema evalúa.

Recordemos *¿qué son los objetos*? Un **objeto** es una entidad cuyo comportamiento se caracteriza por las acciones que realiza. Con más precisión, un objeto se define como una entidad caracterizada por un estado; su comportamiento se define por las operaciones que puede realizar; es una *instancia* de una *clase*; se identifica por un nombre; tiene una visibilidad limitada para otros objetos; se define el objeto mediante su especificación y su implementación.

Una definición muy elaborada se debe a Meyer: «Diseño orientado a objetos es la construcción de sistemas de software como colecciones estructuradas de implementaciones de tipos de datos abstractos».

La construcción de un sistema se suele realizar mediante el ensamblado ascendente (abajo-arriba) de clases preexistentes. Las clases de un sistema pueden tener entre sí, como se verá en los siguientes capítulos, relaciones de uso (*cliente*), relaciones de derivación (*herencia*) o relaciones de agregación (*composición*) o incluso sólo relaciones de asociación. Así, por ejemplo, con una relación de cliente, una clase puede utilizar los objetos de otra clase; con una relación de herencia, una *clase puede heredar o derivar* sus propiedades definidas en otra clase.

El *hardware* se ensambla a partir de componentes electrónicos, tales como circuitos integrados (clips), que se pueden utilizar repetidamente para diseñar y construir conjuntos mucho más grandes, que son totalmente reutilizables. La calidad de cada nivel de diseño se asegura mediante componentes del sistema

que han sido probados previamente a su utilización. El ensamblado de componentes electrónicos se garantiza mediante interfaces adecuados.

Estos conceptos se aplican también con tecnologías de objetos. Las clases (tipos de objetos) son como los *chips de hardware*, Cox les llamó *chips de software*, que no sólo se pueden enlazar (ensamblar) entre sí, sino que también se pueden reutilizar (volver a utilizar). Las clases se agruparán normalmente en bibliotecas de clases, que son componentes reutilizables, fácilmente legibles.

En la actualidad existe gran cantidad de software convencional, en su mayoría escrito normalmente para resolver problemas específicos; por esta razón, a veces es más fácil escribir nuevos sistemas que convertir los existentes.

Los objetos, al reflejar entidades del mundo real, permiten desarrollar aplicaciones, creando nuevas clases y ensamblándolas con otras ya existentes. Normalmente, los desarrolladores experimentados gastan un porcentaje alto de su tiempo (20 al 40 por 100) en crear nuevas clases y el tiempo restante en ensamblar componentes probados de sistemas, construyendo sistemas potentes y fiables.

1.11. BENEFICIOS DE LAS TECNOLOGIAS DE OBJETOS (TO)

Una pregunta que hoy día se hacen muchos informáticos es: *¿Cuál es la razón para introducir métodos de TO en los procesos de desarrollo?* La principal razón, sin lugar a dudas, son los beneficios de dichas TO: aumento de la fiabilidad y productividad del desarrollador. La fiabilidad se puede mejorar, debido a que cada objeto es simplemente «una caja negra» con respecto a objetos externos con los que debe comunicarse. Las estructuras de datos internos y métodos se pueden refinar sin afectar a otras partes de un sistema (Fig. 1.7).

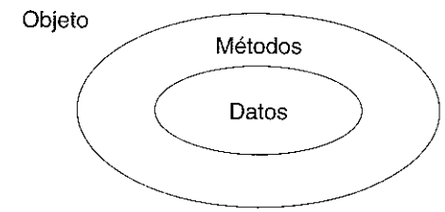


Figura 1.7. El objeto como *caja negra*.

Los sistemas tradicionales, por otra parte, presentan con frecuencia efectos laterales no deseados. Las tecnologías de objetos ayudan a los desarrolladores a tratar la complejidad en el desarrollo del sistema.

La productividad del desarrollador se puede mejorar, debido a que las clases de objetos se pueden hacer reutilizables de modo que cada subclase o instancia de un objeto puede utilizar el mismo código de programa para la clase. Por otra parte, esta productividad también aumenta, debido a que existe una asociación más natural entre objetos del sistema y objetos del mundo real.

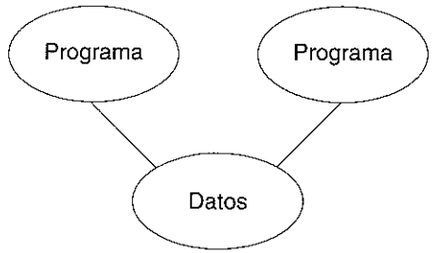


Figura 1.8. Proceso tradicional de datos.

Taylor[11] considera que los beneficios del modelado y desarrollo de objetos son:

1. Desarrollo más rápido.
2. Calidad más alta.
3. Mantenimiento más fácil.
4. Coste reducido.
5. Incrementò en escalabilidad
6. Mejores estructuras de información.
7. Incremento de adaptabilidad

Sin embargo, Taylor[12] también considera algunos inconvenientes, aunque algunos de ellos ya han sido superados o al menos reducido su impacto.

1. Inmadurez de la tecnología (hoy día ya no se puede considerar así).
2. Necesidades de estándares (el grupo OMG es una realidad).
3. Necesidad de mejores herramientas.
4. Velocidad de ejecución.
5. Disponibilidad de personal cualificado.
6. Coste de conversión.
7. Soporte para modularidad a gran escala.

La Figura 1.9 muestra los beneficios genéricos de las tecnologías de objetos.

- *Reutilización.* Las clases se construyen a partir de otras clases.
- *Sistemas más fiables.*
- *Proceso de desarrollo más rápido.*
- *Desarrollo más flexible.*
- *Modelos que reflejan mejor la realidad.*
- *Mejor independencia e interoperatividad de la tecnología.*
- *Mejor informática distribuida y cliente-servidor.*
- *Bibliotecas de clases comerciales disponibles.*
- *Mejor relaciones con los clientes.*
- *Mejora la calidad del producto software terminado.*

Figura 1.9. Beneficios de las tecnologías de objetos.

[11] TAYLOR, David A: *Objet-Oriented Technology Reading* MA: Addison-Wesley, 1992, páginas 103-107

[12] *Ibíd* págs. 108-113

RESUMEN

Este capítulo es una introducción a los métodos de desarrollo orientados a objetos. Se comienza con una breve revisión de los problemas encontrados en el desarrollo tradicional de software que condujeron a la crisis del software y que se han mantenido hasta los años actuales. El nuevo modelo de programación se apoya esencialmente en el concepto de objetos.

La orientación a objetos modela el mundo real de un modo más fácil a la perspectiva del usuario que el modelo tradicional. La orientación a objetos proporciona mejores técnicas y paradigmas para construir componentes de software reutilizables y bibliotecas ampliables de módulos de software. Esta característica mejora la extensibilidad de los programas desarrollados a través de metodologías de orientación orientada a objetos. Los usuarios finales, programadores de sistemas y desarrolladores de aplicaciones se benefician de las tecnologías de modelado y programación orientadas a objetos.

Los conceptos fundamentales de orientación a objetos son tipos abstractos de datos, herencia e identidad de los objetos. Un tipo abstracto de datos describe una colección con la misma estructura y comportamiento. Los tipos abstractos de datos extienden la noción de tipos de datos, ocultando la implementación de operaciones definidas por el usuario (mensajes) asociados con los tipos de datos. Los tipos abstractos de datos se implementan a través de clases. Las clases pueden heredar unas de otras. Mediante la herencia se pueden construir nuevos módulos de software (tales como clases) en la parte superior de una jerarquía existente de módulos. La herencia permite la compartición de código (y por consiguiente reutilización) entre módulos de software. La identidad es la propiedad de un objeto que diferencia cada objeto de los restantes. Con la identidad de un objeto, los objetos pueden contener o referirse a otros objetos. La identidad del objeto organiza los objetos del espacio del objeto manipulado por un programa orientado a objetos.

Este capítulo examina el impacto de las tecnologías orientadas a objetos en lenguajes de programación, así como los beneficios que producen en el desarrollo de software.

Los conceptos claves de la programación orientada a objetos se examina en el capítulo; si no ha leído hasta ahora nada sobre tecnologías de objetos, deberá examinar con detenimiento todos los elementos conceptuales del capítulo, que se ampliarán en detalle en capítulos posteriores; si ya tiene conocimientos básicos de la orientación a objetos, este capítulo debe consolidarlos y prepararle para una eficiente lectura de los siguientes capítulos.

CAPITULO 2

MODULARIDAD: TIPOS ABSTRACTOS DE DATOS

CONTENIDO

En este capítulo se examinarán los conceptos de *modularidad* y *abstracción de datos*. La modularidad es la posibilidad de dividir una aplicación en piezas más pequeñas llamadas módulos. *Abstracción de datos* es la técnica de inventar nuevos tipos de datos que sean más adecuados a una aplicación y, por consiguiente, facilitar la escritura del programa. La técnica de abstracción de datos es una técnica potente de propósito general, que cuando se utiliza adecuadamente puede producir programas más cortos, más legibles y flexibles.

Los lenguajes de programación soportan en sus compiladores *tipos de datos fundamentales o básicos (predefinidos)*, tales como `int`, `char` y `float` en C y C++, o bien `integer`, `real` o `boolean` en Pascal. Algunos lenguajes de programación tienen características que permiten ampliar el lenguaje añadiendo sus propios tipos de datos.

Un tipo de dato definido por el programador se denomina *tipo abstracto de dato, **TAD*** (*Abstract Data Type,* **ADT**). El término abstracto se refiere al medio en que un programador abstrae algunos conceptos de programación creando un nuevo tipo de dato.

La modularización de un programa utiliza la noción de *tipo abstracto de dato* (***TAD***) siempre que sea posible. Si el **TAD** soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado *objeto*.

2.1. MODULARIDAD

La programación modular trata de descomponer un programa en un pequeño número de abstracciones coherentes que pertenecen al dominio del problema y cuya complejidad interna es susceptible de ser enmascarada por la descripción de un interfaz.

Si las abstracciones que se desean representar pueden en ciertos casos corresponder a una única acción abstracta y se implementan en general con la noción de *objeto abstracto (o tipo abstracto)* caracterizado en todo instante por:

- Un *estado actual*, definido por un cierto número de atributos.
- Un *conjunto de acciones posibles*.

En consecuencia, la *modularidad* es la posibilidad de subdividir una aplicación en piezas más pequeñas (denominadas *módulos*), cada una de las cuales debe ser tan independiente como sea posible, considerando la aplicación como un todo, así como de las otras piezas de las cuales es una parte. Este principio básico desemboca en el principio básico de *construir programas modulares*. Esto significa que, aproximadamente, ha de subdividir un programa en piezas más pequeñas, o módulos, que son generalmente independientes de cada una de las restantes y se pueden *ensamblar* fácilmente para construir la aplicación completa.

En esencia, las abstracciones se implementan en *módulos*, conocidos en la terminología de Booch como objetos, que agrupan en una sola entidad:

- Un conjunto de datos.
- Un conjunto de operaciones que actúan sobre los datos.

Liskov define la modularización como «el proceso de dividir un programa en módulos que se pueden compilar separadamente, pero que tienen conexiones con otros módulos». Parnas va más lejos y dice que «las conexiones entre módulos deben seguir el criterio de ocultación de la información: un sistema se debe descomponer de acuerdo al criterio general, de que cada módulo oculta alguna decisión de diseño del resto del sistema; en otras palabras, cada módulo *oculta* un secreto».

Si un programa se descompone (o subdivide en módulos) de modo consistente con el criterio de Parnas —es decir, aplicando el principio de ocultación de la información—, se reduce la complejidad de cada módulo que compone la solución. Estos se constituyen en cierto modo independientes de los restantes, con lo que se reduce la necesidad de tomar decisiones globales, operaciones y datos.

2.1.1. La estructura de un módulo

Un módulo se caracteriza fundamentalmente por su *interfaz* y por *su implementación*. Parnas define el módulo como «un conjunto de acciones denominadas, funciones o submódulos que corresponden a una abstracción coherente, que compartan un conjunto de datos comunes implantadas estáticamente llamadas *atributos*, eventualmente asociadas a definiciones lógicas de tipos. Las acciones o funciones de un módulo que son susceptibles de ser llamadas desde el exterior

se denominan *primitivas* o *puntos de entrada* del módulo. Los tipos lógicos eventualmente definidos en el interfaz permiten representar los parámetros de estas primitivas».

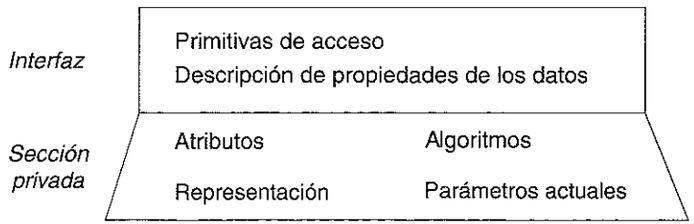


Figura 2.1. Estructura de un módulo.

2.1.2. Reglas de modularización

En primer lugar, un método de diseño debe ayudar al programador a resolver un problema, dividiendo el problema en subproblemas más pequeños, que se puedan resolver independientemente unos de otros. También debe ser fácil conectar los diferentes módulos a los restantes, dentro del programa que esté escribiendo

Cada módulo tiene un significado específico propio y debe asegurarse que cualquier cambio en su implementación no afecte a su exterior (o al menos lo mínimo). De igual modo, asegurar que los errores posibles, condiciones de límites o frontera, comportamientos erráticos, no se propaguen más allá del módulo (o como máximo a los módulos que estén directamente en contacto con el afectado).

Para obtener módulos con las características anteriores deben seguirse las siguientes reglas:

Unidades modulares

El lenguaje debe proporcionar estructuras modulares con las cuales se puedan describir las diferentes unidades. De este modo, el lenguaje (y el compilador) puede reconocer un módulo y debe ser capaz de manipular y gobernar su uso, además de las ventajas evidentes relativas a la legibilidad del código resultante. Estas construcciones modulares pueden, como en el caso de los lenguajes orientados a objetos, mostrar características que facilitan la estructura del programa, así como la escritura de programas. En otras palabras, nos referimos a las unidades modulares lingüísticas, que en el caso de C++ se conocen como *clases*. En C/C++ los módulos son archivos compilados separadamente, aunque la representación ideal en C++ es la *clase*. La práctica tradicional en C/C++ es situar interfaces del módulo en archivos cuyos nombres contienen el sufijo .h (archivos de cabecera). Las implementaciones del módulo se sitúan en un archivo cuyo nombre tiene el sufijo .c. Las dependencias entre archivos se pueden declarar utilizando la macro `#include`. En Turbo Pascal, los módulos se denominan *unidades*. La sintaxis de las unidades diferencia entre el interfaz y la implementación del módulo. Las dependencias entre unidades se pueden declarar sólo en un interfaz del módulo. Ada va más lejos y define el *paquete* en dos partes: la especificación del paquete y el cuerpo del paquete. Al contrario que Object Pascal, Ada permite que la conexión entre módulos se declaren independientemente en la especificación y en el cuerpo de un paquete.

Interfaces adecuados

En la estructuración de un programa en unidades es beneficioso que existan *pocos interfaces* y que éstos sean pequeños. Es conveniente que existan pocos enlaces entre los diferentes módulos en que se descompone un programa. *El interfaz de un módulo* es la parte del módulo (datos, procedimientos, etc.) que es visible fuera del módulo.

Los interfaces deben ser también *pequeños* (esto es, su tamaño debe ser pequeño con respecto al tamaño de los módulos implicados). De este modo, los módulos están acoplados débilmente; se enlazarán por un número pequeño de llamadas (Fig. 2.3).

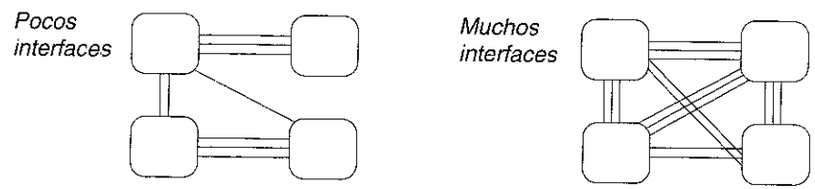


Figura 2.2. Interfaces adecuados (pocos-muchos).

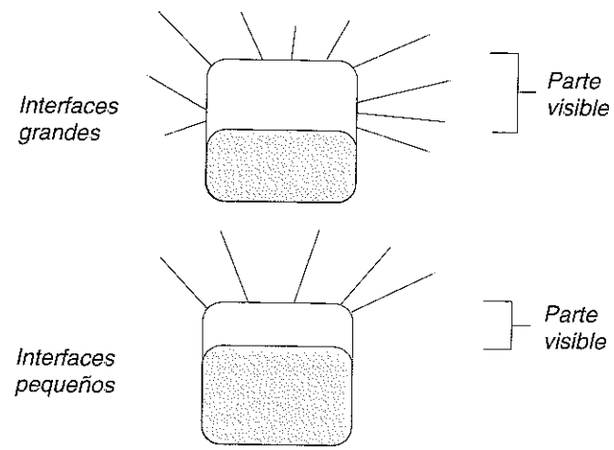


Figura 2.3. Interfaces adecuados (grandes-pequeños).

Interfaces explícitos

El interfaz o parte visible externamente de un módulo se debe declarar y describir explícitamente; el programa debe especificar cuáles son los datos y procedimientos que un módulo trata de exportar y cuáles deben permanecer ocultos del exterior. El interfaz debe ser fácilmente legible, tanto para el programador como para el compilador. Es decir, el programador debe comprender cómo funciona el programa y el compilador ha de poder comprobar si el código que accede al módulo se ha escrito correctamente.

Ocultación de la información

Todos los módulos deben seguir el principio de ocultación de la información; cada módulo debe representar al menos un elemento de diseño (por ejemplo, la estructura de un registro, un algoritmo, una abstracción, etc.).

Otro criterio a tener en cuenta es la subdivisión de un sistema en módulos, es el principio denominado *abierto-cerrado*[1], formulado por Meyer. Este principio entiende que cada módulo se considerará *cerrado* (esto es, terminado, y por consiguiente útil o activo desde dentro de otros módulos), y al mismo tiempo debe ser *abierto* (esto es, sometido a cambios y modificaciones). El principio *abierto-cerrado* debe producirse sin tener que rescribir todos los módulos que ya utilizan el módulo que se está modificando.

> En lenguajes de programación clásicos, la modularización se centra en los subprogramas (procedimientos, funciones y subrutinas). En lenguajes orientados a objetos, la modularización o partición del problema se resuelve a través de los tipos abstractos de datos.

En diseño estructurado, la modularización —como ya se ha comentado— se centra en el agrupamiento significativo de subprogramas, utilizando el criterio de acoplamiento y cohesión.

En diseño orientado a objetos, el problema es sutilmente diferente: la tarea consiste en decidir dónde se empaquetan físicamente las clases y objetos de la estructura lógica del diseño, que son claramente diferentes de los subprogramas.

[1] Sobre el principio *abierto-cerrado* y su implementación en C++ y Eiffel, se puede consultar la bibliografía de Miguel Katrib. Algunos títulos destacados sobre orientación a objetos son: *Programación Orientada a Objetos a través de C++ y Eiffel*. V Escuela Internacional en Temas Selectos de Computación, Zacatecas, México, 1994 (esta Escuela está organizada por la UNAM, México); *Programación Orientada a Objetos en C++*. Infosys, México, 1994; *Collections and Iterators in Eiffel*. Joop, vol. 6, n.º 7, nov/dic. 1993.

2.2. DISEÑO DE MODULOS

Aunque el diseño modular persigue la división de un sistema grande en módulos más pequeños y a la vez manejables, no siempre esta división es garantía de un sistema bien organizado. Los módulos deben diseñarse con los criterios de *acoplamiento* y *cohesión*. El primer criterio exige independencia de módulos y el segundo criterio se corresponde con la idea de que cada módulo debe realizarse con una sola función relacionada con el problema.

Desde esta perspectiva, Booch[2] define la modularidad como *«la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados»*.

2.2.1. Acoplamiento de módulos

El *acoplamiento* es una medida del grado de interdependencia entre módulos, es decir, el modo en que un módulo está siendo afectado por la *estructura interna* de otro módulo. El grado de acoplamiento se puede utilizar para evaluar la calidad de un diseño de sistema. El objetivo es minimizar el acoplamiento entre módulos, es decir, minimizar su interdependencia, de modo que un módulo sea afectado lo menos posible por la *estructura de otro módulo*. El acoplamiento entre módulos varía en un amplio rango. Por un lado, el diseño de un sistema puede tener una jerarquía de módulos totalmente desacoplados. Sin embargo, dado que un sistema debe realizar un conjunto de funciones o tareas de un modo organizado, no puede constar de un conjunto de módulos totalmente desacoplados. En el otro extremo se tendrá una jerarquía de módulos estrechamente acoplados; es decir, hay un alto grado de dependencia entre cada pareja de módulos del diseño.

Tal como define Booch, un sistema modular débilmente acoplado facilita:

1. La sustitución de un módulo por otro, de modo que sólo unos pocos módulos serán afectados por el cambio.
2. El seguimiento de un error y el aislamiento del módulo defectuoso que produce ese error.

Existen varias clases de acoplamiento entre dos módulos (Tabla 2.1). Examinaremos los cinco tipos de acoplamiento, desde el menos deseable (esto es, acoplamiento estrecho o impermeable) al más deseable (esto es, acoplamiento más débil). La fuerza de acoplamiento entre dos módulos está influenciada por el tipo de conexión, el tipo de comunicación entre ellos y la complejidad global de su interfaz.

2.2.2. Cohesión de módulos

La cohesión es una extensión del concepto de ocultamiento de la información. Dicho de otro modo, la cohesión describe la naturaleza de las interacciones

[2] BOOCH, Grady: *Object Oriented Design with applications*. Benjamin/Cummings, 1991, pág. 52.

dentro de un módulo software. Este criterio sugiere que un sistema bien modularizado es aquel en el cual los interfaces de los módulos son claros y simples. Un módulo cohesivo ejecuta una tarea sencilla de un procedimiento de software y requiere poca interacción con procedimientos que ejecutan otras partes de un programa. En otras palabras, un módulo cohesivo sólo hace (idealmente) una cosa.

La cohesión y el acoplamiento se miden como un *«espectro»* que muestra las escalas que siguen los módulos. La Tabla 2.1 muestra la clasificación de acoplamientos de módulos y su grado de acoplamiento.

Tabla 2.1. Clasificación del acoplamiento de módulos.

Tipo de acoplamiento	Grado de acoplamiento	Grado de mantenibilidad
Por contenido	Alto (fuerte)	Bajo
Común	↓	↓
De control		
Por sellado (estampado)		
Datos		
Sin acoplamiento	Bajo (débil)	Alto

La Tabla 2.2 muestra los grados de cohesión: baja cohesión (no deseable) y alta cohesión (deseable), así como los diferentes tipos de cohesión.

Tabla 2.2. Clasificación de cohesión de módulos.

Tipo de cohesión	Grado de cohesión	Grado de mantenimiento
Por coincidencia	Bajo	Bajo
Lógica	↓	↓
Temporal		
Por procedimientos		
Por comunicaciones		
Secuencial		
Funcional		
Informacional	Alto	Alto

Idealmente, se buscan módulos altamente cohesivos y débilmente acoplados.

2.3. TIPOS DE DATOS

Todos los lenguajes de programación soportan algún tipo de datos. Por ejemplo, el lenguaje de programación convencional Pascal soporta tipos base tales como enteros, reales y caracteres, así como tipos compuestos tales como *arrays* (vectores y matrices) y registros. Los tipos abstractos de datos extienden la

función de un tipo de datos; ocultan la implementación de las operaciones definidas por el usuario asociadas con el tipo de datos. Esta capacidad de ocultar la información permite el desarrollo de componentes de software reutilizables y extensibles.

Un tipo de dato es un conjunto de valores, y un conjunto de operaciones definidas por esos valores.

Un valor depende de su representación y de la interpretación de la representación, por lo que una definición informal de un tipo de dato es:

Representación + Operaciones

Un *tipo de dato* describe un conjunto de objetos con la misma representación. Existen un número de operaciones asociadas con cada tipo. Es posible realizar aritmética sobre tipos de datos enteros y reales, concatenar cadenas o recuperar o modificar el valor de un elemento.

La mayoría de los lenguajes tratan las variables y constantes de un programa como *instancias* de un *tipo de dato*. Un tipo de dato proporciona una descripción de sus instancias que indican al compilador cosas como cuánta memoria se debe asignar para una instancia, cómo interpretar los datos en memoria y qué operaciones son permisibles sobre esos datos. Por ejemplo, cuando se escribe una declaración tal como `float z` en C o C++, se está declarando una instancia denominada `z` del tipo de dato `float`. El tipo de datos `float` indica al compilador que reserve, por ejemplo, 32 bits de memoria, y que operaciones tales como *«sumar»* y *«multiplicar»* están permitidas, mientras que operaciones tales como el *«el resto» (módulo)* y *«desplazamiento de bits»* no lo son. Sin embargo, no se necesita escribir la declaración del tipo `float` —el autor de compilador lo hizo por nosotros y se construyen en el compilador—. Los tipos de datos que se construyen en un compilador de este modo se conocen como *tipos de datos fundamentales (predefinidos)*, y por ejemplo en C y C++ son, entre otros: `int`, `char` y `float`.

Cada lenguaje de programación incorpora una colección de tipos de datos fundamentales, que incluyen normalmente enteros, reales, carácter, etc. Los lenguajes de programación soportan también un número de constructores de tipos incorporados que permiten generar tipos más complejos. Por ejemplo, Pascal soporta registros y arrays.

En lenguajes convencionales tales como C, Pascal, etc., las operaciones sobre un tipo de dato son composiciones de constructores de tipo y operaciones de tipos bases.

Operaciones = Operaciones constructor + Operaciones base

Algunos tipos de constructores incluyen registros, arrays, listas, conjuntos, etcétera.

2.4. ABSTRACCION EN LENGUAJES DE PROGRAMACION

Los lenguajes de programación son las herramientas mediante las cuales los diseñadores de lenguajes pueden implementar los modelos abstractos. La abstracción ofrecida por los lenguajes de programación se puede dividir en dos categorías: *abstracción de datos* (perteneciente a los datos) y *abstracción de control* (perteneciente a las estructuras de control).

Desde comienzo del decenio de los sesenta, en que se desarrollaron los primeros lenguajes de programación de alto nivel, ha sido posible utilizar las abstracciones más primitivas de ambas categorías (variables, tipos de datos, procedimientos, control de bucles, etc.). Ambas categorías de abstracciones han producido una gran cantidad de lenguajes de programación no siempre bien definidos.

2.4.1. Abstracciones de control

Los microprocesadores ofrecen directamente sólo dos mecanismos para controlar el flujo y ejecución de las instrucciones: *secuencia* y *salto*. Los primeros lenguajes de programación de alto nivel introdujeron las estructuras de control: sentencias de bifurcación (`if`) y bucles (`for`, `while`, `do-loop`, etc.).

Las estructuras de control describen el orden en que se ejecutan las sentencias o grupos de sentencia (*unidades de programa*). Las unidades de programa se utilizan como bloques básicos de la clásica descomposición «descendente» En todos los casos, los subprogramas constituyen una herramienta potente de abstracción, ya que durante su implementación el programador describe en detalle cómo funcionan los subprogramas. Cuando el subprograma se llama, basta con conocer lo que hace y no cómo lo hace. De este modo, los subprogramas se convierten en cajas negras que amplían el lenguaje de programación a utilizar. En general, los subprogramas son los mecanismos más ampliamente utilizados para reutilizar código, a través de colecciones de subprogramas en bibliotecas.

Las abstracciones y estructuras de control se clasifican en estructuras de control a nivel de sentencia y a nivel de unidades Las abstracciones de control a nivel de unidad se conoce como *abstracción procedimental*

Abstracción procedimental (por procedimientos)

Es esencial para diseñar software modular y fiable La *abstracción procedimental* se basa en la utilización de procedimientos o funciones, sin preocuparse de cómo se implementan. Esto es posible sólo si conocemos qué hace el procedimiento; esto es, conocemos la sintaxis y semántica que utiliza el procedimiento o función El único mecanismo en Pascal estándar para establecer abstracción procedimental es el subprograma (procedimientos y funciones). La abstracción aparece en los subprogramas debido a las siguientes causas:

- Con el nombre de los subprogramas, un programador puede asignar una descripción abstracta que captura el significado global del subprograma. Utilizando el nombre en lugar de escribir el código, permite al programador aplicar la acción en términos de su descripción de alto nivel, en lugar de sus detalles de bajo nivel.
- Los subprogramas en Pascal proporcionan ocultación de la información. Las variables locales y cualquier otra definición local se encapsulan en el subprograma, ocultándolos realmente, de forma que no se pueden utilizar fuera del subprograma. Por consiguiente, el programador no tiene que preocuparse sobre las definiciones locales; sin embargo, pueden utilizarse los componentes sin conocer nada sobre sus detalles.
- Los parámetros de los subprogramas, junto con la ocultación de la información anterior, permite crear subprogramas que constituyen entidades de software propias Los detalles locales de la implementación pueden estar ocultos, mientras que los parámetros se pueden utilizar para establecer el interfaz *público*.

Otros mecanismos de abstracción de control

La evolución de los lenguajes de programación ha permitido la aparición de otros mecanismos para la abstracción de control, tales como *manejo de excepciones, corrutinas, unidades concurrentes, plantillas (templates)* Estas construcciones son soportadas por los lenguajes de programación basados y orientados a objetos, tales como Modula-2, Ada, C++, Smalltalk o Eiffel.

2.4.2. Abstracción de datos

Los primeros pasos hacia la abstracción de datos se crearon con lenguajes tales como FORTRAN, COBOL y ALGOL 60, con la introducción de tipos de variables diferentes, que manipulan enteros, números reales, caracteres, valores lógicos, etc Sin embargo, estos tipos de datos no podían ser modificados y no siempre se ajustaban al tipo de uno para el que se necesitaban Por ejemplo, el tratamiento de cadenas es una deficiencia en FORTRAN, mientras que la precisión y fiabilidad para cálculos matemáticos es muy alta

La siguiente generación de lenguajes, incluyendo Pascal, SIMULA-67 y ALGOL 68, ofreció una amplia selección de tipos de datos y permitió al programador modificar y ampliar los tipos de datos existentes mediante construcciones específicas (por ejemplo arrays y registros). Además, SIMULA-67 fue el primer lenguaje que mezcló datos y procedimientos mediante la construcción de clases, que eventualmente se convirtió en la base del desarrollo de programación orientada a objetos

La *abstracción de datos* es la técnica de programación que permite inventar o definir nuevos tipos de datos (tipos de datos definidos por el usuario) adecuados a la aplicación que se desea realizar. La abstracción de datos es una técnica muy potente que permite diseñar programas más cortos, legibles y flexibles La

esencia de la abstracción es similar a la utilización de un tipo de dato, cuyo uso se realiza sin tener en cuenta cómo está representado o implementado.

Los tipos de datos son abstracciones y el proceso de construir nuevos tipos se llaman abstracciones de datos. Los nuevos tipos de datos definidos por el usuario se llaman *tipos abstractos de datos*.

El concepto de tipo, tal como se definió en Pascal y ALGOL 68, ha constituido un hito importante hacia la realización de un lenguaje capaz de soportar programación estructurada. Sin embargo, estos lenguajes no soportan totalmente una metodología. La abstracción de datos útil para este propósito no sólo clasifica objetos de acuerdo a su estructura de representación, sino que se clasifican de acuerdo al comportamiento esperado. Tal comportamiento es expresable en términos de operaciones que son significativas sobre esos datos, y las operaciones son el único medio para crear, modificar y acceder a los objetos.

En términos más precisos, Ghezzi indica que un tipo de dato definible por el usuario se denomina *tipo abstracto de dato* (**TAD**) si:

- Existe una construcción del lenguaje que le permite asociar la representación de los datos con las operaciones que lo manipulan.
- La representación del nuevo tipo de dato está oculta de las unidades de programa que lo utilizan [Ghezzi 87]

Las clases en SIMULA sólo cumplían la primera de las dos condiciones, mientras que otros lenguajes actuales cumplen las dos condiciones: Ada, Modula-2 y C++.

Los tipos abstractos de datos proporcionan un mecanismo adicional mediante el cual se realiza una separación clara entre el *interfaz* y la *implementación* del tipo de dato. La implementación de un tipo abstracto de dato consta de:

1 La representación: elección de las estructuras de datos.
2 Las operaciones: elección de los algoritmos.

El interfaz del tipo abstracto de dato se asocia con las operaciones y datos *visibles* al exterior del TAD.

2.5. TIPOS ABSTRACTOS DE DATOS

Algunos lenguajes de programación tienen características que nos permiten ampliar el lenguaje, añadiendo sus propios tipos de datos. Un tipo de dato definido por el programador se denomina *tipo abstracto de datos* (**TAD**) para diferenciarlo del tipo fundamental (predefinido) de datos. Por ejemplo, en Turbo Pascal un tipo *Punto*, que representa a las coordenadas x e y de un sistema de coordenadas rectangulares, no existe. Sin embargo, es posible implementar el tipo abstracto de datos considerando los valores que se almacenan en las variables y qué operaciones están disponibles para manipular estas variables. En esencia, un tipo abstracto de datos es un tipo de datos que consta de datos (estructuras de datos propias) y operaciones que se pueden realizar sobre esos datos.

Un **TAD** se compone de *estructuras de datos* y los *procedimientos* o *funciones* que manipulan esas estructuras de datos.

> **TAD** = Representación (datos) + Operaciones (funciones y procedimientos)

Las operaciones desde un enfoque orientado a objetos se suelen denominar *métodos*.

La estructura de un tipo abstracto de dato (*clase*), desde un punto de vista global, se compone del interfaz y de la implementación (Fig. 2.4).

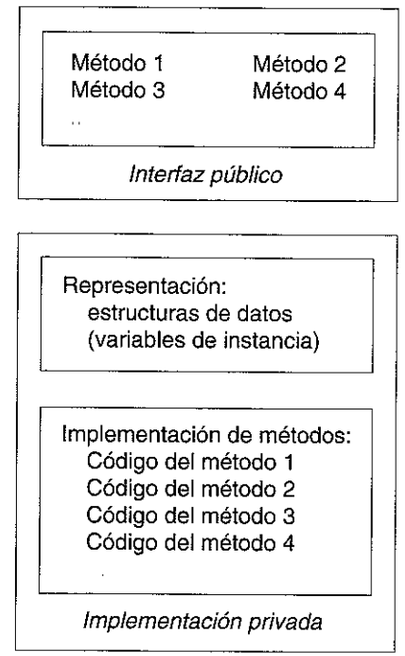


Figura 2.4. Estructura de un tipo abstracto de datos (**TAD**).

Las estructuras de datos reales elegidas para almacenar la representación de un tipo abstracto de datos son invisibles a los usuarios o clientes. Los algoritmos utilizados para implementar cada una de las operaciones de los TAD están encapsuladas dentro de los propios TAD. La característica de ocultamiento de la información del TAD significa que los objetos tienen *interfaces públicos*. Sin embargo, las representaciones e implementaciones de esos interfaces son *privados*.

2.5.1. Ventajas de los tipos abstractos de datos

Un *tipo abstracto de datos* es un modelo (estructura) con un número de operaciones que afectan a ese modelo. Es similar a la definición que daremos en el capítulo siguiente de objeto, y de hecho están unidos íntimamente. Los tipos abstractos de datos proporcionan numerosos beneficios al programador, que se pueden resumir en los siguientes:

1. Permite una mejor conceptualización y modelización del mundo real. Mejora la representación y la comprensibilidad. Clarifica los objetos basados en estructuras y comportamientos comunes.
2. Mejora la robustez del sistema. Si hay características subyacentes en los lenguajes, permiten la especificación del tipo de cada variable, los tipos abstractos de datos permiten la comprobación de tipos para evitar errores de tipo en tiempo de ejecución.
3. Mejora el rendimiento (prestaciones). Para sistemas tipeados, el conocimiento de los objetos permite la optimización de tiempo de compilación.
4. Separa la implementación de la especificación. Permite la modificación y mejora de la implementación, sin afectar al interfaz público del tipo abstracto de dato.
5. Permite la extensibilidad del sistema. Los componentes de software reutilizables son más fáciles de crear y mantener.
6. Recoge mejor la semántica del tipo. Los tipos abstractos de datos agrupan o localizan las operaciones y la representación de atributos.

2.5.2. Implementación de los TAD

Los lenguajes convencionales, tales como Pascal, permiten la definición de nuevos tipos y la declaración de procedimientos y funciones para realizar operaciones sobre objetos de los tipos. Sin embargo, tales lenguajes no permiten que los datos y las operaciones asociadas sean declaradas juntos como una unidad y con un solo nombre. En los lenguajes en el que los módulos (TAD) se pueden implementar como una unidad, éstos reciben nombres distintos:

Turbo/Borland Pascal	*unidad, objeto*
Modula-2	*módulo*
Ada	*paquete*
C++	*clase*
Java	*clase*

En estos lenguajes se definen la *especificación* del TAD, que declara las operaciones y los datos ocultos al exterior, y la *implementación*, que muestra el código fuente de las operaciones y que permanece oculto al exterior del módulo.

Las ventajas de los TAD se pueden manifestar en toda su potencia, debido a que las dos partes de los módulos (*especificación* e *implementación*) se pueden compilar por separado mediante la técnica de compilación separada (*«separate compilation»*).

2.6. TIPOS ABSTRACTOS DE DATOS EN TURBO PASCAL

Una *pila* es una de las estructuras de datos más utilizadas en el mundo de la compilación. Una *pila* es un tipo de dato clásico utilizado frecuentemente para introducir al concepto de tipo abstracto de datos; es una lista lineal de elementos en la que los elementos se añaden o se quitan por un solo extremo de la lista. La *pila* almacena elementos del mismo tipo y actúan sobre ella las operaciones clásicas de `Meter` y `Sacar` elementos en dicha *pila*, teniendo presente la estructura lógica **LIFO** (*último en entrar, primero en salir*).

Pila

Estructura de datos

Almacena una serie de elementos del tipo *elemento*. La *pila* está inicialmente vacía y los elementos se meten o sacan en la *pila* por el mismo extremo.

Operaciones (*Procedimientos que actúan sobre la estructura de datos*)

```
Meter
Sacar
Crear
Destruir
PilaVacía
```

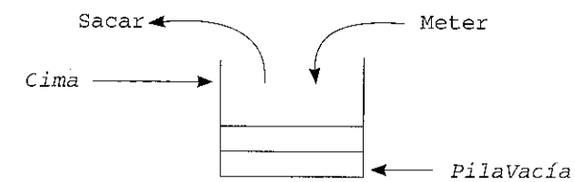


Figura 2.5. Estructura de datos `Pila`.

En Turbo Pascal, los **TAD** se implementan mediante estructuras tipo *unidad*. Recordemos que una **unidad** es una biblioteca de funciones y procedimientos que pueden ser utilizados por cualquier programa con la condición de incluir el nombre de la unidad en la cláusula `uses` de su sintaxis.

```
unit <nombre unidad>;
interface
  <clausula uses>
  <constantes, tipos y variables publicas>
  <cabeceras de procedimientos y funciones publicas>
implementation
  <clausula uses>
  <constantes, tipos y variables privadas>
  <procedimientos/funciones privadas y cuerpos de
   procedimientos/funciones publicas>
begin
  <secuencia de sentencias para inicializacion>
end.
```

La implementación de una *pila* con capacidad para 1 000 elementos del tipo entero es:

```
unit Pila;
interface
  const
    MaxPila = 1000;
  type
    TipoElemento = integer;
    ListaElementos = array [1..MaxPila] of TipoElemento;
    tipo =  record
              Elems : ListaElementos;

                Cima : integer;
              end;

  procedure Crear (var S:tipo);
  (* S se inicializa y se limpia o vacía *)

  procedure Destruir (var S:tipo);
  (* Se libera memoria asignada a S *)
  (* S no está inicializada         *)

  procedure Meter (var S:tipo; Item: tipoElemento);
  (* Se añade un elemento a la cima de la pila *)

  procedure Sacar (var S:tipo; Item:tipoElemento);
  (* quitar un elemento de la pila *)

  procedure PilaVacia (var S:tipo):boolean;
  (* devuelve true si S es vacia; false en caso contrario *)

  implementation
  procedure Crear (var S:tipo);
  begin
    S.Cima := 0;
  end;

  procedure Destruir (var S:tipo);
  begin
    S.  (* no hace nada *)
  end;

  procedure Meter (var S:tipo; Item:TipoElemento);
  begin
    S Cima := S.Cima + 1;
    S Elems[S.Cima] := Item;
  end;

  procedure Sacar (var S:tipo; var Item:TipoElemento);
  begin
    Item := S Elems[S Cima];
  end;

  procedure PilaVacia (var S:tipo):boolean;
  begin
    PilaVacia := (S.Cima=0);
  end;
end.
```


Una vez que se ha implementado el tipo de dato *Pila*, éste puede ser utilizado en cualquier programa con tal de invocar en la cláusula `uses` a dicho tipo de dato *Pila*.

2.6.1. Aplicación del tipo abstracto de dato *Pila*

El siguiente programa lee una secuencia de enteros, uno por línea, tales como estos:

```
100
4567
-20
250
```

y genera la misma secuencia de números en orden inverso (los saca de la pila).

```
250
-20
4567
100
```

Esta tarea se consigue metiendo números en la pila y a continuación vaciando dicha pila.

```
program Numeros;

uses
  Pila;
var
  S:Pila.Tipo;

procedure LeeryAlmacenarNumeros(var NumPila:Pila.Tipo);
  var
    Aux:Pila TipoElemento;
begin
  while not eof do
    begin
      readln(Aux);
      Pila Meter(NumPila, Aux);
  end;
end;

procedure VerNumeros(var NumPila: Pila.Tipo);
  var
    Aux: Pila.TipoElemento;
begin
  while not Pila Pilavacia(NumPila) do
  begin
    Pila.Sacar(NumPila, Aux);
    WriteLn(Aux);
  end;      (* while *)
end;        (* VerNumeros *)
```

```
begin       (* programa principal *)
  Pila.Crear(S)
  LeeryAlmacenarNumeros(S);
  Pila.Destruir(S);
end.        (* fin de principal *)
```

Al igual que se ha definido el tipo *Pila* con estructuras estáticas tipo *array*, se podía haber realizado con estructuras dinámicas tales como listas enlazadas. De igual modo se pueden implementar otros tipos de datos abstractos, tales como, por ejemplo, las colas que se utilizan en muchas aplicaciones: sistemas operativos, sistemas de comunicaciones, etc.

2.7. TIPOS ABSTRACTOS DE DATOS EN MODULA-2

Modula-2, el segundo lenguaje inventado por Niclaus Wirth (el diseñador de Pascal), soporta esencialmente las características de:

- *Compilación separada de módulos*
- *Abstracción de datos.*

2.7.1. Módulos

La construcción que proporciona las características citadas anteriormente se denomina *módulo*. Sintácticamente, el módulo posee la misma estructura que las uniones en Pascal (versiones USCD y Turbo de Borland).

Un programa en Modula-2 proporciona una colección de módulos biblioteca que pueden ser ampliados por el programador escribiendo sus propios módulos biblioteca. Los módulos programa y biblioteca pueden contener anidados módulos más pequeños ocultos del resto del programa, que se denominan *módulos locales*.

Los módulos pueden compilarse por separado y facilitan a los programadores el desarrollo de bibliotecas de código reutilizable y la construcción de grandes programas.

El formato básico de un módulo es:

```
MODULE identificador_mod;

  <declaraciones>

BEGIN

  <sentencias>

END identificador_mod.
```

En Modula-2, la especificación y la implementación de un tipo abstracto de datos se compilan por separado en un *módulo de definición* y en un *módulo de implementación*. Esta separación es consistente con el principio fundamental del

TAD: los tipos de datos y las definiciones de las operaciones asociadas (que manipulan) con esos datos se agrupan en una parte (módulo de definición: `DEFINITION MODULE`) y el cuerpo de las operaciones asociadas en otra parte (módulo de implementación: `IMPLEMENTATION MODULE`).

2.7.2. Módulos locales

Un módulo anidado (declarado) dentro de otro módulo o procedimiento se denomina *módulo local*. Las declaraciones de los módulos locales aparecen en la misma posición que las declaraciones de otras entidades.

Los módulos locales tienen propiedades similares a los módulos biblioteca, incluyendo la capacidad de importar y exportar entidades.

La abstracción de datos es posible realizarla en Modula-2 a través de la construcción módulo que proporciona el encapsulamiento y ocultamiento de los datos. La visibilidad y ocultamiento de los datos se puede obtener mediante tipos *opacos* (ocultos o privados) o *transparentes* (públicos).

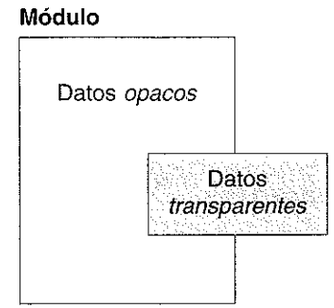


Figura 2.6. Ocultamiento de datos.

2.7.3. Tipos opacos

Un *tipo opaco* es, como su nombre indica, uno cuya estructura se oculta de sus usuarios. Un tipo opaco se puede exportar desde un módulo y se puede utilizar para declarar objetos, tales como variables, elementos de arrays y campos de registros, en módulos que los importan.

El uso de un tipo opaco en el módulo de definición soporta el principio de ocultamiento: los detalles de la implementación del tipo de dato (`Pila` en el ejemplo) están separados de su declaración y están ocultos en el módulo de implementación, junto con la implementación de las operaciones.

El *tipo opaco* se define en el módulo de definición en la sección **`TYPE`**. Un módulo cliente que importa un tipo opaco se puede utilizar de igual modo que cualquier tipo en Modula-2. Dado que los módulos clientes no conocen la estructura del tipo, las únicas operaciones que pueden realizar sobre variables del tipo dado sin asignaciones y pruebas de igualdad y desigualdad.

Cuando la parte de definición de un módulo biblioteca define un tipo opaco, la parte de implementación del módulo debe incluir una declaración completa del tipo. La declaración completa de un tipo opaco se debe especificar como un tipo abstracto de dato *Pila*, y es el siguiente:

```
DEFINICION MODULE PILA;
  TYPE Pila;        (* Tipo opaco *)
  PROCEDURE Inicializar(VAR s: Pila);
  PROCEDURE Meter(VAR s: Pila) : INTEGER;
  PROCEDURE Sacar( s: Pila) : INTEGER;
  PROCEDURE Pilavacia(s: Pila) : BOOLEAN;
  PROCEDURE Pilallena(s: Pila) : BOOLEAN;
END PILA.
```

La declaración de un tipo opaco se hace en el módulo de implementación.

```
IMPLEMENTATION MODULE PILA;

CONST
  LongPila = 100;
TYPE RegistroPila = RECORD
    PilaArray : ARRAY [1..LongPila] OF Char;
    CimaDePila: [0..LongPila + 1]
END;
Pila = POINTER TO RegistroPila;
```

2.7.4. Tipos transparentes

Un módulo de definición puede contener la declaración de —estructura— un tipo de dato, en cuyo caso el tipo se denomina *transparente* o *público*, y se puede acceder a él y a sus campos desde cualquier módulo cliente.

Veamos cómo se escriben los módulos de definición y de implementación de un tipo abstracto de datos `PilaCaracteres`. Se puede definir el tipo de dato mediante arrays o punteros; en este caso consideremos la implementación mediante arrays. El módulo `PilaCaracteres` permite la exportación del tipo de dato `PilaCar`, que tiene carácter transparente.

```
DEFINITION MODULE PilaCaracteres;
(* exporta tipo PilaCar (pila de caracteres)
   y procedimientos para manipulación de variables de tipo
   PilaCar *)
  CONST TamanyoPila = 100:
    (* máximo tamaño de un tipo PilarCar *)
  TYPE PilaCar = RECORD
      ArrayPila : ARRAY [1..TamanyoPila] OF CHAR;
      CimaPila : [0..TamanyoPila + 1]
        END;
    (* exportación transparentes de PilaCar *)
  PROCEDURE Meter(VAR s: PilaCar; ch: CHAR);
    (* empuja ch en la pila s; termina programa si pila
      está llena *)
```



```
  PROCEDURE Sacar(VAR s: PilaCar; VAR ch: CHAR);
    (* almacena elemento superior de la pila s en ch, a
      continuación saca s; termina programa si pila está
      vacía *)
  PROCEDURE EsVacia(s: PilarCar): BOOLEAN;
    (* devuelve TRUE si s está vacía *)
END PilaCaracteres
```

Este módulo tiene algunos inconvenientes. Si se decide incrementar el tamaño máximo de la pila, se deben recompilar todos los módulos que importan `PilaCaracteres`, incluso cuando cambie la representación de `PilaCar`. Estos inconvenientes y otros más no citados se pueden mejorar haciendo `PilaCar` de tipo opaco, permitiendo que `PilaCaracteres` exporte `PilarCar` sin proporcionar ninguna información a los módulos cliente sobre la estructura real de la variable `PilaCar`. Este enfoque resuelve nuestros problemas: se puede cambiar la representación de `PilaCar` en cualquier momento sin cambiar los módulos cliente y se asegura que los clientes no pueden modificar variable `PilaCar`, excepto a través de procedimiento exportados por `PilaCaracteres`.

2.7.5. Una versión del tipo abstracto de dato *Pila* con datos opacos

Una versión mejorada del módulo de definición de `PilaCaracteres` con `PilaCar` definida como un tipo opaco es:

```
DEFINITION MODULE PilaCaracteres;
(* exporta tipo PilaCar (pila de caracteres)
  y procedimientos para manipulación de variables de tipo
  PilaCar *)

  TYPE PilaCar; (* exportación opaca *)
  PROCEDURE Inicializar(VAR s: PilaCar);
    (* crea una pila s y la hace vacía *)
  PROCEDURE Meter(VAR s: PilaCar; ch: CHAR);
    (* empuja ch en pila s; termina programa si pila está
       llena *)
  PROCEDURE Sacar(VAR s: PilaCar; VAR ch: CHAR);
    (* almacena elemento superior de pila s en ch, a
      continuación saca s; termina programa si pila está
      vacía *)
  PROCEDURE EsVacia(s: PilaCar) : BOOLEAN;
    (* devuelve TRUE si s es vacía, FALSE en caso
      contrario *)
END PilaCaracteres
```

Los módulos que importan `PilaCar` pueden declarar variables de tipo `PilaCar`, pero el único medio para manipular estas variables es a través de los procedimientos `Inicializar`, `Meter`, `Sacar` y `EsVacia`.

La especificación completa del tipo `PilaCar` debe aparecer en la sección de implementación de `PilaCaracteres`. Dado que `PilaCar` debe ser un tipo

puntero, se declara como un puntero a un registro que contiene los campos ArrayPila y CimaPila. El procedimiento Inicializar asigna espacio para estos registros y después fija el campo CimaPila a 0. El módulo de implementación es:

```
IMPLEMENTATION MODULE PilaCaracteres;
  FROM Storage IMPORT ALLOCATE
  FROM IO IMPORT WrStr, WrLn;
  CONST TamanyoPila = 100;
    (* tamaño máximo de una PilaCar *)
  TYPE PilaReg =
    RECORD
      ArrayPila : Array[1 .. TamanyoPila] OF CHAR;
      CimaPila : [0 .. TamanyoPila + 1]
    END;
    PilaCar = POINTER TO PilaReg;

  PROCEDURE Inicializar(VAR s: PilaCar);
    (* crea una pila y la hace vacía *)
  BEGIN
    ALLOCATE (s, SIZE (PilaREg);
    s^.CimaPila := 0
  END Inicializar;

  PROCEDURE Meter(VAR s: PilaCar; ch: CHAR);
    (* empuja ch en la pila s; termina programa si pila llena *)
  BEGIN
    INC (s^.cimaPila);
    IF s^ cimaPila < tamanyoPila THEN
      WrStr( Desbordamiento Pila");
      WrLn;
      HALT
    END;
  S^.ArrayPila [s^.cimaPila] := ch
  END Meter;

  PROCEDURE Sacar(VAR s: PilaCar; VAR ch: CHAR);
    (* almacena elemento superior de la pila s en ch,
      después saca s; termina programa si pila está vacía *)
  BEGIN
    SF s^.cimaPila = 0 THEN
      WrStr( Desbordamiento negativo pila");
      WrLn;
      HALT
    END;
    ch := s^.ArrayPila[s^ cimaPila];
    DEC(s^.cimaPila)
  END Meter;

  PROCEDURE EsVacia(s: PilaCar) : BOOLEAN;
    (* devuelve TRUE si s está vacía *)
  BEGIN
    RETURN s^ cimaPila = 0
  END EsVacia;
END PilaCaracteres
```


Una aplicación Inversa utiliza un módulo programa que lee una cadena de caracteres y visualiza su inversa. Este programa es fácil de escribir si se utiliza una pila para almacenar la cadena. A medida que lee caracteres, se van empujando a la pila, deteniéndose cuando se alcanza el final de la cadena. Inversa saca los caracteres de la pila hasta que se vacíe, escribiendo a continuación cada carácter a medida que se saca el carácter.

```
MODULE Inversa;
  (* invierte una cadena escrita por el usuario *)

  FROM IO IMPORT RdKey, WrChar, WrStr, WrLn;
  FROM PilaCaracteres IMPORT PilaCar, Inicializar, Meter,
      Sacar, EsVacia;

  CONST rc = 15c;       (* retorno de carro *)

  VAR  s : PilaCar;
      car: CHAR

  BEGIN
    Inicializar(s);     (* vacia pila s *)

    WrStr("Introduzca cadena:");
    car := RdKey();
    WHILE car # rc Do
      WrChar (car);
      Meter(s,car);
      car := RdKey
    END;
    WrLn;

    WrStr("Cadena inversa es: );
    WHILE NOT EsVacia (s) DO
      Sacar(s,car);
      WrChar(car)
    END;
    WrLn;
END Inversa.
```

2.7.6. Otra aplicación del TAD *Pila*

Diseñar un TAD Pila que disponga de un tipo opaco (su representación sólo será visible en el módulo de implementación; no serán visibles a los módulos clientes). Los tipos opacos se restringen a punteros y, en consecuencia, es preciso asignar memoria dinámicamente antes de utilizarlos. Por consiguiente, hay que incluir un procedimiento de inicialización de modo que el usuario pueda crear una pila para su uso.

```
definition module pilamod;
  type pilatipo;
  procedure vacía (stk:pilatipo):boolean;
```

```
  procedure meter (var stk:pilatipo; elemento:integer);
  procedure quitar (var stk:pilatipo);
  procedure cima (stk:pilatipo):integer;
end pilamod.

implementation module pilamod;
  from InOut import WriteString, WriteLn;
  from Storage import allocate
  const max = 100;

  type pilatipo = pointer to
                  record
                    lista: array[1..max]of integer;
                    cimasub: [0..max]
                  end;
  procedure vacia (stk:pilatipo):boolean;
    begin
      return stk^.cimasub = 0
    end vacía;

  procedure meter (var stk:pilatipo; elemento:integer);
    begin
      if stk^.cimapila = max then
        WriteString("error - desbordamiento pila");
        WriteLn;
      else
        Inc (stk^.lista[stk^ cimapila] := elemento
    end
  end meter;

  procedure quitar (stk:pilatipo):integer;
    begin
      if vacia (stk) then
        WriteString("Error - desbordamiento negativo");
        WriteLn
      else
        dec (stk^.cimasub)
      end (* fin de if vacia *)
  end quitar;

  procedure cima (stk:pilatipo):integer;
    begin
      if vacia (stk) then
        WriteString("Error - desbordamiento negativo");
        WriteLn;
      else
        return stk^.cimasub[stk^.cimasub]
      end
  end cima;

  procedure crear (var stk:pilatipo);
    begin
      new (stk);
      stk^.cimasub := 0
  end crear;

end pilamod.
```


El siguiente código crea e inicializa una pila, mete dos valores, 42 y 27, luego saca el 27 y deja el valor 42.

```
module usodepila;
  from InOut import WriteLn, WriteInt, WriteString;
  frompilamodimportpilatipo vacia,meter,quitar,cima crear;

  var pila:pilatipo;
  var aux:integer;
  ...
  begin

    crear (pila);
    meter (pila, 42);
    meter (pila, 27);
    quitar (pila);
    aux := cima (pila);

end usodepila.
```

2.8. Tipos abstractos de datos en Ada

Ada proporciona unidades de programa: construcciones que están definidas con un nombre y son autónomas en el diseño de un programa. Las unidades en Ada son:

- **Subprogramas:** procedimientos y funciones análogos a sus homónimos en Pascal.
- **Tareas:** unidades de programa diseñadas para ejecutarse concurrentemente, que se identifican como colecciones estáticas de código y procesos.
- **Paquetes:** construcciones que soportan abstracción de datos y son la base de componentes software
- **Unidades genéricas:** unidades que permiten la creación de tipos genéricos o parametrizados. Son plantillas que sirven para construir subprogramas y paquetes y que constituye el mecanismo principal para construir componentes de software reutilizable.

Ada es un lenguaje que soporta el tipo abstracto de datos identificado como módulo en una entidad denominada paquete. Un *paquete* representa en Ada la idea de la encapsulación, que es la clave para la abstracción de datos.

El *paquete* en Ada representa el nivel más alto de abstracción del programa y actúa como un módulo. La ocultación de datos se realiza a través del uso de tipos `private` (privado) o `limited private` (privado limitado). Las operaciones de asignación y verificación de igualdad son las únicas operaciones que están definidas en el lenguaje para tipos privados. Para tipos privados limitados no existen operaciones predefinidas proporcionadas por el lenguaje. El programador debe prever todas las operaciones soportadas, incluyendo asignación y verificación de igualdad.

Ada divide el paquete en dos construcciones sintácticamente independientes: la *especificación* del paquete y el *cuerpo* del paquete. El contenido de la especificación es muy similar al módulo de definición (DEFINITION MODULE) en Modula-2. El cuerpo se corresponde al módulo de implementación (IMPLEMENTATION MODULE).

Especificación del paquete

La especificación del paquete tiene la siguiente sintaxis:

```
package <nombrepaquete> is
  <items declarativos>
  [private
    <items declarativos>]
end <nombrepaquete>
```

La especificación tiene dos partes:

1. *Visible.* Los componentes definidos en la parte visible son accesibles fuera de la especificación del paquete.
2. *Privada.* La parte privada de la especificación del paquete no es accesible a los clientes del paquete.

Cuerpo del paquete

El cuerpo del paquete contiene las definiciones de los componentes que están ocultas a los clientes del paquete. Un cuerpo del paquete se puede definir en un módulo separado de la especificación del paquete. La sintaxis es:

```
package body <nombrepaquete> is
<declaraciones>
<implementación subprogramas>
[manejador de excepciones]
begin <inicializaciones>
end <nombrepaquete>
```

Así, el tipo abstracto Pila más simple (sólo el procedimiento METER y la función SACAR) se escribiría así:

```
package TADPila is --especificación
type Pila is private;  | Utilizado para declarar
                       | instancia de la clase
  procedure METER(x:integer, p: in out Pila);
  procedure SACAR (p: in out Pila  x: out integer);
private
  max:constant := 100;
  type Pila is record                    |
    P: array(1 .. Max) of integer;       | Interfaz privado
    Cima:integer range 0 .. max := 0;    |
end record;
end Pila;
```



```
package body TADPila is --cuerpo

  procedure Meter(x:integer, p: in out Pila) is
  begin
    Cima := Cima+1;
    P(Cima) := x;
  end Meter;

  procedure Sacar (p: in out Pila, x: out integer) is
  begin
    Cima := Cima-1;
    x! = P(Cima+1);
  end Sacar;
begin inicializaciones
  Cima := 1;
end TADPila;
```

Los paquetes permiten ocultar a los usuarios de los mismos los objetos internos de los mismos. Los tipos privados nos permiten ocultar a los usuarios de los mismos los detalles de construcción de los tipos. Ada soporta dos tipos de datos: *privados* y *privados limitados*.

2.8.1. Tipos privados

La parte de la especificación que está antes de la palabra reservada `private` es la parte visible y la información está disponible fuera del paquete. Después de `private` se tienen que dar los detalles de los tipos declarados como privados y dar los valores iniciales de las constantes definidas correspondientes.

Las características adicionales en la especificación de un paquete que exporta un tipo privado son:

- La declaración, en la parte pública de la declaración, de uno o más tipos privados. Cada declaración debe ser de la forma:

  type identificador is private

- Una sección a continuación de la palabra reservada `private` y terminada en `end`, al final de la especificación, en la que se declaran los detalles de implementación del tipo o tipos privados.

Así, por ejemplo, la sección de especificación de un tipo de datos abstracto Pilas es:

```
package Pilas is
  --comentarios
  --un paquete que exporta un tipo abstracto Pila de enteros
  type Pila_ent is private;
  procedure Iniciar (S:in out Pila_ent);
  procedure Meter (S:in out Pila_ent; Inst:integer);
  procedure Sacar (S:in out Pila_ent, x: out integer);
  function Cima (P:Pila_ent) return integer;
  function Es_vacía (S:Pila_ent) return boolean;
  function Es_llena (S:Pila_ent) return boolean;
```

```
  private
    -- definiciones del tipo Pila
      limite_pila: constant := 100;
      Subtype rango_pila is integer range 1 .limite_pila;
    type Pila_ent is
      record
        Pila_array:array (rango_pila) of integer;
        Cima:rango_cima;
      end record;
end Pilas;
```

Las únicas operaciones disponibles en objetos de tipo privado son igualdad y asignación

2.8.2. Tipos privados limitados

Las operaciones disponibles para un tipo privado pueden restringirse completamente a las especificadas en la parte visible del paquete. Esto se consigue declarando el tipo como limitado, al mismo tiempo que privado, del siguiente modo:

```
type t is limited private;
```

La ventaja de hacer que un tipo privado limitado es que el programador del paquete tiene un control completo sobre los objetos del tipo. Se puede vigilar la copia de recursos, etc. Así, un tipo `Pila` definido como un tipo privado (`Private`) limitado:

```
package Pila is
  type Pila_ent is limited Private;
  procedure Iniciar (S:in out Pila_ent);
  procedure Meter (S: in out Pila_ent; valor_int:integer);
  procedure Sacar (S: in out Pila_ent, x: out integer);
  procedure Cima (S: Cima) return integer;
  function Es_vacia (S: Pila_ent) return boolean;
  function Es_llena (S: Pila_ent) return boolean;
  function "=" (izda, dcha: Pila_ent) return boolean;
private
  Max : constant := 100;
  type Vector_Entero is array (Integer range <>) of integer;
  type Pila_ent is
    record
      P: Vector_Entero(1..Max);
      Cima:Integer range 0..Max := 0;
    end record;
end;
```

Cuando una variable se declara como tipo privado limitado, no están disponibles ni las operaciones predefinidas de igualdad ni de asignación. Si se requieren, se puede definir una nueva función =, que sólo compara los elementos significativos, pero el operador de asignación := no se puede redefinir, y en consecuencia, si se requiere una operación de asignación, habrá que declarar un procedimiento especial *asignar*.

2.9. TIPOS ABSTRACTOS DE DATOS EN C

El lenguaje C proporciona algún soporte para abstracción de datos, aunque ésta es independiente de la implementación particular. La única estructura de programa definida en el lenguaje es la *función*, que es un subprograma que puede poseer parámetros de sólo lectura y generalmente devuelve un valor mediante la sentencia **`return`**. Las implementaciones estándar de C reconocen una estructura de más alto nivel —tipo **`file`**—, que es simplemente un archivo que contiene código fuente. El archivo en C es la unidad de compilación; al igual que el módulo, los archivos son unidades de compilación. De este modo, la mayoría de los programas se componen de diferentes archivos compilados separadamente. Sin embargo, esta propiedad se convierte normalmente en un inconveniente, ya que C, al contrario que sus rivales Modula-2 y Ada, no es capaz de realizar verificación de tipos durante el proceso de compilación separada: es decir, cuando se disponen de varios archivos, no existen fórmulas claras que permitan definir dicha verificación.

El interfaz del módulo utilizado es la declaración del *archivo de cabecera (.h)*, que mezcla en el archivo fuente merced al preprocesador aquellas declaraciones de funciones incluidas en los citados archivos. No existe en C ninguna estructura para encapsular datos y funciones en una sola entidad.

Normalmente, un programa C se organiza en uno o más *archivos fuente* o módulos. Cada archivo tiene una estructura similar, con comentarios, directivas de preprocesador, declaraciones de variables y funciones, y sus definiciones. Normalmente se situará cada grupo de variables y funciones relacionadas en un único archivo fuente. Algunos archivos son simplemente un conjunto de declaraciones que se utilizan en otros archivos a través de la directiva `#include` del preprocesador C. Estos archivos, ya citados, se conocen como *archivos de cabecera* y sus nombres terminan con la extensión *.h* y constituyen la *especificación* del TAD. Las operaciones indicadas en la sección de especificación han de ser implementadas; la opción más segura es un archivo independiente con un sufijo o extensión *.c*. Como C no soporta el tipo de dato *clase* que implementa en una única unidad los datos y operaciones que manipulan los datos, la implementación del tipo abstracto de datos se realiza mediante `typedef`, los datos con `struct` y la implementación de las operaciones con un conjunto de funciones.

Un ejemplo de un TAD que representa un punto en un sistema de coordenadas rectangulares podría ser:

especificación

```
# include "punto.L'
# define CANEVAS_LONG 10000
typedef short int Coord;
```

implementación

```
# include <math h>
# include "punto.h
# define norma(x) (.....)
```

```
typedef struct {

   Coord x, y;
} Punto;

double Distancia (Punto p1,
Punto p2);
/* ... restantes funciones */
```

```
Punto Distancia (Punto p1, Punto p2)

}
   long dx=p1.x - p2.x
        dy=p1.y - p2.y;
return sqrt (dx*dx+dy*dy);
}
/* ... restantes implementaciones */
```

2.9.1. Un ejemplo de un tipo abstracto de datos en C

Definir un tipo abstracto de datos complejo en C que represente números complejos (*datos* y *operaciones* sobre números complejos: parte real, parte imaginaria, sumar, restar, igualdad, multiplicar y dividir)

Un número complejo z tiene parte real y parte imaginaria:

$$z = x + iy$$

```
/* tipo abstracto de datos en C */
/* archivo: complex.h */

#ifndef COMPLEX
#define COMPLEX

typedef struct {
  float r, i;
} complejo;

complejo nuevo_complejo(float x, float y);
float real(complejo c);
float imag(complejo c);
complejo sumar(complejo a, complejo b);
complejo restar(complejo a, complejo b);
int igual(complejo a, complejo b);
complejo multiplicar(complejo a, complejo b);
complejo dividir(complejo a, complejo b);

#endif
```

Los cuerpos de las funciones se implementan en el archivo *complex.c*:

```
/* FICHERO: complex.c */

#include "complex.h"

complejo nuevo_complejo(float x, float y) {
  complejo c;
  c.r = x;
  c.i = y;
  return c;
}
```



```
float real(complejo c) {
  return c.r;
}

float imag(complejo c) {
  return c.i;
}

complejo sumar(complejo a, complejo b) {
  complejo c;
  c.r = a.r + b.r;
  c.i = a.i + b.i;

  return c;
}

complejo restar(complejo a, complejo b) {
  complejo c;
  c.r = a.r - b.r;
  c.i = a.i - b.i;
  return c;
}

int igual(complejo a, complejo b) {
  return(a.r == r.r && a.i == b.i);
}

complejo multiplicar(complejo a, complejo b) {
  complejo c;
  c.r = a.r * b.r - a.i * b.i;
  c.i = a.r * b.i - a.i * b.r;
  return c;
}

complejo dividir(complejo a, complejo b) {
  complejo c;
  float denom = b.r * b.r + b.i * b.i;
  c.r = (a.r * b.r + a.i * b.i)/denom;
  c.i = (a.i * b.r - a.r * b.i)/denom;
  return c;
}
```

Una vez definido el tipo de dato complejo, se puede invocar a los mismos dentro de un programa principal.

```
/* FICHERO: principa.c */

#include "complex.h"

int main() {
  complejo x, y, z;

  x = nuevo_complejo(7.0, 7.0);
  y = nuevo_complejo(5.0, 5.0);

  printf("x = (%f, %f)\n",real(x), imag(x));
```

```
    printf('y = (%f, %f)\n' real(y), imag(y));

    z = sumar(multiplicar(x, y), x);
    printf('(x+y)*x = (%f, %f)\n",z);

    return 0;
}
```

C no permite definir los signos + y − para trabajar con tipos definidos por el usuario, cosa que sí podrá realizarse con C++ mediante la propiedad de sobrecarga de operadores.

2.10. TIPOS ABSTRACTOS DE DATOS EN C++

En C++ el equivalente del paquete o módulo se denomina **clase**, y es el tipo de dato que soporta la abstracción de datos. Una clase es un tipo de dato que incluye datos y operaciones que manipulan esos datos. La clase es la entidad principal de los lenguajes de programación orientados a objetos (C++, Smalltalk, Eiffel, etc.).

Una definición de una clase en C++ consta de *miembros dato* y *funciones miembro*, a través de las cuales se puede acceder a los detalles internos de la clase. Una clase se define como una estructura en C (`struct`), con la diferencia de que puede contener tres secciones: *pública, privada* y *protegida*, y que puede definirse con las palabras reservadas `struct` y `class`.

Así, la definición de una clase (TAD) pila de caracteres (`pila_carac`) en C++ es su especificación, y su sintaxis es:

```
class pila_carac {
  char lista[100];
  int cima_de_la_pila;
public:
  pila_carac() {cima_de_la_pila = -1};
  void meter(char);
  void sacar(char *);
  char cima();
};
```

La nueva clase `pila_carac` contiene las operaciones `meter`, `sacar` y `cima`. Las clases pueden incluir otras funciones miembro con igual nombre que la clase y que se denominan **constructores** y **destructores**, cuya utilidad se verá posteriormente (Capítulo 6).

El encapsulamiento en C++ se consigue declarando todos los datos como privados (en la sección privada, por defecto las sentencias que vienen a continuación de la llave de apertura hasta la primera cláusula `public` o `protected`). A la sección privada sólo se puede acceder mediante las funciones miembro de la clase.

La sección pública es accesible (visible) a cualquier otra clase y sus datos pueden ser modificados por funciones miembros externas a la clase.

La implementación de la clase (los cuerpos de la función) se pueden declarar dentro de una definición de la clase, pero es más frecuente incluir los prototipos de la función en la especificación y declarar los cuerpos de la función separadamente, aprovechando la propiedad de compilación separada que posee el lenguaje C++. Además, cada parte se compila en un archivo: la especificación con la extensión `.h` y la implementación con la extensión `.cpp`.

En el caso de la citada clase `Pila`, los cuerpos de las funciones se declaran y compilan por separado mediante el operador de resolución de ámbito (`::`), que permite asociar los nombres de las funciones miembro con la clase correspondiente. Así, los códigos fuente serían:

```
void pila_carac::meter(char x)
{lista[++cima_de_la_pila] = x;}

void pila_carac::sacar(char * x)
{* x = lista[cima_de_la_pila--];}

char pila_carac::cima()
{return lista[cima_de_la_pila];}
```

Los usuarios de la información declarada en una clase se conocen como **clientes**. Las variables de la clase `pila_carac` se pueden declarar y manipular por un cliente de la forma siguiente:

```
main()
{
  pila_carac a, b;     // se crean dos objetos de la clase
  char ch;

  a meter('f'); b meter('g');
  ch = a.cima();
  ....
}
```

2.10.1. Definición de una clase *Pila* en C++

La clase `Pila` se define en el archivo de cabecera `pila.h`, de modo que podrá ser utilizado por otros programas. La implementación de la pila utiliza un array `items`, que contiene los elementos de la pila, y un índice `cuentas`, que contiene cuantos elementos existen realmente en la pila. Las funciones que manipulan la pila son `meter`, `sacar`, `cima` y `vacía`. Estas operaciones se desea que estén disponibles a los usuarios de la pila y, por consiguiente, se deben definir *públicas* (mediante la palabra reservada `public`), lo que significa que cualquier usuario de la pila puede llamar a las diferentes funciones miembro que implementan las operaciones de la pila.

Con el tipo `struct`, el acceso por omisión en público (`public`) para todos sus miembros. En la definición de `pila` existe un prototipo especial denominado `pila()`. Cualquier función miembro con el mismo nombre que el tipo dato es un *constructor*. Un constructor se llama automáticamente siempre que crea-

mos una *instancia* de un tipo de dato particular Su trabajo es inicializar los campos dato dentro de un objeto, ahorrando al programador la molestia de llamar a una rutina de inicialización específica. De modo similar, se puede especificar un *destructor*, una función miembro con el mismo nombre que la clase, con un símbolo ~ delante

```
//archivo pila.h
//definición de una pila, con operaciones

const int MAXPILA = 100;     //tamaño por defecto de la pila

struct pila
{
  private:
    int cuenta;              //número de elementos de la pila
    int items[MAXPILA];      //definición de la pila
  public:
    pila();                  //inicializar pila, constructor
    void meter(int item);    //meter elementos en la pila
    void sacar();            //quitar elementos de la pila
    int cima();              //devuelve elemento cima de la pila
    int vacía();             //¿la pila está vacía?
};
```

Las operaciones de la pila implementadas en el cuerpo de la clase:

```
//definición de la pila: pila.cpp
//
#include "pila h"

pila::pila()                 //inicializa la pila a vacía
{
  cuenta = 0;
}

void pila::meter(int item)   //meter un elemento en la pila
{
  items[cuenta++] = item;
}

void pila::sacar()           //quitar elemento superior de la
                             //pila
  cuenta--;
}

int pila::cima()             //devuelve elemento superior de la
                             //pila
  return items[cuenta-1];
}

int pila::vacia()            //¿está la pila vacía?
{
  return cuenta == 0;
}
```


Un programa que invierte un flujo de enteros:

```
//utilizar pila para invertir un flujo de enteros
//archivo inver.cpp

#include <iostream.h>
#include "pila.h"

main()
{
  pila s;                          //crear una pila
  int i;

  while (cin >> i)
    s meter(i);                    //meter entrada en la pila
  for (; !s vacia(); s.sacar())    //visualizar pila
    cout << s.cima() << '\n";
  return 0;
}
```

El ejercicio anterior se puede implementar mediante el tipo `class`. La compilación separada se consigue con el archivo de cabecera *pila.h*, el archivo de implementación de funciones *pila.cpp* y el programa principal que hace uso del archivo *pila.h*

```
//ARCHIVO: pila h
//Interfaz de una pila de enteros

#ifndef PILA
#define PILA

class pila {
  private:
    const unsigned int maximo;
    int cuenta;
    int * items;
  public:
    pila();
    ~pila();
    void meter(int item);
    void sacar();
    int cima();
    int vacía();
};

#endif

//ARCHIVO: pila cpp

#include "pila.h"

pila::pila()
    maximo(100)
    cuenta(0) {
    items = new int [maximo];
}
```

```
pila::~pila() {
  delete items;
}

void pila::meter(int item) {
  items[cuenta++] = item;
}

int pila::sacar() {
  cuenta--;
}

int pila::cima() {
  return items[cuenta-1];
}

int pila::vacia() {
  return cuenta == 0;
}
```

El operador de resolución de ámbito (::) permite asociar una función miembro con su clase, y ha de definirse así cuando se implementa el cuerpo de la función

```
//ARCHIVO: principa.cpp

#include <iostream.h>
#include "pila.h"

int main() {

  pila s;
  int i;

  for (; cin.good(); cin >> ii)
    s.meter(i);

  cout << 'NUMEROS INVERTIDOS" << endl;
  for (; s.vacia(); s.sacar())
    cout << s.cima() << endl;

  return 0;
}
```

RESUMEN

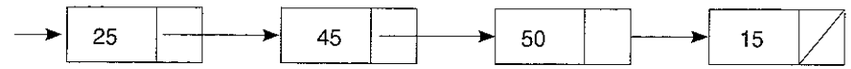
Este capítulo examina el concepto fundamental de la orientación a objetos, el tipo abstracto de datos. Los tipos abstractos de datos (TAD) describen un conjunto de objetos con la misma representación y comportamiento. Los tipos abstractos de datos representan una separación clara entre la interfaz externa de un tipo de datos y su implementación interna. La implementación de un tipo abstracto de datos está oculta. Por consiguiente, se pueden utilizar implementaciones alternativas para el mismo tipo abstracto de datos sin cambiar su interfaz.

En la mayoría de los lenguajes de programación orientados a objetos, los tipos abstractos de datos se implementan mediante **clases** (*unidades* en Pascal, *módulos* en Modula-2, *paquetes* en Ada).

En este capítulo se analizan y describen las implementaciones de tipo abstractos de datos en los lenguajes Turbo Pascal (versiones 5.5 a 7), Modula-2, Ada, C y C++.

EJERCICIOS

2.1. Construir un tipo abstracto lista enlazada de nodos que contienen enteros



2.2. Diseñar un tipo abstracto de datos pila de números enteros y que al menos soporte las siguientes operaciones:

Borrar:	Eliminar todos los números de la pila.
Copiar:	Hace una copia de la pila actual.
Meter:	Añadir un nuevo elemento en la cima de la pila.
Sacar:	Quitar un elemento de la pila.
Longitud:	Devuelve un número natural igual al número de objetos de la pila
Llena:	Devuelve *verdadero* si la pila está llena (no existe espacio libre en la pila).
Vacía:	Devuelve *verdadero* si la pila está vacía y *falso* en caso contrario
Igual:	Devuelve verdadero si existen dos pilas que tienen la misma profundidad y las dos secuencias de números son iguales cuando se comparan elemento a elemento desde sus respectivas cimas de la pila; *falso* en caso contrario

2.3. Crear un tipo abstracto `Cola` que sirva para implementar una estructura de de datos cola.

2.4. Crear un TAD para representar:

- Un vector (representación gráfica y operaciones)
- Una matriz y sus diferentes operaciones.
- Un número complejo y sus diferentes operaciones.

2.5. Crear un TAD que represente un dato tipo cadena (*string*) y sus diversas operaciones: cálculo, longitud, buscar posición de un carácter dado, concatenar cadenas, extraer una subcadena, etc.

CAPITULO 3

CONCEPTOS FUNDAMENTALES DE PROGRAMACION ORIENTADA A OBJETOS

CONTENIDO

La programación orientada a objetos es un importante conjunto de técnicas que pueden utilizarse para hacer el desarrollo de programas más eficientes, a la par que mejora la fiabilidad de los programas de computadora. En la programación orientada a objetos, los objetos son los elementos principales de construcción. Sin embargo, la simple comprensión de lo que es un objeto, o bien el uso de objetos en un programa, no significa que esté programado en un modo orientado a objetos. Lo que cuenta es el sistema en el cual los objetos se interconectan y comunican entre sí.

En este texto nos limitaremos al campo de la programación, pero es también posible hablar de sistemas de administración de bases de datos orientadas a objetos, sistemas operativos orientados a objetos, interfaces de usuarios orientados a objetos, etc.

3.1. PROGRAMACION ESTRUCTURADA

La programación estructurada se emplea desde el principio de la década de los setenta y es uno de los métodos más utilizados en el campo de la programación

La *técnica descendente* o *el refinamiento sucesivo* comienza descomponiendo el programa en piezas manejables más pequeñas, conocidas como *funciones* (subrutinas, subprogramas o procedimientos), que realizan tareas menos complejas. Un programa estructurado se construye rompiendo el programa en funciones. Esta división permite escribir código más claro y mantener el control sobre cada función

Un concepto importante se introdujo con la programación estructurada, ya comentado anteriormente: *la abstracción*, que se puede definir como la capacidad para examinar algo sin preocuparse de sus datos internos. En un programa estructurado es suficiente conocer que un procedimiento dado realiza una tarea específica. El cómo se realiza esta tarea no es importante, sino conocer cómo se utiliza correctamente la función y lo que hace.

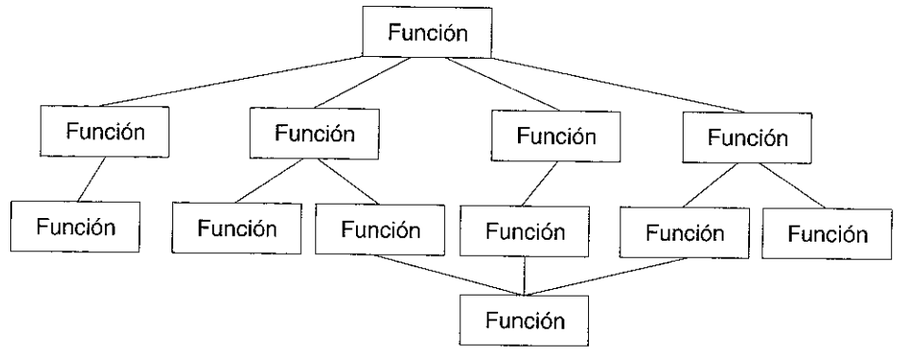


Figura 3.1. Programa estructurado.

A medida que la complejidad de un programa crece, también crece su independencia de los tipos de datos fundamentales que procesa. En un programa estructurado, las estructuras de datos de un programa son tan importantes como las operaciones realizadas sobre ellas. Esto se hace más evidente a medida que crece un programa en tamaño. Los tipos de datos se procesan en muchas funciones dentro de un programa estructurado, y cuando se producen cambios en esos tipos de datos, las modificaciones se deben hacer en cada posición que actúa sobre esos tipos de datos dentro del programa. Esta tarea puede ser frustrante y consumir un tiempo considerable en programas con millones de líneas de código y centenares de funciones.

En un programa estructurado, los datos locales se ocultan dentro de funciones y los datos compartidos se pasan como argumentos (Fig. 3.2).

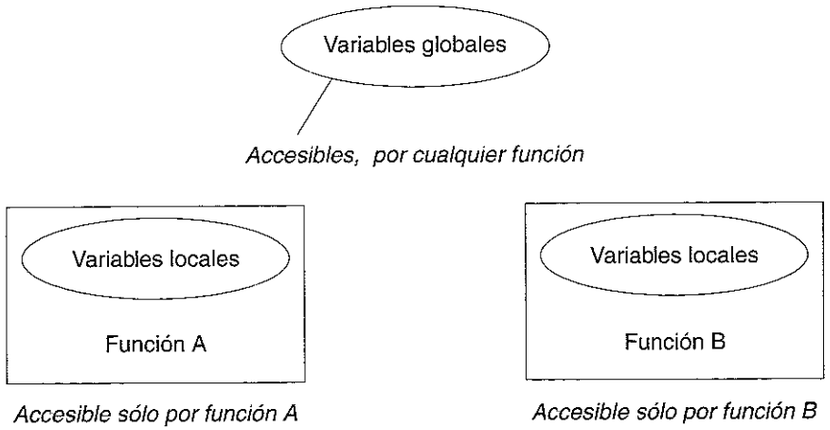


Figura 3.2. Variables globales y locales.

Otro problema es que, dado que muchas funciones acceden a los mismos datos, el medio en que se almacenan los datos se hace más crítico. La disposición de los datos no se pueden cambiar sin modificar todas las funciones que acceden a ellos. Si por ejemplo se añaden nuevos datos, se necesitará modificar todas las funciones que acceden a los datos, de modo que ellos puedan también acceder a esos elementos.

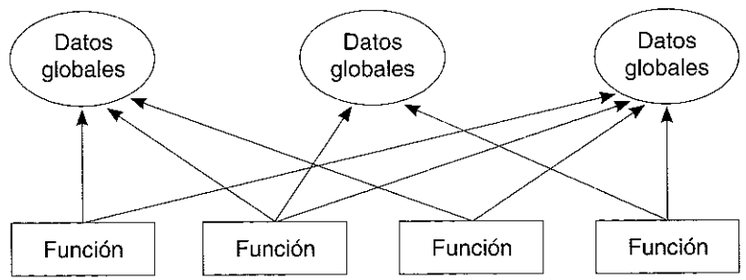


Figura 3.3. Descomposición de un programa en módulos (funciones).

Los programas basados en funciones son difíciles de diseñar. El problema es que sus componentes principales —funciones y estructuras de datos— no modelan bien el mundo real. Por ejemplo, supongamos que se está escribiendo un programa para crear los elementos de un interfaz gráfico de usuario: menús, ventanas, cuadros de diálogo, etc. ¿Qué funciones se necesitarán? ¿Qué estructuras de datos? La solución sería más aproximada si hubiera una correspondencia lo más estrecha posible entre los menús y ventanas y sus correspondientes elementos de programa.

3.1.1. Desventajas de la programación estructurada

Además de los inconvenientes citados anteriormente, comentaremos algunos otros que influyen considerablemente en el diseño.

Cuando diferentes programadores trabajan en equipo para diseñar una aplicación, a cada programador se le asigna la construcción de un conjunto específico de funciones y tipos de datos. Dado que los diferentes programadores manipulan funciones independientes que relacionan a tipos de datos compartidos mutuamente, los cambios que un programador hace a los datos se deben reflejar en el trabajo del resto del equipo. Aunque los programas estructurados son más fáciles de diseñar en grupo, los errores de comunicación entre miembros de equipos pueden conducir a gastar tiempo en rescritura.

Por otra parte, los lenguajes tradicionales presentan una dificultad añadida: *la creación de nuevos tipos de datos*. Los lenguajes de programación típicamente tienen tipos de datos incorporados: enteros, coma flotante, caracteres, etc. ¿Cómo inventar sus propios tipos de datos? Los tipos definidos por el usuario y la facilidad para crear tipos abstractos de datos en determinados lenguajes es la propiedad conocida como *extensibilidad*. Los lenguajes tradicionales no son normalmente extensibles, y eso hace que *los programas tradicionales sean* más complejos de escribir y mantener.

En resumen, con los métodos tradicionales, un programa se divide en dos componentes: *procedimientos y datos*. Cada procedimiento actúa como una caja negra. Esto es, es un componente que realiza una tarea específica, tal como convertir un conjunto de números o visualizar una ventana. Este procedimiento permite empaquetar código programa en funciones, pero ¿qué sucede con los datos? Las estructuras de datos utilizadas en programas son con frecuencia globales o se pasan explícitamente con parámetros.

3.2. ¿QUE ES LA PROGRAMACION ORIENTADA A OBJETOS?

Grady Booch, autor del método de diseño orientado a objetos, define la *programación orientada a objetos* (**POO**) como

> «un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representan una instancia de alguna clase, y cuyas clases son todas miembros de una jerarquía de clases unidas mediante relaciones de herencia» [1].

Existen tres importantes partes en la definición: la programación orientada a objetos 1) utiliza *objetos*, no algorítmicos, como bloques de construcción lógicos (*jerarquía de objetos*); 2) cada objeto es una instancia de una *clase*, y 3) las clases se relacionan unas con otras por medio de relaciones de herencia.

[1] BOOCH, Grady: *Análisis y diseño orientado a objetos con aplicaciones*. 2ª edición. Addison-Wesley/Díaz de Santos, 1995.

Un programa puede parecer orientado a objetos, pero si cualquiera de estos elementos no existe, no es un programa orientado a objetos. Específicamente, la programación sin herencia es distinta de la programación orientada a objetos; se denomina *programación con tipos abstractos de datos o programación basada en objetos*.

El concepto de objeto, al igual que los tipos abstractos de datos o tipos definidos por el usuario, es una colección de elementos de datos, junto con las funciones asociadas utilizadas para operar sobre esos datos. Sin embargo, la potencia real de los objetos reside en el modo en que los objetos pueden definir otros objetos. Este proceso, ya comentado en el Capítulo 1: se denomina *herencia* y es el mecanismo que ayuda a construir programas que se modifican fácilmente y se adaptan a aplicaciones diferentes.

Los conceptos fundamentales de programación son: *objetos, clases, herencia, mensajes* y *polimorfismo*.

> Los programas orientados a objetos constan de objetos. Los objetos de un programa se comunican con cada uno de los restantes pasando mensajes.

3.2.1. El objeto

La idea fundamental en los lenguajes orientados a objetos es combinar en una sola unidad *datos y funciones que operan sobre esos datos*. Tal unidad se denomina *objeto*. Por consiguiente, dentro de los objetos residen los datos de los lenguajes de programación tradicionales, tales como números, *arrays*, cadenas y registros, así como funciones o subrutinas que operan sobre ellos.

Las funciones dentro del objeto (*funciones miembro* en C++, *métodos* en Object-Pascal y Smalltalk) son el único medio de acceder a los datos privados de un objeto. Si se desea leer un elemento datos de un objeto se llama a la función miembro del objeto. Se lee el elemento y se devuelve el valor. No se puede acceder a los datos directamente. Los datos están ocultos, y eso asegura que no se pueden modificar accidentalmente por funciones externas al objeto.

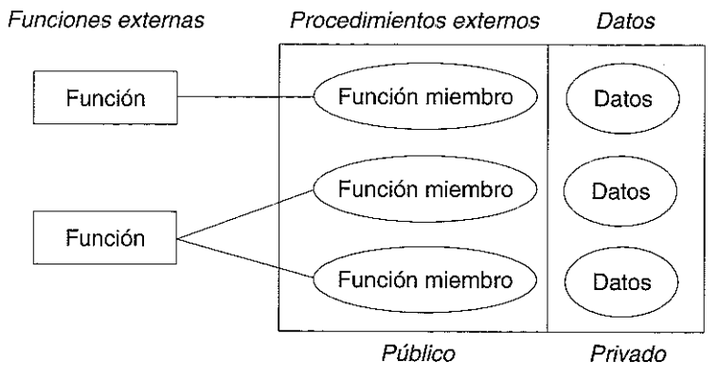


Figura 3.4. El modelo objeto.

Los datos y las funciones (procedimientos en Object-Pascal) asociados se dicen que están *encapsulados* en una única entidad o módulo. La *encapsulación de* datos y ocultación de datos son términos importantes en la descripción de lenguajes orientados a objetos.

Si se desea modificar los datos de un objeto, se conoce exactamente cuáles son las funciones que interactúan con el mismo. Ninguna otra función puede acceder a los datos. Esta característica simplifica la escritura, depuración y mantenimiento del programa.

3.2.2. Ejemplos de objetos

¿Qué clase de cosas pueden ser objetos en un programa orientado a objetos? La respuesta está sólo limitada a su imaginación. Algunos ejemplos típicos pueden ser:

- *Objetos físicos*
  Aviones en un sistema de control de tráfico aéreo.
  Automóviles en un sistema de control de tráfico terrestre.
  Casas
- *Elementos de interfaces gráficos de usuario*
  Ventanas
  Menús
  Objetos gráficos (cuadrados, triángulos, etc.).
  Teclado.
  Cuadros de diálogo.
  Ratón
- *Animales*
  Animales vertebrados.
  Animales invertebrados.
  Pescados
- *Tipos de datos definidos por el usuario*
  Datos complejos.
  Puntos de un sistema de coordenadas
- *Alimentos*
  Carnes
  Frutas.
  Pescados.
  Verduras.
  Pasteles.

Un objeto es una entidad que contiene los atributos que describen el estado de un objeto del mundo real y las acciones que se asocian con el objeto del mundo real. Se designa por un nombre o identificador del objeto.

Dentro del contexto de un lenguaje orientado a objetos (LOO), un objeto encapsula datos y los procedimientos/funciones (*métodos*) que manejan esos datos. La notación gráfica de un objeto varía de unas metodologías a otras.

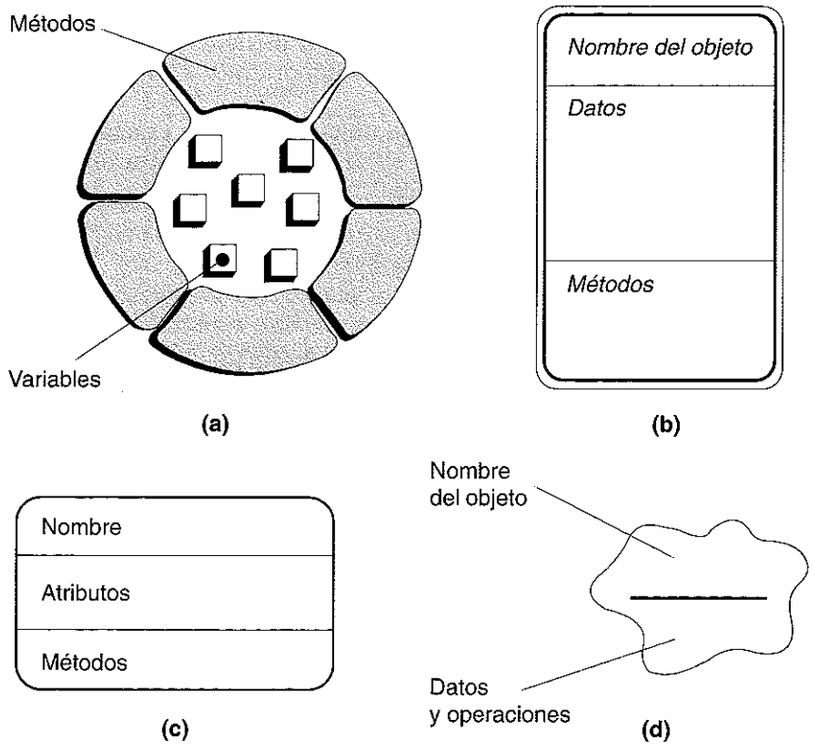


Figura 3.5. Notaciones gráficas de objetos: **(a)** Taylor; **(b)** Yourdon/Coad; **(c)** OMT; **(d)** Booch.

Consideremos una ilustración de un coche vendido por un distribuidor de coches. El identificador del objeto es `Coche1`. Los atributos asociados pueden ser: `número de matrícula`, `fabricante`, `precio_compra`, `precio_actual`, `fecha_compra`. El objeto `Coche1` se muestra en la Figura 3.6.

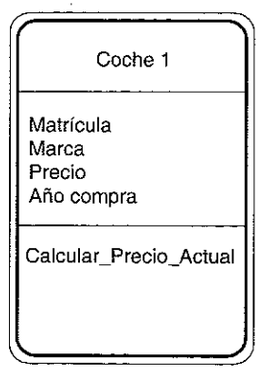


Figura 3.6. El objeto `Coche1`.

Atributos: Datos o variables que caracterizan el estado de un objeto.
Métodos: Procedimientos o acciones que cambian el estado de un objeto.

El objeto retiene cierta información y conoce cómo realizar ciertas operaciones. La encapsulación de operaciones e información es muy importante. Los métodos de un objeto sólo pueden manipular directamente datos asociados con ese objeto. Dicha encapsulación es la propiedad que permite incluir en una sola entidad (el módulo u objeto) la *información* (los datos o atributos) y las *operaciones* (los métodos o funciones) que operan sobre esa información.

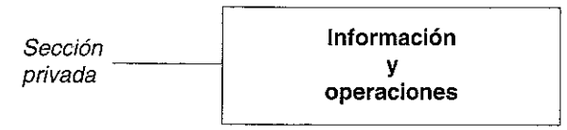


Figura 3.7. Encapsulamiento de datos.

Los objetos tienen un interfaz público y una representación privada que permiten ocultar la información que se desee al exterior.

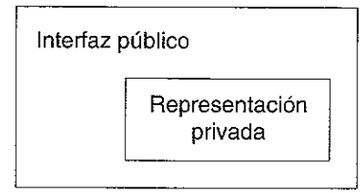


Figura 3.8. Interfaz público de un objeto.

3.2.3. Métodos y mensajes

Un programa orientado a objetos consiste en un número de objetos que se comunican unos con otros llamando a funciones miembro. Las funciones miembro (en C++) se denominan *métodos* en otros lenguajes orientados a objetos (tales como Smalltalk y Turbo Pascal 5.5/6.0/7.0).

Los procedimientos y funciones, denominados *métodos o funciones miembro*, residen en el objeto y determinan cómo actúan los objetos cuando reciben un mensaje. Un *mensaje* es la acción que hace un objeto. Un método es el procedimiento o función que se invoca para actuar sobre un objeto. Un método especifica *cómo* se ejecuta un mensaje.

El conjunto de mensajes a los cuales puede responder un objeto se denomina *protocolo* del objeto. Por ejemplo, el protocolo de un icono puede constar de

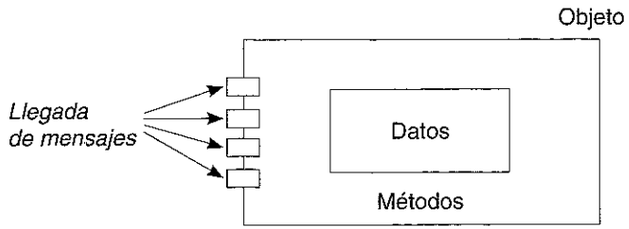


Figura 3.9. Métodos y mensajes de un objeto.

mensajes invocados por el clic de un botón del ratón cuando el usuario localiza un puntero sobre un icono.

Al igual que en las cajas negras, la estructura interna de un objeto está oculta a los usuarios y programadores. Los mensajes que recibe el objeto son los únicos conductos que conectan el objeto con el mundo externo. Los datos de un objeto están disponibles para ser manipulados sólo por los métodos del propio objeto.

Cuando se ejecuta un programa orientado a objetos ocurren tres sucesos. Primero, los objetos se crean a medida que se necesitan. Segundo, los mensajes se mueven de un objeto a otro (o desde el usuario a un objeto) a medida que el programa procesa información internamente o responde a la entrada del usuario. Tercero, cuando los objetos ya no son necesarios, se borran y se libera la memoria.

La Figura 3.10 representa un diagrama orientado a objetos.

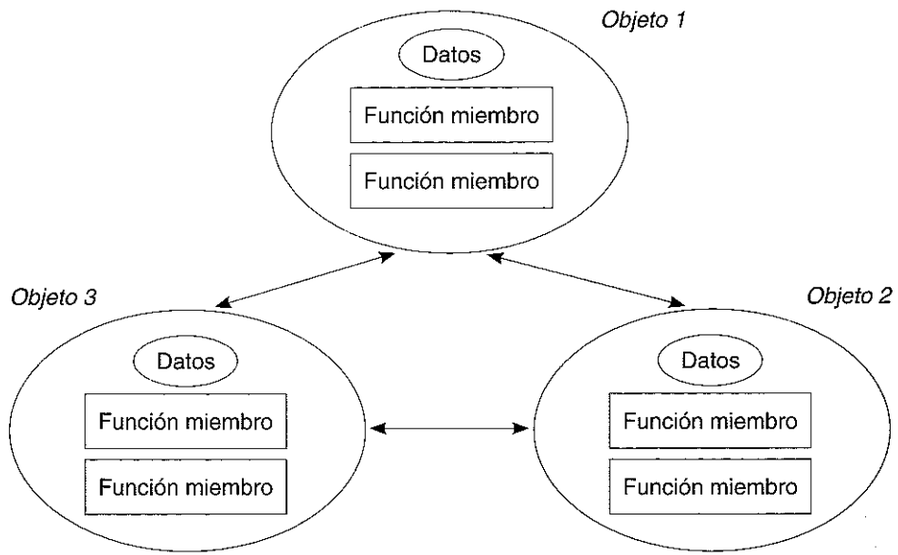


Figura 3.10. Diagrama orientado a objetos.

3.3. CLASES

Una *clase* es la descripción de un conjunto de objetos; consta de métodos y datos que resumen características comunes de un conjunto de objetos. Se pueden definir muchos objetos de la misma clase. Dicho de otro modo, una clase es la declaración de un tipo objeto.

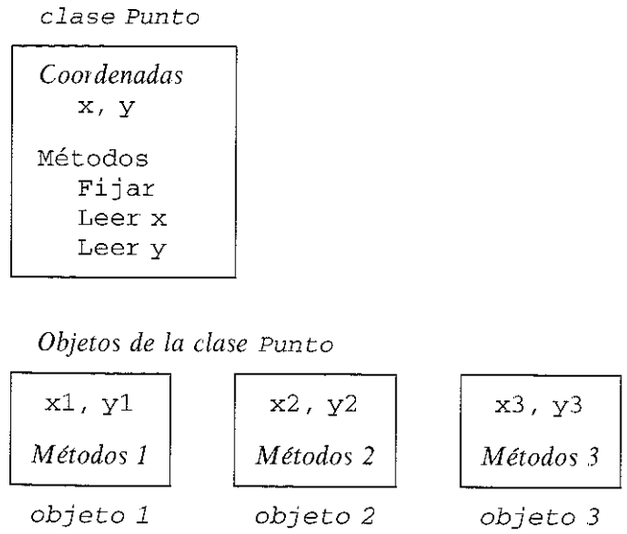
Las clases son similares a los tipos de datos y equivalen a modelos o plantillas que describen cómo se construyen ciertos tipos de objetos. Cada vez que se construye un objeto a partir de una clase, estamos creando lo que se llama una *instancia* de esa clase. Por consiguiente, los objetos no son más que instancias de una clase. *Una instancia es una variable de tipo objeto.* En general, instancia de una clase y objeto son términos intercambiables.

> Un objeto es una instancia de una clase.

Cada vez que se construye un objeto de una clase, se crea una instancia de esa clase. Los objetos se crean cuando un mensaje de petición de creación se recibe por la clase base.

3.3.1. Implementación de clases en lenguajes

Supongamos una clase `Punto` que consta de los campos dato (coordenadas x e y) y los campos función (métodos leer dichas coordenadas x e y).



- *Turbo Pascal 5.5/6.0/7.0 llama a la clase objeto* y al objeto una *instancia* de un objeto.
- Un objeto es una variable de una clase dada, y se denomina *instancia* de esa clase.
- Las funciones del objeto se denominan *métodos* (Turbo Pascal) y *funciones miembro* (en C++).

En realidad, una *clase* es un tipo de dato definido por el usuario que determina las estructuras de datos y operaciones asociadas con ese tipo. Dicho de otro modo, *una clase es una colección de objetos similares*. La definición de una clase no crea ningún objeto, de igual modo que la declaración de variables tampoco crea variables.

```
int x;
int pesetas;
```

Tenga cuidado no confundir clases con objetos de esas clases: un automóvil rojo y un automóvil azul no son objetos de clases diferentes, sino objetos de la misma clase con un atributo diferente.

3.3.2. Sintaxis

En C++ se puede declarar una clase de la siguiente forma:

```
class Punto
{
   int x;
   int y;
public:
   void fijarXY (int a, int b)
   {
      x = a;
      y = b;
   }
   int leerX ()     (return x;)
   int leerY ()     (return y;)
};
```

La sintaxis anterior ha definido la clase `Punto`, pero no ha creado ningún objeto. Para crear un objeto de tipo `Punto` tendrá que utilizarse una declaración del tipo correspondiente, al igual que se declara cualquier variable de un tipo incorporado C++.

```
Punto P;    //  se define una variable de tipo Punto
```


En Turbo Pascal se define una clase (en terminología de Borland se denomina objeto) con la palabra **object**

```
type
   <nombre-clase> =  object
      <lista de campos de datos>
      <lista de cadenas de funciones y procedimientos>
                  end;
```

y el ejemplo de un objeto `Punto`

```
type Punto =  object
      x,y : Integer;
      procedure operar;
            end;

var p : Punto;
```

- Una clase es una colección de objetos similares.
- Madonna, Michael Jackson, Prince, Mecano y Dire Straits son objetos de una clase, «cantantes de rock»; sin embargo, personas específicas con nombres específicos son miembros de esa clase si poseen ciertas características.

3.4. UN MUNDO DE OBJETOS

Una de las ventajas ineludibles de la orientación a objetos es la posibilidad de reflejar sucesos del mundo real mediante tipos abstractos de datos extensibles a objetos. Así pues, supongamos el fenómeno corriente de la conducción de una bicicleta, un automóvil, una motocicleta o un avión: usted conoce que esos vehículos comparten muchas características, mientras que difieren en otros. Por ejemplo, cualquier vehículo puede ser conducido: aunque los mecanismos de conducción difieren de unos a otros, se puede generalizar el fenómeno de la conducción. En esta consideración, enfrentados con un nuevo tipo de vehículo (por ejemplo una nave espacial), se puede suponer que existe algún medio para conducirla. Se puede decir que *vehículo* es un tipo base y *nave espacial* es un tipo derivado de ella.

En consecuencia, se puede crear un tipo base que representa el comportamiento y características comunes a los tipos derivados de este tipo base.

Un objeto es en realidad una clase especial de variable de un nuevo tipo que algún programador ha creado. Los tipos objeto definidos por el usuario se comportan como tipos incorporados que tienen datos internos y operaciones externas. Por ejemplo, un número en coma flotante tiene un exponente, mantisa y bit de signo y conoce cómo sumarse a sí mismo con otro número de coma flotante.

Los tipos objeto definidos por el usuario contienen datos definidos por el usuario (*características*) y operaciones (*comportamiento*). Las operaciones

definidas por el usuario se denominan *métodos*. Para llamar a uno de estos métodos se hace una petición al objeto: esta acción se conoce como «enviar un mensaje al objeto» Por ejemplo, para detener un objeto automóvil se envía un mensaje de *parada* («stop»). Obsérvese que esta operación se basa en la noción de encapsulación (encapsulamiento): se indica al objeto lo que ha de hacer, pero los detalles de cómo funciona se han encapsulado (ocultado).

3.4.1. Definición de objetos[2]

Un *objeto* (desde el punto de vista formal se debería hablar de **clase**), como ya se ha comentado, es una abstracción de cosas (entidades) del mundo real, tales que:

- Todas las cosas del mundo real dentro de un conjunto —denominadas *instancias*— tienen las mismas características.
- Todas las instancias siguen las mismas reglas

Cada objeto consta de:

- Estado (*atributos*)
- Operaciones o comportamiento (métodos invocados por mensajes).

Desde el punto de vista informático, los objetos son *tipos abstractos de datos* (tipos que encapsulan datos y funciones que operan sobre esos datos).

Algunos ejemplos típicos de objetos:

- *Número racional*
  Estado (valor actual)
  Operaciones (sumar, multiplicar, asignar...)
- *Vehículo*
  Estado (velocidad, posición, precio...).
  Operaciones (acelerar, frenar, parar...).
- *Conjunto*
  Estado (elementos).
  Operaciones (añadir, quitar, visualizar...).
- *Avión*
  Estado (fabricante, modelo, matrícula, número de pasajeros...).
  Operaciones (aterrizar, despegar, navegar...)

3.4.2. Identificación de objetos

El primer problema que se nos plantea al analizar un problema que se desea implementar mediante un programa orientado a objetos es *identificar los objetos*; es decir, ¿qué cosas son objetos?; ¿cómo deducimos los objetos dentro del dominio de la definición del problema?

La identificación de objetos se obtiene examinando la descripción del problema (análisis gramatical somero del enunciado o descripción) y localizando los nombres o cláusulas nominales. Normalmente, estos nombres y sus sinónimos se suelen escribir en una tabla de la que luego deduciremos los objetos reales.

Los objetos, según Shlaer, Mellor y Coad/Yourdon, pueden caer dentro de las siguientes categorías:

- *Cosas tangibles* (avión, reactor nuclear, fuente de alimentación, televisor, libro, automóvil).
- *Roles o papeles* jugados o representados por personas (gerente, cliente, empleado, médico, paciente, ingeniero).
- *Organizaciones* (empresa, división, equipo...)
- *Incidentes* (representa un suceso —evento— u ocurrencia, tales como vuelo, accidente, suceso, llamada a un servicio de asistencia técnica...).
- *Interacciones* (implican generalmente una transacción o contrato y relacionan dos o más objetos del modelo: compras —comprador, vendedor, artículo—, matrimonio —esposo, esposa, fecha de boda).
- *Especificaciones* (muestran aplicaciones de inventario o fabricación: refrigerador, nevera...).
- *Lugares* (sala de embarque, muelle de carga...).

Una vez identificados los objetos, será preciso identificar los atributos y las operaciones que actúan sobre ellos.

Los *atributos* describen la abstracción de características individuales que poseen todos los objetos.

AVION
Matrícula
Licencia del piloto
Nombre de avión
Capacidad de carga
Número de pasajeros

EMPLEADO
Nombre
Número de identificación
Salario
Dirección
Nombre del departamento

Las *operaciones* cambian el objeto —su comportamiento— de alguna forma, es decir, cambian valores de uno o más atributos contenidos en el objeto. Aunque existen gran número de operaciones que se pueden realizar sobre un objeto, generalmente se dividen en tres grandes grupos[3]:

- Operaciones que *manipulan* los datos de alguna forma específica (añadir, borrar, cambiar formato...).
- Operaciones que realizan un *cálculo o proceso*

[2] Cuando se habla de modo genérico, en realidad se debería hablar de CLASES, dado que la clase en el tipo de dato y objeto es sólo una instancia, ejemplar o caso de la clase. Aquí mantenemos el término objeto por conservar la rigurosidad de la definición «orientado a objetos», aunque en realidad la definición desde el punto de vista técnico sería la clase

[3] PRESSMAN, Roger: *Ingeniería del software. Un enfoque práctico*. 3.ª edición. McGraw-Hill, 1993

- Operaciones que comprueban (*monitorizan*) un objeto frente a la ocurrencia de algún suceso de control.

La identificación de las operaciones se realiza haciendo un nuevo análisis gramatical de la descripción del problema y buscando y aislando los verbos del texto.

3.4.3. Duración de los objetos

Los objetos son entidades que existen en el tiempo; por ello deben ser creados o instanciados (normalmente a través de otros objetos). Esta operación se hace a través de operaciones especiales llamadas *constructores* en C++ o *inicializadores*. Estas operaciones se ejecutarán implícitamente por el compilador o explícitamente por el *programador*, mediante invocación a los citados constructores.

3.4.4. Objetos frente a clases. Representación gráfica (Notación de Ege)

Los objetos y las clases se comparan a *variables* y *tipos* en lenguajes de programación convencional. Una variable es una instancia de un tipo, al igual que un objeto es una instancia de una clase; sin embargo, una clase es más expresiva que un tipo. Expresa la estructura y todos los procedimientos y funciones que se pueden aplicar a una de sus instancias.

En un lenguaje estructurado, un tipo `integer`, por ejemplo, define la estructura de una variable entera, por ejemplo una secuencia de 16 bits y los procedimientos y funciones que se pueden realizar sobre enteros. De acuerdo a nuestra definición de «clase», el tipo `integer` será una clase. Sin embargo, en estos lenguajes de programación no es posible agrupar nuevos tipos y sus correspondientes nuevas funciones y procedimientos en una única unidad. En un lenguaje orientado a objetos una clase proporciona este servicio.

Además de los términos objetos y clases, existen otros términos en orientación a objetos. Las variables o campos que se declaran dentro de una clase se denominan *datos miembro* en C++; otros lenguajes se refieren a ellos como *variables instancia*. Las funciones que se declaran dentro de una clase se denominan *funciones miembro* en C++; otros lenguajes utilizan el término *método*. Las funciones y campos miembro se conocen como *características miembro*, o simplemente *miembros*. A veces se invierten las palabras, y las funciones miembros se conocen como *miembro función* y los campos se denominan *miembro datos*.

Es útil ilustrar objetos y clases con diagramas[4]. La Figura 3.11 muestra el esquema general de un diagrama objeto. Un objeto se dibuja como una caja.

[4] Las notaciones de clases y objetos utilizada en esta sección se deben a Raimund K. Ege, que las dio a conocer en su libro *Programming in an Object-Oriented Environment*. Academic Press (AP), 1992.

La caja se etiqueta con el nombre del objeto y representa el límite o frontera entre el interior y el exterior de un objeto.

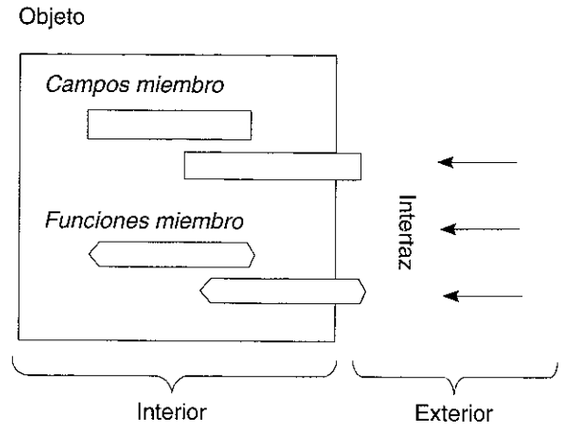


Figura 3.11. Diagrama de un objeto.

Un campo se dibuja por una caja rectangular, una función por un hexágono largo. Los campos y funciones se etiquetan con sus nombres. Si una caja rectangular contiene algo, entonces se representa el valor del campo para el objeto dibujado. Los campos y funciones miembro en el interior de la caja están ocultos al exterior, que significa estar *encapsulados*. El acceso a las características de los miembros (campos y funciones) es posible a través del interfaz del objeto. En una clase en C++, el interfaz se construye a partir de todas las características que se listan después de la palabra reserva **`public`**; puede ser funciones y campos.

La Figura 3.12 muestra el diagrama objeto del objeto `"hola mundo"`. Se llama `Saludo1` y permite acceder a su estado interno a través de las funciones miembro públicas `cambiar` y `anunciar`. El campo miembro privado contiene el valor `Esto es saludo1`.

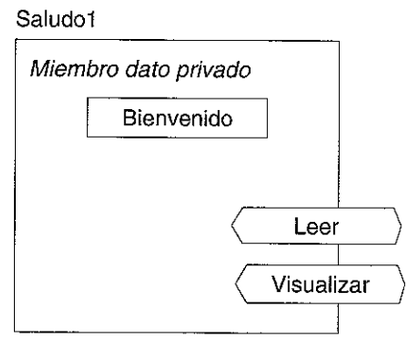


Figura 3.12. El objeto `Saludo1`.

¿Cuál es la diferencia entre clase y objeto?

Un objeto es un simple elemento, no importa lo complejo que pueda ser. Una clase, por el contrario, describe una familia de elementos similares. En la práctica, una clase es como un esquema o plantilla que se utiliza para definir o crear objetos

A partir de una clase se puede definir un número determinado de objetos. Cada uno de estos objetos generalmente tendrá un estado particular propio (una pluma estilográfica puede estar llena, otra puede estar medio llena y otra totalmente vacía) y otras características (como su color), aunque compartan algunas operaciones comunes (como «escribir» o «llenar su depósito de tinta»).

Los objetos tienen las siguientes características:

- Se agrupan en tipos llamados *clases*.
- Tienen *datos internos* que definen su estado actual.
- Soportan *ocultación de datos*.
- Pueden *heredar* propiedades de otros objetos.
- Pueden comunicarse con otros objetos pasando *mensajes*.
- Tienen *métodos* que definen su *comportamiento*.

La Figura 3.13 muestra el diseño general de diagramas que representan a una clase y a objetos pertenecientes a ella.

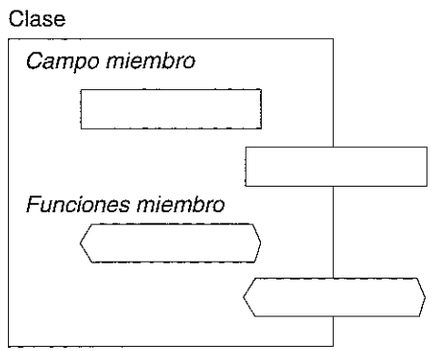


Figura 3.13. Diagrama de una clase.

Una clase es un tipo definido que determina las estructuras de datos y operaciones asociadas con ese tipo. Las clases son como plantillas que describen cómo ciertos tipos de objetos están construidos. Cada vez que se construye un objeto de una clase, estamos creando lo que se llama una *instancia* (modelo o ejemplar) de una clase y la operación correspondiente se llama *instanciación* (creación de instancias). Por consiguiente, los objetos no son más que instancias de clases. En general, los términos *objeto* e *instancia de una clase* se pueden utilizar indistintamente.

- Un objeto es una instancia de una clase.
- Una clase puede tener muchas instancias y cada una es un objeto independiente.
- Una clase es una plantilla que se utiliza para describir uno o más objetos de un mismo tipo.

3.4.5. Datos internos

Una propiedad importante de los objetos es que almacenan información de su *estado* en forma de datos internos. El estado de un objeto es simplemente el conjunto de valores de todas las variables contenidas dentro del objeto en un instante dado. A veces se denominan a las variables que representan a los objetos *variables de estado*. Así por ejemplo, si tuviésemos una clase `ventana` en C++:

```
class ventana {
   int posx, posy;
   int tipo_ventana;
   int tipo_borde;
   int color_ventana;
public:
   move_hor (int dir, int ang);
   move_ver (int dir, int ang);
};
```

Las variables de estado pueden ser las coordenadas actuales de la ventana y sus atributos de color actuales.

En muchos casos, las variables de estado se utilizan sólo indirectamente. Así, en el caso del ejemplo de la ventana, suponga que una orden (mandato) típica a la ventana es:

```
reducir en 5 filas y 6 columnas
```

Esto significa que la ventana se reducirá en tamaño una cantidad dada por 5 filas y 6 columnas:

```
mensaje reducir
```

Afortunadamente, no necesita tener que guardar la posición actual de la ventana, ya que el objeto hace esa operación por usted. La posición actual se almacena en una variable de estado que mantiene internamente la ventana. Naturalmente, se puede acceder a esta variable estado cuando se desee, enviando un mensaje tal como:

```
indicar posicion actual
```

3.4.6. Ocultación de datos

Con el fin de mantener las características de caja negra de POO, se debe considerar cómo se accede a un objeto en el diseño del mismo. Normalmente es una buena práctica restringir el acceso a las variables estado de un objeto y a otra información interna que se utiliza para definir el objeto. Cuando se utiliza un objeto no necesitamos conocer todos los detalles de la implementación. Esta práctica de limitación del acceso a cierta información interna se llama *ocultación de datos*.

En el ejemplo anterior de ventana, el usuario no necesita saber cómo se implementa la ventana; sólo cómo se utiliza. Los detalles internos de la implementación pueden y deben ser ocultados. Considerando este enfoque, somos libres de cambiar el diseño de la ventana (bien para mejorar su eficiencia o bien para obtener su trabajo en un hardware diferente), sin tener que cambiar el código que la utiliza.

> C++ soporta las características de ocultación de datos con las palabras reservadas **`public`**, **`private`** y **`protected`**.

3.5. HERENCIA

La encapsulación es una característica muy potente, y junto con la ocultación de la información, representan el concepto avanzado de objeto, que adquiere su mayor relevancia cuando encapsula e integra datos, más las operaciones que manipulan los datos en dicha entidad. Sin embargo, la orientación a objetos se caracteriza, además de por las propiedades anteriores, por incorporar la característica de *herencia*, propiedad que permite a los objetos ser construidos a partir de otros objetos. Dicho de otro modo, la capacidad de un objeto para utilizar las estructuras de datos y los métodos previstos en antepasados o ascendientes. El objetivo final es la **reutilizabilidad** o **reutilización** (*reusability*)[5], es decir reutilizar código anteriormente ya desarrollado.

La herencia se apoya en el significado de ese concepto en la vida diaria. Así, las clases básicas o fundamentales se dividen en subclases. Los animales se dividen en mamíferos, anfibios, insectos, pájaros, peces, etc. La clase vehículo se divide en subclase automóvil, motocicleta, camión, autobús, etc. El principio en que se basa la división de clases es la jerarquía compartiendo características comunes. Así, todos los vehículos citados tienen un motor y ruedas, que son características comunes; si bien los camiones tienen una caja para transportar mercancías, mientras que las motocicletas tienen un manillar en lugar de un volante.

[5] Este término también se suele traducir por *reusabilidad*, aunque no es un término aceptado por el Diccionario de la Real Academia Española.

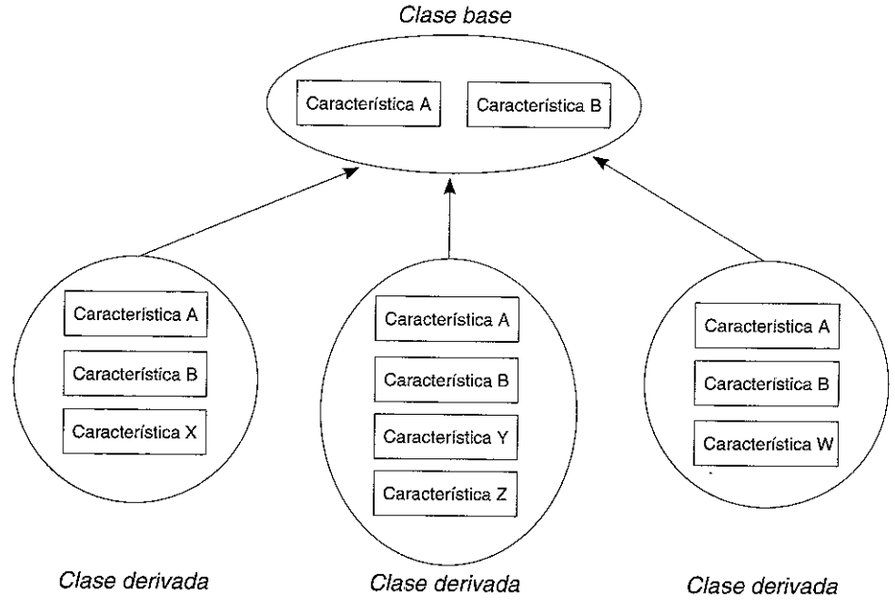


Figura 3.14. Jerarquía de clases.

La herencia supone una *clase base* y una *jerarquía de clases* que contienen las *clases derivadas* de la clase base. Las clases derivadas pueden heredar el código y los datos de su clase base, añadiendo su propio código especial y datos a ellas, incluso cambiar aquellos elementos de la clase base que necesita sean diferentes.

No se debe confundir las relaciones de los objetos con las clases, con las relaciones de una clase base con sus clases derivadas. Los objetos existentes en la memoria de la computadora expresan las características exactas de su clase y sirven como un módulo o plantilla. Las clases derivadas heredan características de su clase base, pero añaden otras características propias nuevas.

> Una clase *hereda* sus características (datos y funciones) de otra clase.

Así, se puede decir que una clase de objetos es un conjunto de objetos que comparten características y comportamientos comunes. Estas características y comportamientos se definen en una clase base. Las clases derivadas se crean en un proceso de definición de nuevos tipos y reutilización del código anteriormente desarrollado en la definición de sus clases base. Este proceso se denomina *programación por herencia*. Las clases que heredan propiedades de una clase base pueden a su vez servir como definiciones base de otras clases. Las jerarquías de clases se organizan en forma de árbol.

3.5.1. Sintaxis

En lenguaje C++ la propiedad de herencia se implementa con la siguiente sintaxis:

```
class <derivadas> : <lista clases base>
{
   <datos propios>
   <funciones miembro propias>
}
```

En lenguaje Pascal orientado a objetos (versiones Turbo 5.5 a 7.0), la sintaxis de una clase (objeto en su terminología) es:

```
type
   <nombre-clase> = object (clase ascendiente)
   <campos propios de nuevo objeto>
   <métodos propios del nuevo objeto>
end;
```

Así, por ejemplo, en C++, si se crea una clase base:

```
class base {
   int x, y;
   public:
   void hacerAlgo ();
}
```

en Turbo Pascal se escribiría la clase equivalente:

```
base = object
   x, y : integer;
   procedure hacerAlgo;
end;
```

Se desea construir un nuevo tipo derivado del tipo base existente, tal que el tipo derivado sea idéntico al tipo base, con una excepción: extender `base` añadiendo un método llamado `hacerOtraCosa`. Se construye una clase derivada. En C++:

```
class derivada : class base {
public:
   void hacerOtraCosa ();
};
```

En Turbo Pascal,

```
derivada = object (base)
   procedure hacerOtraCosa;
end;
```


Dado que `derivada` hereda todos los datos y métodos de `base`, no se necesita redefinirlos; simplemente *indicar* al compilador que desea derivar un nuevo tipo (*derivado*) de un tipo base (base) y añadir el nuevo método. La herencia permite construir tipos de datos complejos sin repetir mucho código. El nuevo tipo hereda unas características y comportamiento de un ascendiente o antepasado. Puede también reimplementar o sobrescribir cualquier método que elija. Esta reimplementación de métodos del tipo base en los tipos derivados es fundamental el concepto de polimorfismo, que se verá más tarde.

3.5.2. Tipos de herencia

Existen dos mecanismos de herencia utilizados comúnmente en programación orientada a objetos: *herencia simple* y *herencia múltiple*.

En *herencia simple*, un objeto (*clase*) puede tener sólo un ascendiente, o dicho de otro modo, una subbase puede heredar datos y métodos de una única clase, así como añadir o quitar comportamientos de la clase base. Turbo Pascal sólo admite este tipo de herencia. C++ admite herencia simple y múltiple.

La *herencia múltiple* es la propiedad de una clase de poder tener más de un ascendiente inmediato, o lo que es igual, adquirir datos y métodos de más de una clase.

Una representación gráfica de los tipos de herencia con una clase base genérica 01 se muestra en la Figura 3.16.

La Figura 3.15 representa los gráficos de herencia simple y herencia múltiple de la clase `figura` y `persona`, respectivamente.

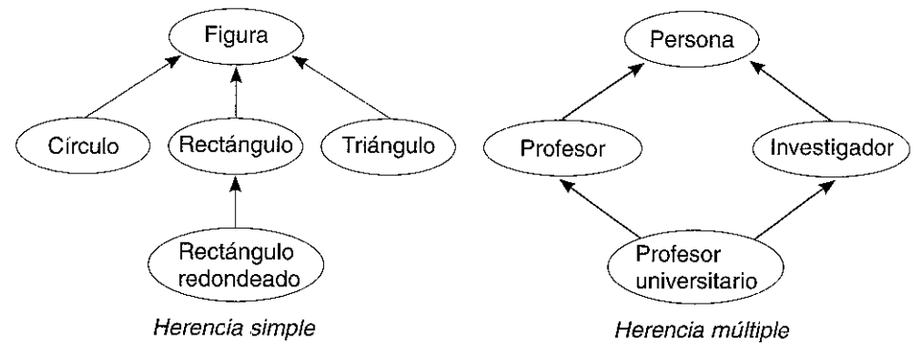


Figura 3.15. Tipos de herencia.

En la Figura 3.16 se muestran gráficamente las relaciones de herencia, apreciándose fácilmente los dos tipos (simple y múltiple).

A primera vista, se puede suponer que la herencia múltiple es mejor que la herencia simple; sin embargo, como ahora comentaremos, no siempre será así.

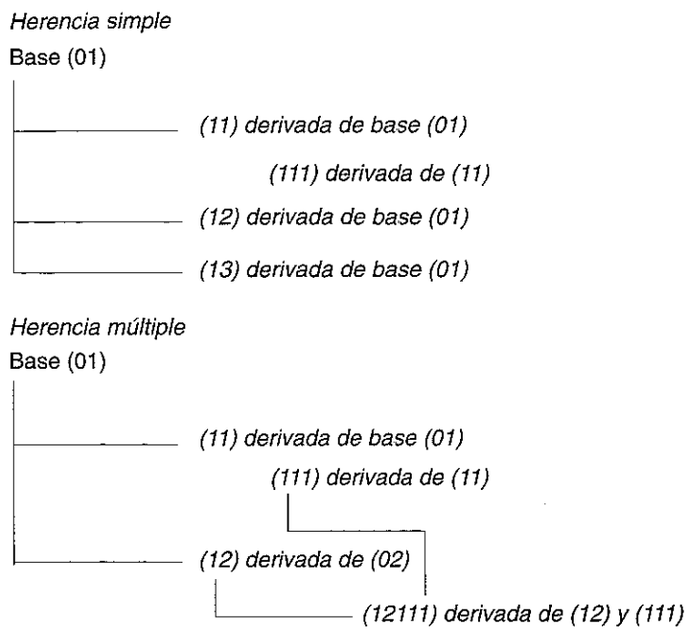


Figura 3.16. Herencia simple y múltiple.

En los lenguajes de POO —incluyendo Object-Pascal, Objective-C, Smalltalk y Acter— implementan sólo herencia simple. Eiffel y C++ (a partir de la versión 2.0), por otra parte, soportan herencia múltiple.

En general, prácticamente todo lo que se puede hacer con herencia múltiple se puede hacer con herencia simple, aunque a veces resulta más difícil. Una dificultad surge con la herencia múltiple cuando se combinan diferentes tipos de objetos, cada uno de los cuales define métodos o campos iguales. Supongamos dos tipos de objetos pertenecientes a las clases `Gráficos` y `Sonidos`, y se crea un nuevo objeto denominado `Multimedia` a partir de ellos. `Gráficos` tiene tres campos datos: `tamaño`, `color` y `mapasdebits`, y los métodos `dibujar`, `cargar`, `almacenar` y `escala`; `sonidos` tiene dos campos datos, `duración`, `voz` y `tono`, y los métodos `reproducir`, `cargar`, `escala` y `almacenar`. Así, para un objeto `Multimedia`, el método escala significa poner el sonido en diferentes tonalidades, o bien aumentar/reducir el tamaño de la escala del gráfico.

Naturalmente, el problema que se produce es la ambigüedad, y se tendrá que resolver con una operación de prioridad que el correspondiente lenguaje deberá soportar y entender en cada caso.

En realidad, ni la herencia simple ni la herencia múltiple son perfectas en todos los casos, y ambas pueden requerir un poco más de código extra que represente bien las diferencias en el modo de trabajo.

3.6. COMUNICACIONES ENTRE OBJETOS: LOS MENSAJES

Ya se ha mencionado en secciones anteriores que los objetos realizan acciones cuando ellos reciben mensajes. El mensaje es esencialmente una orden que se envía a un objeto para indicarle que realice alguna acción. Esta técnica de enviar mensajes a objetos se denomina *pasar mensajes*. Los objetos se comunican entre sí enviando mensajes, al igual que sucede con las personas. Los mensajes tienen una contrapartida denominada métodos. Mensajes y métodos son dos caras de la misma moneda. Los *métodos* son los procedimientos que se invocan cuando un objeto recibe un *mensaje*. En terminología de programación tradicional, un mensaje es una *llamada a una función*. Los mensajes juegan un papel crítico en POO. Sin ellos los objetos que se definen no se podrán comunicar entre sí. Como ejemplo, consideramos enviar un mensaje tal como *subir 5 líneas* el objeto ventana definido anteriormente. El aspecto importante no es cómo se implementa un mensaje, sino cómo se utiliza.

Consideremos de nuevo nuestro objeto `ventana`. Supongamos que deseamos cambiar su tamaño, de modo que le enviamos el mensaje

```
Reducir 3 columnas por la derecha
```

Observe que no le indicamos a la ventana cómo cambiar su tamaño, la ventana maneja la operación por sí misma. De hecho, se puede enviar el mismo mensaje a diferentes clases de ventanas y esperar a que cada una realice la misma acción.

Los mensajes pueden venir de otros objetos o desde fuentes externas, tales como un ratón o un teclado.

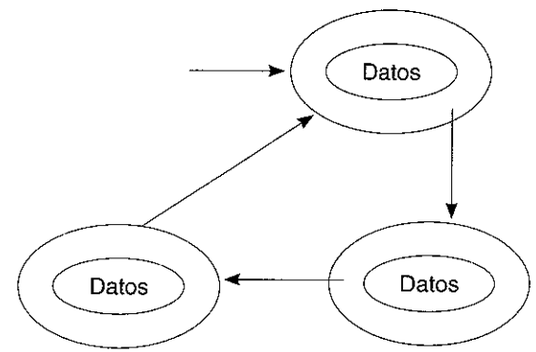


Figura 3.17. Mensajes entre objetos.

Una aplicación Windows es un buen ejemplo de cómo se emplean los mensajes para comunicarse entre objetos. El usuario pulsa un botón para enviar (remitir, despachar) mensajes a otros objetos que realizan una función específica. Si se pulsa el botón `Exit`, se envía un mensaje al objeto responsable de cerrar la aplicación. Si el mensaje es válido, se invoca el método interno. Entonces se cierra la aplicación.

3.6.1. Activación de objetos

A los objetos sólo se puede acceder a través de su interfaz público. ¿Cómo se permite el acceso a un objeto? Un objeto accede a otro objeto enviándole un mensaje.

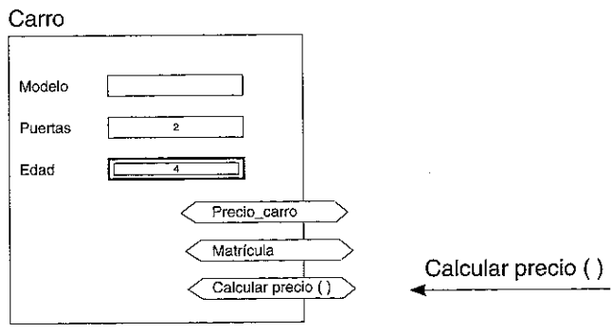


Figura 3.18. Envío de un mensaje.

3.6.2. Mensajes

Un *mensaje* es una petición de un objeto a otro objeto al que le solicita ejecutar uno de sus métodos. Por convenio, el objeto que envía la petición se denomina *emisor* y el objeto que recibe la petición se denomina *receptor*.

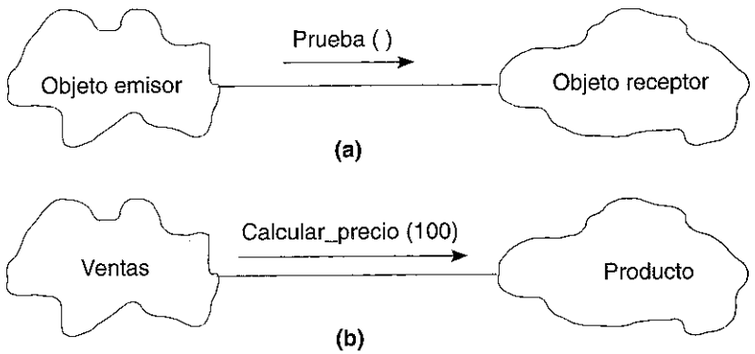


Figura 3.19. Objetos emisor y receptor de un mensaje.

Estructuralmente, un mensaje consta de tres partes:

- *Identidad* del receptor.
- El *método* que se ha de ejecutar.

- *Información especial* necesaria para realizar el método invocado (argumentos o parámetros requeridos).

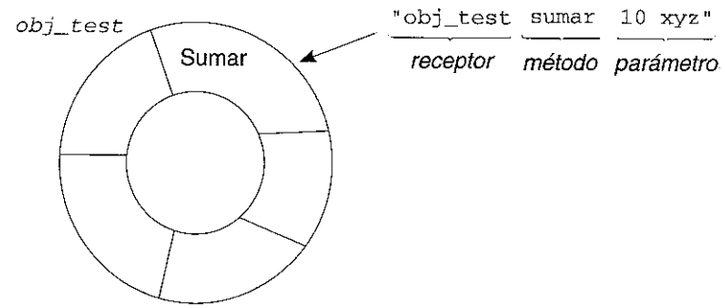


Figura 3.20. Estructura de un mensaje.

Cuando un objeto está inactivo (durmiendo) y recibe un mensaje se hace activo. El mensaje enviado por otros objetos o fuentes tiene asociado un método que se activará cuando el receptor recibe dicho mensaje. La petición no especifica *cómo* se realiza la operación. Tal información se oculta siempre al emisor.

El conjunto de mensajes a los que responde un objeto se denomina *comportamiento* del objeto. No todos los mensajes de un objeto responden; es preciso que pertenezcan al interfaz accesible.

Nombre de un mensaje

Un mensaje incluye el nombre de una operación y cualquier argumento requerido por esa operación. Con frecuencia, es útil referirse a una operación por nombre, sin considerar sus argumentos.

Métodos

Cuando un objeto recibe un mensaje, se realiza la operación solicitada ejecutando un método. Un *método* es el algoritmo ejecutado en respuesta a la recepción de un mensaje cuyo nombre se corresponde con el nombre del método.

La secuencia actual de acontecimientos es que el emisor envía su mensaje; el receptor ejecuta el método apropiado, consumiendo los parámetros; a continuación, el receptor devuelve algún tipo de respuesta al emisor para reconocer el mensaje y devolver cualquier información que se haya solicitado.

El receptor responde a un mensaje.

3.6.3. Paso de mensajes

Los objetos se comunican entre sí a través del uso de mensajes. El interfaz del mensaje se define un interfaz claro entre el objeto y el resto de su entorno.

Esencialmente, el protocolo de un mensaje implica dos partes: el emisor y el receptor. Cuando un objeto emisor envía un mensaje a un objeto receptor, tiene que especificar lo siguiente:

1. Un receptor.
2. Un nombre de mensaje
3. Argumentos o parámetros (si se necesita).

En primer lugar, un objeto receptor que ha de recibir el mensaje que se ha especificado. Los objetos no especificados por el emisor no responderán. El receptor trata de concordar el nombre del mensaje con los mensajes que él entiende. Si el mensaje no se entiende, el objeto receptor no se activará. Si el mensaje se entiende por el objeto receptor, el receptor aceptará y responderá al mensaje invocando el método asociado.

Los parámetros o argumentos pueden ser:

1. Datos utilizados por el método invocado.
2. Un mensaje, propiamente dicho

La estructura de un mensaje puede ser:

```
enviar <Objeto A>.<Método1  (parámetro1,     parámetroN)>
```

El ejemplo siguiente muestra algunos mensajes que se pueden enviar al objeto `Coche1`. El primero de éstos invoca al método `Precio_Coche` y no tiene argumentos, mientras que el segundo, `Fijar_precio`, envía los parámetros 8-10-92, y `Poner_en_blanco` no tiene argumentos.

Ejemplo

```
enviar Coche1.Precio_Coche()          envía a Coche1 el mensaje Precio_Coche
enviar Coche1.Fijar_precio(8-10-92)   envía a Coche1 el mensaje Fijar_precio
                                      con el parámetro 8- 10-92
enviar Coche1.Poner_en_blanco()       envía a Coche1 el mensaje Poner_en_blanco
```

3.7. ESTRUCTURA INTERNA DE UN OBJETO

La estructura interna de un objeto consta de dos componentes básicos:

- Atributos.
- Métodos (operaciones o servicios).

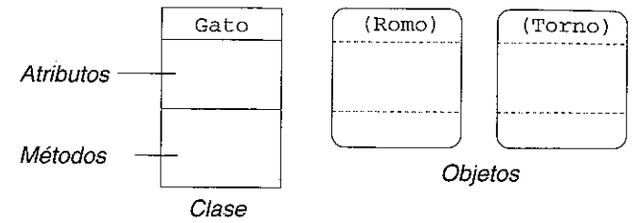


Figura 3.21. Notación gráfica OMT de una clase y de un objeto.

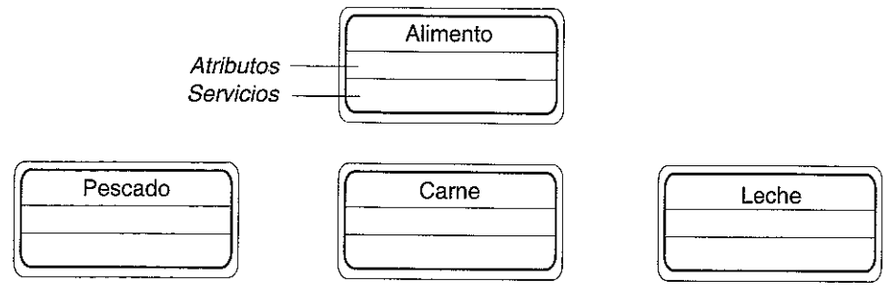


Figura 3.22. Objetos en notación Yourdon/Coad.

3.7.1. Atributos

Los **atributos** describen el estado del objeto. Un atributo consta de dos partes: un nombre de atributo y un valor de atributo.

Los objetos simples pueden constar de tipos primitivos, tales como enteros, carácter, boolean, reales, o tipos simples definidos por el usuario. Los objetos complejos pueden constar de pilas, conjuntos, listas, array, etc., o incluso estructuras recursivas de alguno o todos los elementos.

Los constructores se utilizan para construir estos objetos complejos a partir de otros objetos complejos.

3.7.2. Métodos

Los **métodos** (operaciones o servicios) describen el *comportamiento* asociado a un objeto. Representan las acciones que pueden realizarse por un objeto o sobre un objeto. La ejecución de un método puede conducir a cambiar el estado del objeto o dato local del objeto.

Cada método tiene un nombre y un cuerpo que realiza la acción o comportamiento asociado con el nombre del método. En un LOO, el cuerpo de un método consta de un bloque de código procedimental que ejecuta la acción requerida. Todos los métodos que alteran o acceden a los datos de un objeto se definen dentro del objeto. Un objeto puede modificar directamente o acceder a los datos de otros objetos.

Un método dentro de un objeto se activa por un mensaje que se envía por otro objeto al objeto que contiene el método. De modo alternativo, se puede llamar por otro método en el mismo objeto por un mensaje local enviado de un método a otro dentro del objeto

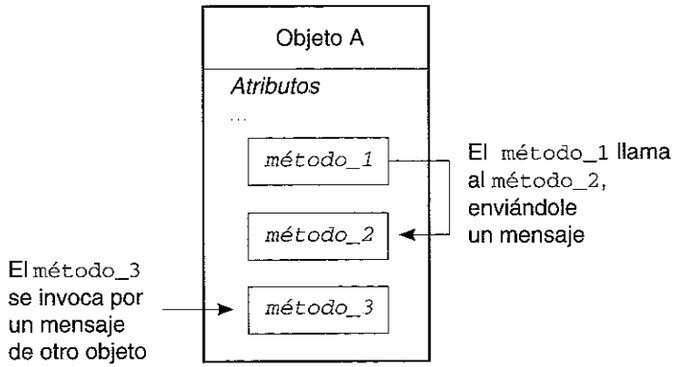


Figura 3.23. Invocación de un método.

```
metodo: Precio_coche
inicio
   Precio_coche := Precio_coste * (Marca+1);
fin.
```

3.8. CLASES

La clase es la construcción del lenguaje utilizada más frecuentemente para definir los tipos abstractos de datos en lenguajes de programación orientados a objetos. Una de las primeras veces que se utilizó el concepto de clase fue en Simula (Dahl y Nygaard, 1966; Dahl, Myhrhang y Nigaard, 1970) como entidad que declara conjuntos de objetos similares. En Simula, las clases se utilizaron principalmente como plantillas para crear objetos de la misma estructura. Los atributos de un objeto pueden ser tipos base, tales como enteros, reales y booleans; o bien pueden ser arrays, procedimientos o instancias de otras clases.

Generalmente, una clase se puede definir como una descripción abstracta de un grupo de objetos, cada uno de los cuales se diferencia por un *estado* específico y es capaz de realizar una serie de *operaciones*.

Por ejemplo, una pluma estilográfica es un objeto que tiene un estado (llena de tinta o vacía) que puede realizar algunas operaciones (por ejemplo escribir, poner/quitar el capuchón, rellenar si está vacía).

En programación, una clase es una estructura que contiene datos y procedimientos (o funciones) que son capaces de operar sobre esos datos. Una clase pluma estilográfica puede tener, por ejemplo, una variable que indica si está llena o vacía; otra variable puede contener la cantidad de tinta cargada realmente. La clase contendrá algunas funciones que operan o utilizan esas variables.

Dentro de un programa, las clases tienen dos propósitos principales: definir abstracciones y favorecer la modularidad.

¿Cuál es la diferencia entre una clase y un objeto, con independencia de su complejidad? Una clase verdaderamente describe una familia de elementos similares. En realidad, una clase es una plantilla para un tipo particular de objetos. Si se tienen muchos objetos del mismo tipo, sólo se tienen que definir las características generales de ese tipo una vez, en lugar de en cada objeto.

A partir de una clase se puede definir un número de objetos. Cada uno de estos objetos tendrá generalmente un estado peculiar propio (una pluma puede estar rellenada, otra puede estar medio-vacía y otra estar totalmente vacía) y otras características (como su color), aunque compartirán operaciones comunes (como «escribir», «llenar», «poner el capuchón», etc.).

En resumen, un objeto es una instancia de una clase.

3.8.1. Una comparación con tablas de datos

Una *clase* se puede considerar como la extensión de un registro. Aquellas personas familiarizadas con sistemas de bases de datos pueden asociar clase e instancias con tablas y registros, respectivamente. Al igual que una clase, una tabla define los nombres y los tipos de datos de la información que contenga. Del mismo modo que una instancia, un registro de esa tabla proporciona los valores específicos para una entrada particular. La principal diferencia, a nivel conceptual, es que las clases contienen métodos, además de las definiciones de datos.

Una clase es una caja negra o módulo en la que está permitido conocer *lo que hace* la clase, pero no *cómo* lo hace. Una clase será un *módulo* y un *tipo*. Como módulo la clase encapsula los recursos que ofrece a otras clases (sus clientes). Como *tipo* describe un conjunto de *objetos* o *instancias* que existen en tiempo de ejecución.

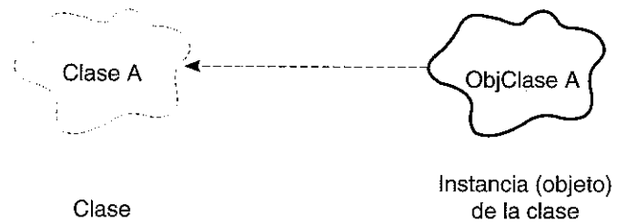


Figura 3.24. Una clase y una instancia (objeto) de la clase (Notación Booch).

Instancias como registros			
Servicio	**Horas**	**Frecuencia**	**Descuento**
S2020	4,5	6	10
S1010	8	2	20
S4040	5	3	15

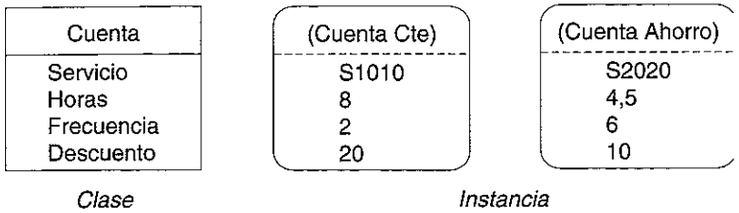


Figura 3.25. Instancias de una clase (notación OMT).

Los objetos ocupan espacio en memoria, y en consecuencia existen en el tiempo, y deberán *crearse o instanciarse* (probablemente a partir de otros objetos). Por la misma razón, se debe liberar el espacio en memoria ocupado por los objetos. Dos operaciones comunes típicas en cualquier clase son:

- *Constructor*: una operación que crea un objeto y/o inicializa su estado.
- *Destructor*: una operación que libera el estado de un objeto y/o destruye el propio objeto.

En C++ los constructores y destructores se declaran como parte de la definición de una clase. La creación se suele hacer a través de operaciones especiales (*constructores* en C++, `Pila`); estas operaciones se aplicarán implícitamente o se deberán llamar explícitamente por otros objetos, como sucede en C++.

Cuando se desea crear una nueva instancia de una clase, se llama a un método de la propia clase para realizar el proceso de construcción. Los métodos constructores se definen como métodos de la clase. De modo similar, los métodos empleados para destruir objetos y liberar la memoria ocupada (*destructores* en C++, `~Pila`) también se definen dentro de la clase.

> Un objeto es una *instancia* (ejemplar, caso u ocurrencia) de una clase.

3.9. HERENCIA Y TIPOS

Los objetos con propiedades comunes (atributos y operaciones) se clasifican en una clase. De igual modo, las clases con propiedades y funciones comunes se agrupan en una **superclase**. Las clases que se derivan de una superclase son las **subclases**.

Las clases se organizan como *jerarquía de clases*. La ventaja de definir clases en una jerarquía es que a través de un mecanismo denominado *herencia*, casos especiales comparten todas las características de sus casos más generales.

La herencia es una característica por la que es posible definir una clase, no de un borrador, sino en términos de otra clase. Una clase *hereda* sus características (datos y funciones) de otra clase. Esta característica proporciona cla-

ramente un soporte poderoso para reutilización y extensibilidad, dado que la definición de nuevos objetos se pueden basar en clases existentes.

Como ejemplo, considérese la jerarquía de herencia mostrada en la Figura 3.26.

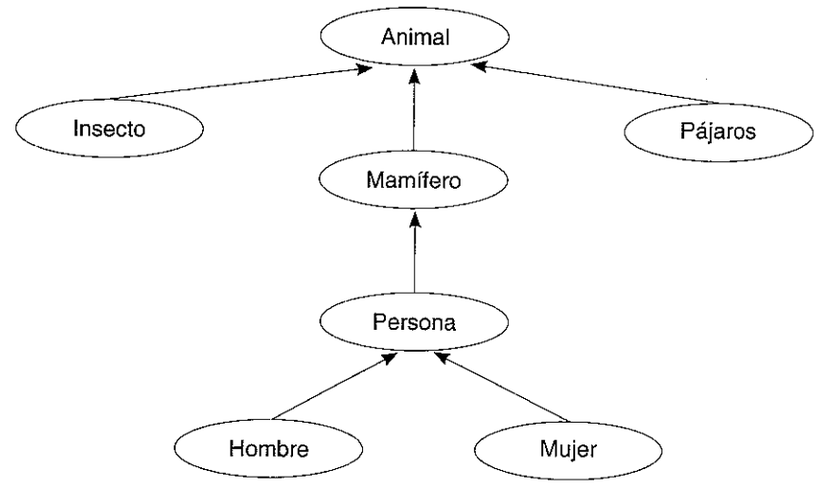


Figura 3.26. Jerarquía de herencia.

Las clases de objeto mamífero, pájaro e insecto se definen como *subclases* de animal; la clase de objeto persona, como una subclase de mamífero, y un hombre y una mujer son subclases de persona.

Las definiciones de clases para esta jerarquía puede tomar la siguiente estructura:

```
clase criatura
   atributos
      tipo : string;
      peso : real;
      habitat : (...algun tipo de habitat... );
   operaciones
      crear() → criatura;
      predadores(criatura) → fijar(criatura);
      esperanza_vida(criatura) → entero;
      ...
end criatura.

clase mamifero inherit criatura;
   atributos (propiedades)
      periodo_gestacion: real;
   operaciones
      ...
end mamifero.

clase persona inherit mamifero;
   propiedades
```

```
        apellidos, nombre: string;
        fecha_nacimiento: date;
        origen: pais;
end persona

clase hombre inherit persona;
   atributos
        esposa: mujer;
        ...
   operaciones
        ...
end hombre

clase mujer inherit persona;
   propiedades
        esposo: man;
        nombre: string;
        ..
end mujer
```

La herencia es un mecanismo potente para tratar con la evolución natural de un sistema y con modificación incremental [Meyer, 1988] Existen dos tipos diferentes de herencia: *simple* y *múltiple*.

3.9.1. Herencia simple (*herencia jerárquica*)

En esta jerarquía cada clase tiene como máximo una sola superclase. La herencia simple permite que una clase herede las propiedades de su superclase en una cadena jerárquica.

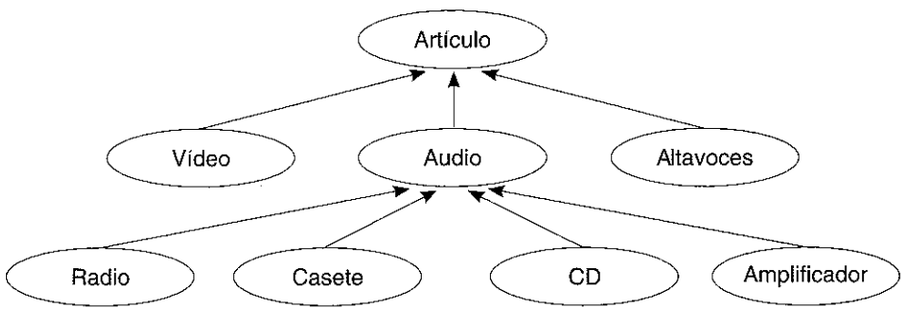


Figura 3.27. Herencia simple.

3.9.2. Herencia múltiple (*herencia en malla*)

Una malla o retícula consta de clases, cada una de las cuales puede tener uno o más superclases inmediatas Una herencia múltiple es aquella en la que cada clase puede heredar métodos y variables de cualquier número de superclase

En la Figura 3 28 la clase C tiene dos superclases, A y D Por consiguiente, la clase C hereda las propiedades de las clases A y D. Evidentemente, esta acción puede producir un conflicto de nombres, donde la clase C hereda las mismas propiedades de A y D.

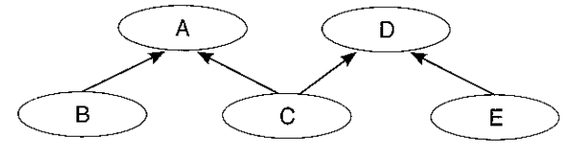


Figura 3.28. Herencia múltiple.

Herencia selectiva

La herencia selectiva es la herencia en que algunas propiedades de las superclases se heredan selectivamente por parte de la clase heredada Por ejemplo, la clase B puede heredar algunas propiedades de la superclase A, mientras que la clase C puede heredar selectivamente algunas propiedades de la superclase A y algunas de la superclase D

Herencia múltiple

Problemas:

1. La propiedad referida solo está en una de las subclases padre.
2. La propiedad concreta existe en más de una superclase.

Caso 1. No hay problemas.

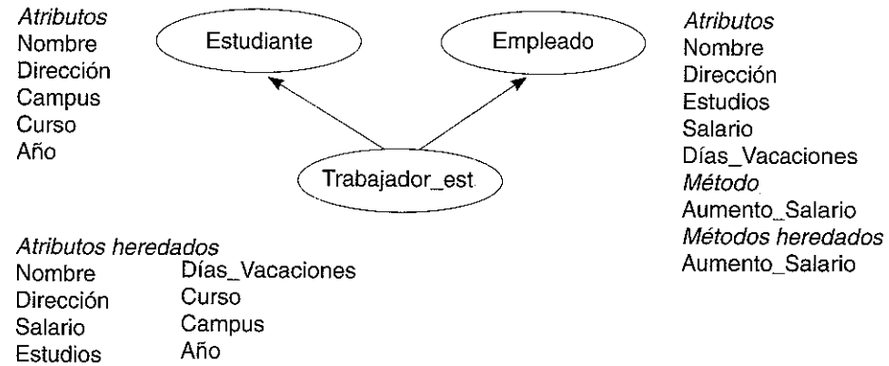


Figura 3.29. Herencia de atributos y métodos.

Caso 2. Existen diferentes tipos de conflictos que pueden ocurrir:

- Conflictos de nombres.

- Conflictos de valores
- Conflictos por defecto
- Conflictos de dominio
- Conflictos de restricciones

Por ejemplo,

Conflicto de nombres	Nombre Nombre_estudiante Nombre_empleado
Valores	Atributos con igual nombre, tienen valores en cada clase
	Universidad con diversos campus.

Reglas de resolución de conflictos

1. Una lista de precedencia de clases, como sucede en LOOPS y FLAVORS.
2. Una precedencia especificada por el usuario para herencia, como en Smalltalk.
3. Lista de precedencia del usuario, y si no sucede así, la lista de precedencia de las clases por profundidad.

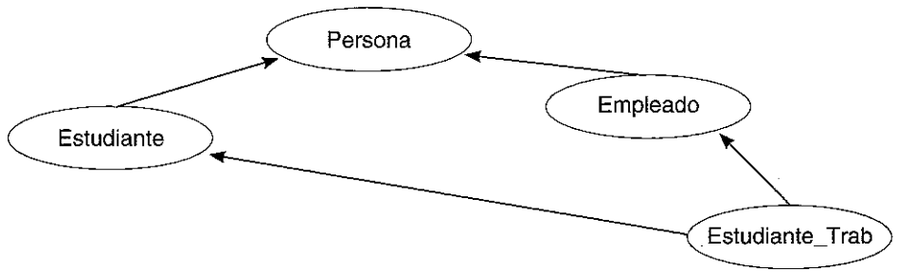


Figura 3.30. Clase derivada por herencia múltiple.

3.9.3. Clases abstractas

Con frecuencia, cuando se diseña un modelo orientado a objetos es útil introducir clases a cierto nivel que pueden no existir en la realidad pero que son construcciones conceptuales útiles. Estas clases se conocen como *clases abstractas*.

Una clase abstracta normalmente ocupa una posición adecuada en la jerarquía de clases que le permite actuar como un depósito de métodos y atributos compartidos para las subclases de nivel inmediatamente inferior.

Las clases abstractas no tienen instancias directamente. Se utilizan para agrupar otras clases y capturar información que es común al grupo. Sin embargo, las subclases de clases abstractas que corresponden a objetos del mundo real pueden tener instancias.

Una clase abstracta es COCHE_TRANSPORTE_PASAJEROS. Una subclase es SEAT, que puede tener instancias directamente, por ejemplo Coche1 y Coche2.

> Una clase abstracta es una clase que sirve como clase base común, pero no tendrá instancias.

Una clase abstracta puede ser una impresora.

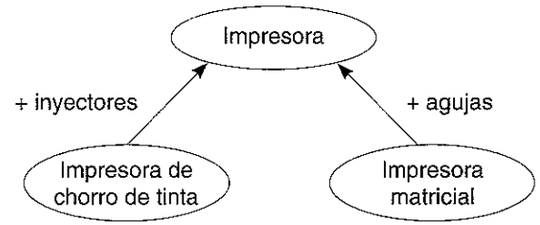


Figura 3.31. La clase abstracta impresora.

Las clases derivadas de una clase base se conocen como *clases concretas*, que ya pueden *instanciarse* (es decir, pueden tener *instancias*).

3.10. ANULACION/SUSTITUCION

Como se ha comentado anteriormente, los atributos y métodos definidos en la superclase se heredan por las subclases. Sin embargo, si la propiedad se define nuevamente en la subclase, aunque se haya definido anteriormente a nivel de superclase; entonces la definición realizada en la subclase es la utilizada en esa subclase. Entonces se dice que anulan las correspondientes propiedades de la superclase. Esta propiedad se denomina **anulación** o **sustitución** (*overriding*).

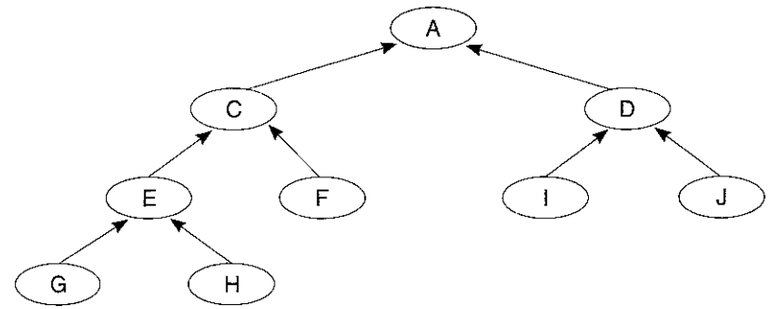


Figura 3.32. Anulación de atributos y métodos en clases derivadas.

Supongamos que ciertos atributos y métodos definidos en la clase A se redefinen en la clase C. Las clases E, F, G y H heredan estos atributos y métodos. La cuestión que se produce es si estas clases heredan las definiciones dadas en la clase A o las dadas en la clase C. El convenio adoptado es que una vez que un atributo o método se redefine en un nivel de clases específico, entonces cualquier hijo de esa clase, o sus hijos en cualquier profundidad, utilizan este método o atributo redefinido. Por consiguiente, las clases E, F, G y H utilizarán la redefinición dada en la clase C, en lugar de la definición dada en la clase A.

3.11. SOBRECARGA

La **sobrecarga** es una propiedad que describe una característica adecuada que utiliza el mismo nombre de operación para representar operaciones similares que se comportan de modo diferente cuando se aplican a clases diferentes. Por consiguiente, los nombres de las operaciones se pueden sobrecargar, esto es, las operaciones se definen en clases diferentes y pueden tener nombres idénticos, aunque su código programa puede diferir.

Si los nombres de una operación se utilizan para nuevas definiciones en clases de una jerarquía, la operación a nivel inferior se dice que anula la operación a un nivel más alto.

Un ejemplo se muestra en la Figura 3.33, en la que la operación `Incrementar` está sobrecargada en la clase `Empleado` y la subclase `Administrativo`. Dado que `Administrativo` es una subclase de `Empleado`, la operación `Incrementar`, definida en el nivel Administrativo, anula la operación correspondiente al nivel Empleado. En `Ingeniero` la operación `Incrementar` se hereda de `Empleado`. Por otra parte, la sobrecarga puede estar situada entre dos clases que no están relacionadas jerárquicamente. Por ejemplo, `Cálculo_Comisión` está sobrecargada en `Administrativo` y en `Ingeniero`. Cuando un mensaje `Calcular_Comisión` se envía al objeto `Ingeniero`, la operación correspondiente asociada con `Ingeniero` se activa.

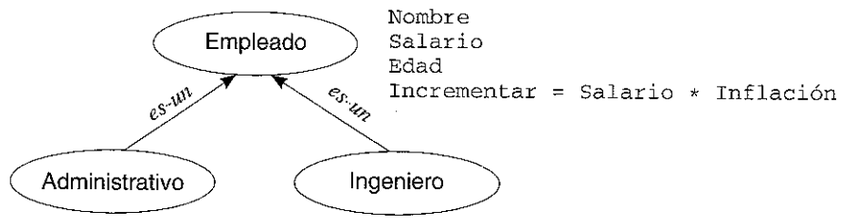


Figura 3.33. Sobrecarga.

Actualmente la sobrecarga se aplica sólo a operaciones. Aunque es posible extender la propiedad a atributos y relaciones específicas del modelo propuesto.

La sobrecarga no es una propiedad específica de los lenguajes orientados a objetos. Lenguajes tales como C y Pascal soportan operaciones sobrecargadas. Algunos ejemplos son los operadores aritméticos, operaciones de E/S y operadores de asignación de valores.

En la mayoría de los lenguajes, los operadores aritméticos «+», «−» y «*» se utilizan para sumar, restar o multiplicar números enteros o reales. Estos operadores funcionan incluso aunque las implementaciones de aritmética entera y real (coma flotante) sean bastante diferentes. El compilador genera código objeto para invocar la implementación apropiada basada en la clase (entero o coma flotante) de los operandos.

Así, por ejemplo, las operaciones de E/S (Entrada/Salida) se utilizan con frecuencia para leer números enteros, caracteres o reales. En Pascal `read(x)` se puede utilizar, siendo `x` un entero, un carácter o un real. Naturalmente, el código máquina real ejecutado para leer una cadena de caracteres es muy diferente del código máquina para leer enteros. `read(x)` es una operación sobrecargada que soporta tipos diferentes. Otros operadores tales como los de asignación («:=» en Pascal o «=» en C) son sobrecargados. Los mismos operadores de asignación se utilizan para variables de diferentes tipos.

Los lenguajes de programación convencionales soportan sobrecarga para algunas de las operaciones sobre algunos tipos de datos, como enteros, reales y caracteres. Los *sistemas orientados a objetos dan un poco más en la sobrecarga y la hacen disponible para operaciones sobre cualquier tipo objeto.*

Por ejemplo, en las operaciones binarias se pueden sobrecargar para números complejos, arrays, conjuntos o listas que se hayan definido como tipos estructurados o clases. Así, el operador binario «+» se puede utilizar para sumar las correspondientes partes reales e imaginarias de los números complejos. Si `A1` y `A2` son dos array de enteros, se pueden definir:

```
A := A1 + A2
```

para sumar:

```
A[i] := A1[i] + A2[i]       //para todo i
```

De modo similar, si `S1` y `S2` son dos conjuntos de objetos, se puede definir:

```
S := S1 + S2
```

como unión de dos conjuntos `S1` y `S2`.

¿Cómo se asocia una operación particular o mensaje a un método? La respuesta es mediante la ligadura dinámica (*dynamic binding*), que se verá posteriormente.

3.12. LIGADURA DINAMICA

Los lenguajes OO tienen la característica de poder ejecutar ligadura tardía (*dinámica*), al contrario que los lenguajes imperativos, que emplean ligadura temprana (*estática*). Por consiguiente, los tipos de variables, expresiones y funciones se conocen en tiempo de compilación para estos lenguajes imperativos. Esto permite el enlazar entre llamadas a procedimientos y los procedimientos utilizados que se establecen cuando se cumple el código. En un sistema OO esto requería el enlace entre mensajes y que los métodos se establezcan en tiempo dinámico.

En el caso de ligadura dinámica o tardía, el tipo se conecta directamente al objeto. Por consiguiente, el enlace entre el mensaje y el método asociado sólo se puede conocer en tiempo de ejecución.

La ligadura estática permite un tiempo de ejecución más rápido que la ligadura dinámica, que necesita resolver estos enlaces en tiempo de ejecución. Sin embargo, en ligadura estática se ha de especificar en tiempo de compilación las operaciones exactas a que responderá una invocación del método o función específica, así como conocer sus tipos.

Por el contrario, en la ligadura dinámica simplemente se especifica un método en un mensaje, y las operaciones reales que realizan este método se determinan en tiempo de ejecución. Esto permite definir funciones o métodos virtuales.

3.12.1. Funciones o métodos virtuales

Las funciones virtuales en C++ permiten especificar un método como virtual en la definición de una clase particular. La implementación real del método se

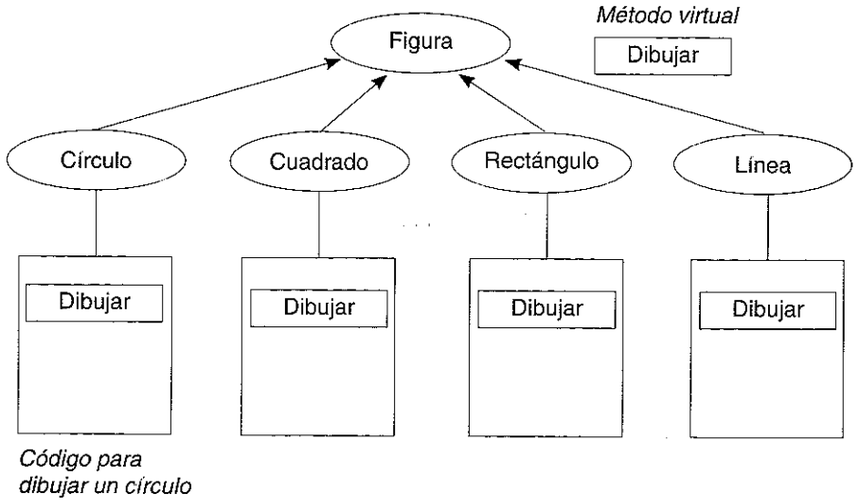


Figura 3.34. Métodos o funciones virtuales.

realiza en las subclases. En este caso, por consiguiente, la selección del método se hace en tiempo de compilación, pero el código real del método utilizado se determina utilizando ligadura dinámica o tardía en tiempo de compilación.

Esto permite definir el método de un número de formas diferentes para cada una de las diferentes clases. Consideremos la jerarquía de clases definida en la Figura 3.34.

Aquí el método virtual se define en la clase `FIGURA` y el código procedimental real utilizado se define en cada una de las subclases `CIRCULO`, `CUADRADO`, `RECTANGULO` y `LINEA`. Ahora, si un mensaje se envía a una clase específica, se ejecuta el código asociado con ella. Esto contrasta con un enfoque más convencional que requiere definir los procedimientos por defecto, con nombres diferentes, tales como `Dibujar_círculo`, `Dibujar_cuadrado`, etc. También se requerirá utilizar una llamada al nombre de la función específica cuando sea necesario.

3.12.2. Polimorfismo

La capacidad de utilizar funciones virtuales y ejecutar sobrecarga conduce a una característica importante de los sistemas OO, conocida como *polimorfismo*, que esencialmente permite desarrollar sistemas en los que objetos diferentes puedan responder de modo diferente al mismo mensaje.

En el caso de un método virtual se puede tener especialización incremental de, o adición incremental a, un método definido anteriormente en la jerarquía.

Más adelante volveremos a tratar este concepto.

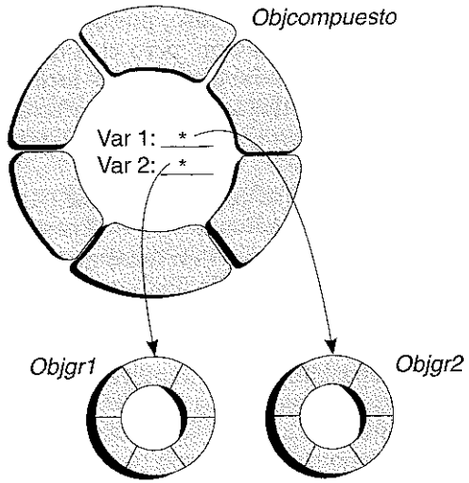
3.13. OBJETOS COMPUESTOS

Una de las características que hacen a los objetos ser muy potentes es que pueden contener otros objetos. Los objetos que contienen otros objetos se conocen como *objetos compuestos*.

En la mayoría de los sistemas, los objetos compuestos no «contienen» en el sentido estricto otros objetos, sino que contienen variables que se refieren a otros objetos. La referencia almacenada en la variable se llama identificador del objeto (ID del objeto).

Esta característica ofrece dos ventajas importantes:

1. Los objetos «contenidos» pueden cambiar en tamaño y composición, sin afectar al objeto compuesto que los contiene. Esto hace que el mantenimiento de sistemas complejos de objetos anidados sea más sencillo, que sería el caso contrario.
2. Los objetos contenidos están libres para participar en cualquier número de objetos compuestos, en lugar de estar bloqueado en un único objeto compuesto.



Notación TAYLOR [6]

Figura 3.35. Un objeto compuesto.

Un objeto compuesto consta de una colección de dos o más objetos componentes. Los objetos componentes tienen una relación **part-of** (*parte-de*) o **component-of** (*componente-de*) con objeto compuesto. Cuando un objeto compuesto se instancia para producir una instancia del objeto, todos sus objetos componentes se deben instanciar al mismo tiempo. Cada objeto componente puede ser a su vez un objeto compuesto [6].

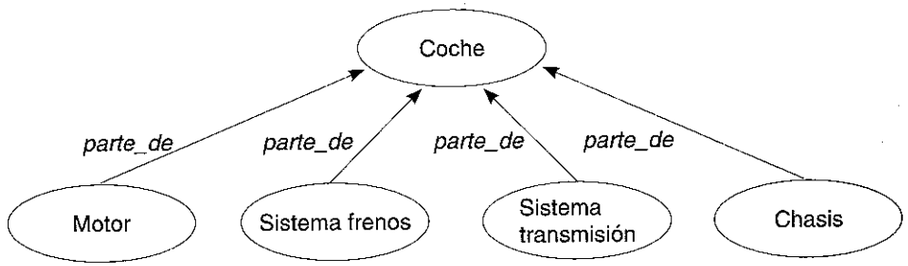


Figura 3.36. Relación de agregación (*parte-de*).

La relación *parte-de* puede representarse también por **has-a** (*tiene-un*), que indica la relación que une al objeto *agregado* o *continente*. En el caso del objeto compuesto COCHE se leerá: COCHE *tiene-un* MOTOR, *tiene-un* SISTEMA_DE_FRENOS, etc.

[6] TAYLOR, David: *Object-Oriented Information System*. Wiley, 1992.

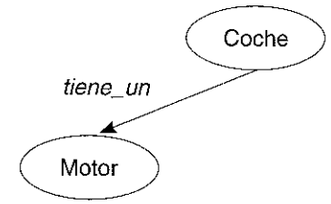


Figura 3.37. Relación de agregación (*tiene-un*).

3.13.1. Un ejemplo de objetos compuestos

La ilustración siguiente muestra dos objetos que representa órdenes de compra. Sus variables contienen información sobre clientes, artículos comprados y otros datos. En lugar de introducir toda la información directamente en los objetos `orden_compra`, se almacenan referencia a estos objetos componentes en el formato del identificador de objeto (IDO).

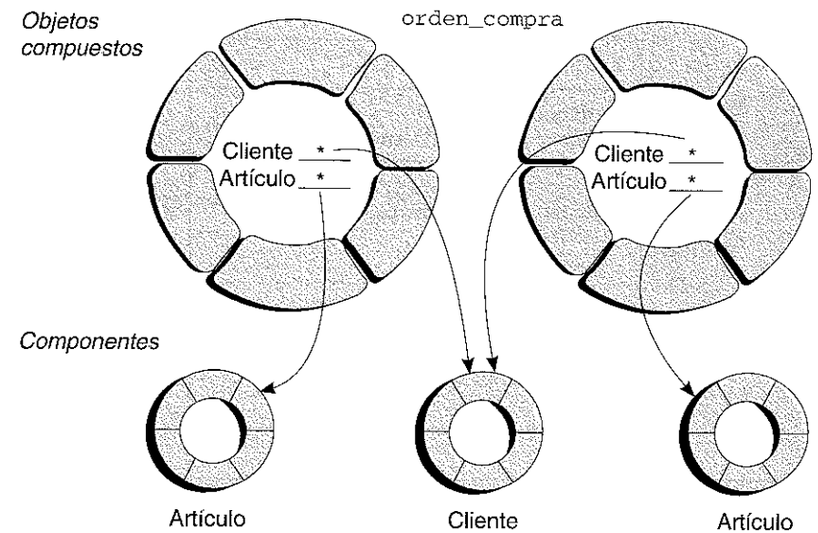


Figura 3.38. Objetos compuestos (dos objetos `orden_compra`[7]).

3.13.2. Niveles de profundidad

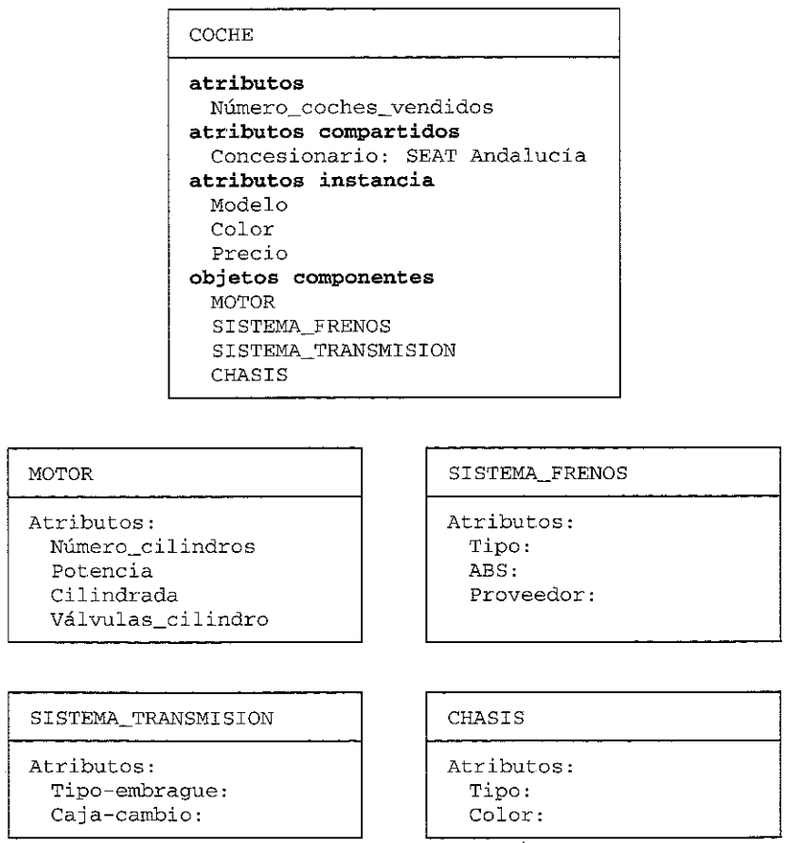
Los objetos contenidos en objetos compuestos pueden, por sí mismos, ser objetos compuestos, y este anidamiento puede ir hasta cualquier profundidad. Esto

[7] Este ejemplo está citado en: TAYLOR, David: *op. cit.* pág. 45. Asimismo, la notación de objetos empleada por Taylor se ha mantenido en varios ejemplos de nuestra obra, ya que la consideramos una de las más idóneas para reflejar el concepto de objeto; esta obra y otras suyas sobre el tema son consideradas como aportaciones muy notables al mundo científico de los objetos.

significa que puede construir estructuras de cualquier complejidad conectando objetos juntos. Esto es importante debido a que normalmente se necesita más de un nivel de modularización para evitar el caos en sistemas a gran escala.

Un objeto compuesto, en general, consta de una colección de dos o más objetos relacionados conocidos como objetos componentes. Los objetos componentes tienen una relación una *parte-de* o un *componente-de* con objeto compuesto. Cuando un objeto compuesto se instancia para producir un objeto instancia, todos sus objetos componentes se deben instanciar al mismo tiempo. Cada objeto componente puede, a su vez, ser un objeto compuesto, resultando, por consiguiente, una jerarquía de *componentes-de*.

Un ejemplo de un objeto compuesto es la clase COCHE. Un coche consta de diversas partes, tales como un motor, un sistema de frenos, un sistema de transmisión y un chasis; se puede considerar como un objeto compuesto que consta de partes diferentes: MOTOR, SISTEMA_FRENOS, SISTEMA_TRANSMISION, CHASIS. Estas partes constituyen los objetos componentes del objeto COCHE, de modo que cada uno de estos objetos componentes pueden tener atributos y métodos que los caracterizan.



La jerarquía *componente-de* (*parte-de*) pueden estar solapadas o anidadas. Una jerarquía de solapamiento consta de objetos que son componentes de más de un objeto padre.

Una jerarquía anidada consta de objetos que son componentes de un objeto padre que, a su vez, puede actuar como componente de otro objeto. El objeto Z es un componente del objeto B, y el objeto B es un componente de un objeto complejo más grande, A.

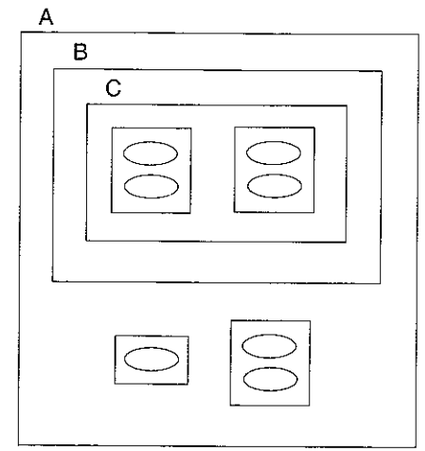


Figura 3.39. Jerarquía de componentes agregados.

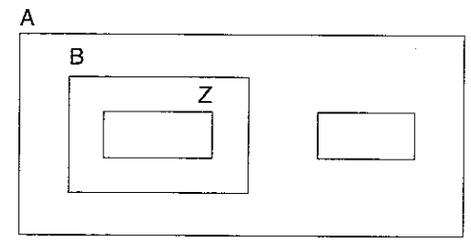


Figura 3.40. Anidamiento de objetos.

Un ejemplo típico de un objeto compuesto anidado es un archivador. Un archivador contiene cajones, un cajón contiene carpetas y una carpeta contiene documentos. El ejemplo COCHE, citado anteriormente, es también un objeto compuesto anidado.

3.14. REUTILIZACION CON ORIENTACION A OBJETOS

Reutilización o reutilizabilidad es la propiedad por la que el software desarrollado puede ser utilizado cuantas veces sea necesario en más de un programa. Así por ejemplo, si se necesita una función que calcule el cuadrado o el cubo de un

número, se puede crear la función que realice la tarea que el programa necesita. Con un esfuerzo suplementario se puede crear una función que pueda elevar cualquier número a cualquier potencia. Esta función se debe guardar para poderla utilizar como herramienta de propósito general en cuantas ocasiones sea necesario.

Las ventajas de la reutilización son evidentes. El ahorro de tiempo es, sin duda, una de las ventajas más considerables, y otra la facilidad para intercambiar software desarrollado por diferentes programadores.

En la programación tradicional, las bibliotecas de funciones (casos de FORTRAN o C) evitan tener que ser escritas cada vez que se necesita su uso.

Ada y Modula-2 incorporan el tipo de dato *paquete* (**package**) y *módulo* (**module**) que consta de definición de tipos y códigos y que son la base de la reutilización de esos lenguajes.

3.14.1. Objetos y reutilización

La programación orientada a objetos proporciona el marco idóneo para la reutilización de las clases. Los conceptos de encapsulamiento y herencia son las bases que facilitan la reutilización. Un programador puede utilizar una clase existente, y sin modificarla, añadirle nuevas características y datos. Esta operación se consigue derivando una clase a partir de la clase base existente. La nueva clase hereda las propiedades de la antigua, pero se pueden añadir nuevas propiedades. Por ejemplo, suponga que se escribe (o compra) una clase menú que crea un sistema de menús (barras de desplazamiento, cuadros de diálogo, botones, etc.); con el tiempo, aunque la clase funciona bien, observa que sería interesante que las leyendas de las opciones de los menús parpadearán o cambiarán de color. Para realizar esta tarea se diseña una clase derivada de menú que añada las nuevas propiedades de parpadeo o cambio de color.

La facilidad para reutilizar clases (y en consecuencia objetos) es una de las propiedades fundamentales que justifican el uso de la programación orientada a objetos. Por esta razón los sistemas y en particular los lenguajes orientados a objetos suelen venir provistos de un conjunto (*biblioteca*) de clases predefinidas, que permite ahorrar tiempo y esfuerzo en el desarrollo de cualquier aplicación. Esta herramienta —*la biblioteca de clases*— es uno de los parámetros fundamentales a tener en cuenta en el momento de evaluar un lenguaje orientado a objetos.

3.15. POLIMORFISMO

Otra propiedad importante de la programación orientada a objetos es el *polimorfismo*. Esta propiedad, en su concepción básica, se encuentra en casi todos los lenguajes de programación. El polimorfismo, en su expresión más simple, es el uso de un nombre o un símbolo —por ejemplo un operador— para representar o significar más de una acción. Así, en C, Pascal y FORTRAN —entre otros lenguajes— los operadores aritméticos representan un ejemplo de esta característica. El símbolo +, cuando se utiliza con enteros, representa un conjunto de instrucciones máquina distinto de cuando los operadores son valores reales de doble precisión. De igual modo, en algunos lenguajes el símbolo + sirve para realizar sumas aritméticas o bien para concatenar (unir) cadenas.

La utilización de operadores o funciones de formas diversas, dependiendo de cómo se estén operando, se denomina *polimorfismo* (múltiples formas). Cuando un operador existente en el lenguaje tal como +, = o * se le asigna la posibilidad de operar sobre un nuevo tipo de dato, se dice que está *sobrecargado*. La *sobrecarga* es una clase de polimorfismo, que también es una característica importante de POO. Un uso típico de los operadores aritméticos es la sobrecarga de los mismos para actuar sobre tipos de datos definidos por el usuario (objetos), además de sobre los tipos de datos predefinidos. Supongamos que se tienen tipos de datos que representan las posiciones de puntos en la pantalla de un computador (coordenadas x e y). En un lenguaje orientado a objetos se puede realizar la operación aritmética

```
posición1 = origen + posición2
```

donde las variables *posición1*, *posición2* y *origen* representan cada una posiciones de puntos, sobrecargando el operador más (+) para realizar suma de posiciones de puntos (x, y). Además de esta operación de suma se podrían realizar otras operaciones, tales como resta, multiplicación, etc., sobrecargando convenientemente los operadores −, *, etc.

En un sentido más general, el polimorfismo supone que un mismo mensaje puede producir acciones (resultados) totalmente diferentes cuando se reciben por objetos diferentes. Con polimorfismo un usuario puede enviar un mensaje genérico y dejar los detalles de la implementación exacta para el objeto que recibe el mensaje. El polimorfismo se fortalece con el mecanismo de herencia.

Supongamos un tipo objeto llamado `vehículo` y tipos de objetos derivados llamados `bicicleta`, `automóvil`, `moto` y `embarcación`. Si se envía un mensaje conducir al objeto `vehículo`, cualquier tipo que herede de `vehículo` puede también aceptar ese mensaje. Al igual que sucede en la vida real, el mensaje `conducir` reaccionará de modo diferente en cada objeto, debido a que cada vehículo requiere una forma distinta de conducir.

RESUMEN

El tipo abstracto de datos se implementa a través de clases. Una clase es un conjunto de objetos que constituyen instancias de la clase, cada una de las cuales tienen la misma estructura y comportamiento. Una clase tiene un nombre, una colección de operaciones para manipular sus instancias y una representación. Las operaciones que manipulan las instancias de una clase se llaman *métodos*. El estado o representación de una instancia se almacena en variables de instancia. Estos métodos se invocan mediante el envío de *mensajes* a instancias. El envío de mensajes a objetos (instancias) es similar a la llamada a procedimientos en lenguajes de programación tradicionales.

El mismo nombre de un método se puede sobrecargar con diferentes implementaciones; el método `Imprimir` se puede aplicar a enteros, arrays y cadenas de caracteres. La sobrecarga de operaciones permite a los programas ser extendidos de un modo elegante. La sobrecarga permite la ligadura de un mensaje a la implementación de código del mensaje y se hace en tiempo de ejecución. Esta característica se llama ligadura dinámica.

El *polimorfismo* permite desarrollar sistemas en los que objetos diferentes pueden responder de modo diferente al mismo mensaje. La ligadura dinámica, sobrecarga y la herencia permite soportar el polimorfismo en lenguajes de programación orientados a objetos.

Los programas orientados a objetos pueden incluir *objetos compuestos*, que son objetos que contienen otros objetos, anidados o integrados en ellos mismos.

Los principales puntos clave tratados son:

- La programación orientada a objetos incorpora estos seis componentes importantes:

  Objetos
  Clases.
  Métodos.
  Mensajes.
  Herencia.
  Polimorfismo

- Un objeto se compone de datos y funciones que operan sobre esos objetos.
- La técnica de situar datos dentro de objetos de modo que no se puede acceder directamente a los datos se llama *ocultación de la información*
- Una clase es una descripción de un conjunto de objetos. Una instancia es una variable de tipo objeto y un objeto es una instancia de una clase.
- La herencia es la propiedad que permite a un objeto pasar sus propiedades a otro objeto, o dicho de otro modo, un objeto puede heredar de otro objeto
- Los objetos se comunican entre sí pasando mensajes.
- La clase padre o ascendiente se denomina clase base y las clases descendientes clases derivadas.
- La reutilización de software es una de las propiedades más importantes que presenta la programación orientada a objetos.
- El polimorfismo es la propiedad por la cual un mismo mensaje puede actuar de diferente modo cuando actúa sobre objetos diferentes ligados por la propiedad de la herencia.

CAPITULO **4**

LENGUAJES DE PROGRAMACION ORIENTADOS A OBJETOS

CONTENIDO

Este capítulo describe:

- La evolución de los lenguajes de programación orientados a objetos, desde Simula a Object COBOL.
- Las características de un lenguaje orientado a objetos.
- La clasificación de los lenguajes orientados a objetos basados en objetos, orientados a objetos (*puros* e *híbridos*) basados en objetos, orientados a objetos basados en clases.
- Una descripción breve de los lenguajes más utilizados: Smalltalk, C++, Objective-C, Eiffel, Object Pascal, Visual BASIC y Ada.

4.1. EVOLUCION DE LOS LPOO

Al principio de la década de los sesenta, Kristen Nygaard y Ole-Johan Dhal desarrollaron Simula en el Norwegian Computer Center (NCC), un lenguaje que soportaba modelización para simulación de procesos industriales y científicos. Fue un lenguaje de propósito general que ofrecía capacidad de simulación de sistemas. Sin embargo, sus usuarios descubrieron pronto que Simula proporcionaba nuevas y poderosas posibilidades para *fines distintos* de la simulación, como la elaboración de prototipos y el diseño de aplicaciones. Las sucesivas versiones de Simula fueron mejorándose hasta llegar a Simula-67 (aparición en diciembre de 1966). Simula-67 se derivaba de Algol 60, donde tiene sus raíces, y se diseñó fundamentalmente como un lenguaje de programación de diseño; tomó de Algol el concepto de bloque e introdujo el concepto de *objeto*. Los objetos de Simula tenían su propia existencia y «podían» comunicarse entre sí durante un proceso de simulación. Conceptualmente, un objeto contenía tanto datos como las operaciones que manipulaban esos datos. Las operaciones se llamaron *métodos*. Simula incorporaba también la noción de *clases*, que se utilizaron para describir la estructura y comportamiento de un conjunto de clases. La herencia de clases fue también soportada por Simula. La herencia organiza las clases en jerarquías, permitiendo la compartimentación de implementación y estructura. En esencia, Simula sentó la base de los lenguajes orientados a objetos y definió algunos de los conceptos clave de la orientación a objetos. Además, Simula era un lenguaje «fuertemente tipificado». Esto significa que el tipo de cada variable se conoce en tiempo de compilación, de modo que los errores que implican tipos se encuentran en esta etapa y no cuando el programa se ejecuta.

Simula-67 fue el primer lenguaje de programación que incorporó mecanismos para soportar los conceptos orientados a objetos más sobresalientes:

- *Encapsulamiento y objetos*, que agrupan juntos atributos de datos y acciones (métodos) que procesen esos datos.
- *Verificación estática de tipos*, que se realiza durante el proceso de compilación para proporcionar seguridad en tiempo de ejecución para la manipulación externa de los atributos de los objetos.
- *Clases*, como plantillas o patrones de objetos.
- *Herencia*, como medio de agrupar clases con propiedades comunes.
- *Ligadura dinámica (polimorfismo)*, para permitir a las clases de objetos que tengan interfaces idénticos y propiedades que se puedan intercambiar.

La Figura 4.1 muestra la evolución (*genealogía*) de los lenguajes de programación orientados a objetos **(LPOO)**. En esta figura se muestra que Simula-67 (Simula) inspiró el desarrollo de Smalltalk, que es el verdadero primer lenguaje de programación orientado a objetos.

Alan Kay creó Smalltalk-80 en Xerox PARC (Xerox Palo Alto Research Park) con Adele Goldberg, que previamente había trabajado con una implementación de Simula. Smalltalk es un lenguaje orientado a objetos *puro*, *«todos es*

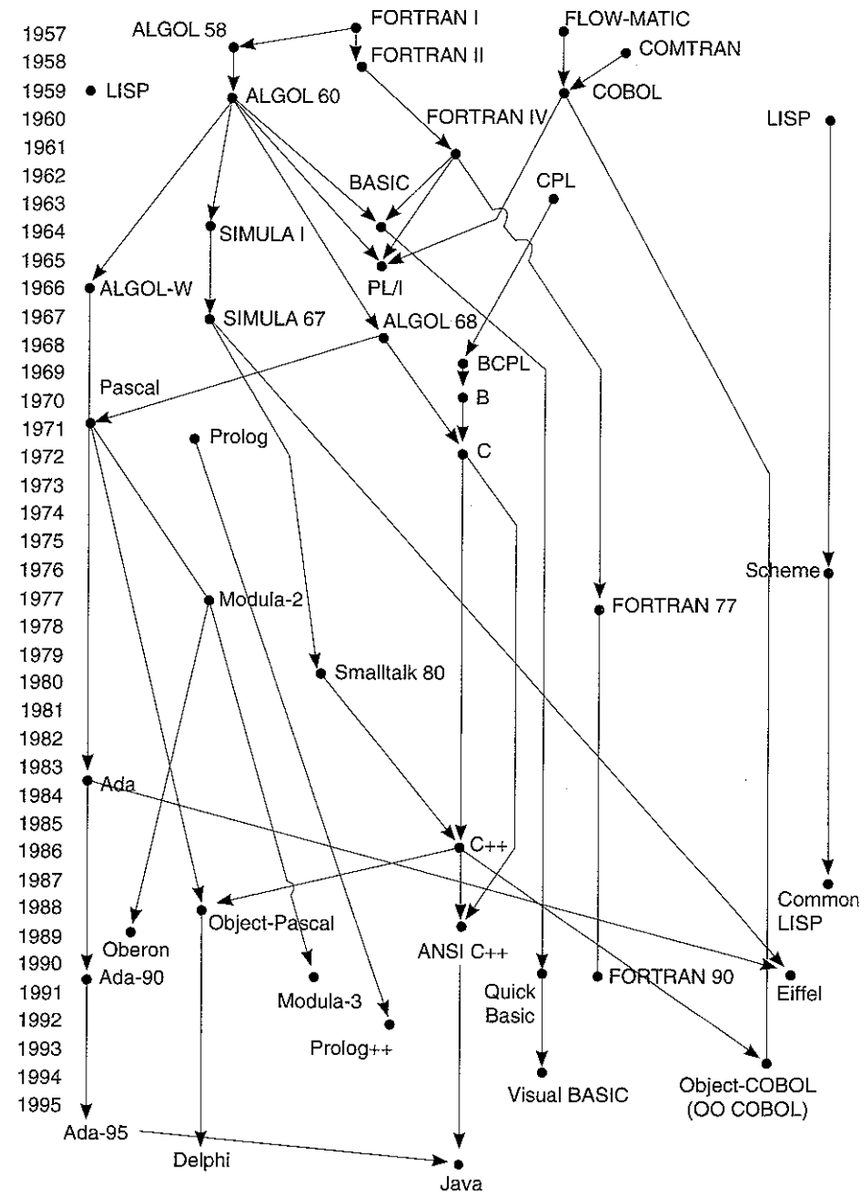


Figura 4.1. Genealogía de los lenguajes de objetos según Sebesta[1].

[1] Esta genealogía ha sido extraída y modificada ligeramente con lenguajes de los noventa, de: Robert W. Sebesta, *Concepts of Programming Languages*. Addison-Wesley, 2nd. edn. 1993

un objeto» de una clase y todas las clases heredan de una única clase base denominada `Object`. Smalltalk afirmó el término *«método»* para describir las acciones realizadas por un objeto y el concepto de *«paso de mensajes»* como el medio para activar «métodos». Es también un lenguaje tipificado dinámicamente, que *liga* (enlaza) un método a un mensaje en tiempo de ejecución

Smalltalk ha sido, a su vez, el *inspirador* de un gran número de lenguajes OO. Entre ellos destaquemos Eiffel, Smalltalk-80, Smalltalk/V, C++, Actor, Objective-C y CLOS, así como extensiones OO de lenguajes tradicionales, tales como Object Pascal, Object COBOL, etc.

Bertran Meyer, el diseñador de Eiffel, fue también un usuario de Simula, e incluso llegó a ser presidente de la Asociación de Usuarios de Simula. Jean Ichbiah, el diseñador jefe de Ada-83, dirigió un equipo que implementó un subconjunto de Simula, y Bjarne Stroustrup, el diseñador de C++, utilizó Simula y siempre ha agradecido su influencia en el diseño de C++.

Existen varias versiones y dialectos de Smalltalk: Smalltalk-72, -74, -76, -78, -80, y más recientemente —y seguramente la más popular— Smalltalk/V de Digitalk. Smalltalk no es un lenguaje tipificado. Smalltalk es extraordinariamente rico en conceptos orientados a objetos. En Smalltalk todo es un objeto, incluyendo clases base y tipo base. Esto significa que la potencia de Smalltalk, como un entorno de programación completo, se fundamenta en el envío de mensajes a objetos.

Otra característica fundamental que diferencia a Smalltalk es su capacidad de concurrencia. La *concurrencia* es un aspecto del mundo real. Por ejemplo, en un entorno de oficina, secretarias, administrativos, gerentes y otros empleados funcionan simultánea e independientemente. Se comunican entre sí, a través de conversaciones, informes, correo electrónico, etc. Smalltalk empleó la construcción denominada *proceso* para soportar concurrencia.

Durante la década de los ochenta, los conceptos orientados a objetos (tipos abstractos de datos, herencia, identidad de objetos y concurrencia), Smalltalk, Simula y otros lenguajes comenzaron a mezclarse y producir nuevos lenguajes orientados a objetos, así como extensiones y dialectos. El desarrollo de lenguajes de orientación a objetos en esa década se muestra en la siguiente clasificación:

1. *Extensiones, dialectos y versiones de Smalltalk.* Xerox y Textronix incorporaron en sus máquinas, a principios de los ochenta, la versión Smalltalk-80, que posteriormente se fue incorporando a otras muchas plataformas. Digitalk lanzó Smalltalk/V para computadoras personales IBM y compatibles.
2. *Extensiones orientadas a objetos de lenguajes convencionales.* Uno de los lenguajes orientados a objetos más populares es C++. Este lenguaje fue diseñado por Bjarne Stroustrup en AT&T al principio de los ochenta [Stroustrup, 1986]. La primera implementación del lenguaje C++ se lanzó como un preprocesador a los compiladores C. C++ proporcionó dos construcciones para definiciones de clases. El primer método es una extensión de la construcción `struct` (*estructura* de C) y la otra nueva construcción `class` (*clase*). C++ incorporó jerarquía de clases y permitía subclases que podían acceder a métodos y variables instancias de otras clases de su jerarquía. El lenguaje C++ permitía la ligadura dinámica y el polimorfismo, así como sobrecarga de funciones y operaciones. Sin embargo, al contrario que Smalltalk, las primeras versiones de C++ no se comercializaban en bibliotecas grandes de clases predefinidas.

   Otro dialecto importante de C+, con propiedades orientadas a objetos, es Objective-C [Cox, 1987]. Este lenguaje es un superconjunto de C, que incorpora características orientadas a objetos de Smalltalk. Al igual que Smalltalk, Objective-C incluía una gran colección de clases predefinidas que permitía simplificar el proceso de desarrollo. Objective-C soportaba tipos abstractos de datos, herencia y sobrecarga de operadores. Sin embargo, al contrario que C++, no ampliaba la definición de construcciones existentes en lenguajes, y diseñó nuevas construcciones y operadores para realizar tareas tales como definición de clases y paso de mensajes. Las computadoras NEXT, cuyo éxito no pasó de unos años, eligieron Objective-C como su principal lenguaje de desarrollo.

   Niklaus Wirt y un grupo de ingenieros informáticos de Apple Computer diseñaron Object Pascal [Schmucker, 1986]. Extendió el lenguaje Pascal con soporte para tipos abstractos de datos, métodos y herencia. Incluyó el concepto de tipo de *objeto* y definición de clases.

   Los nuevos lenguajes **Ada-95** y **Java** son totalmente orientados a objetos.
3. *Lenguajes orientados a objetos fuertemente tipificados.* Simula fue uno de los lenguajes orientados a objetos que se desarrollaron en la década de los ochenta y que fueron implementados en diversas plataformas.

   Otros lenguajes han emergido en la década de los ochenta, con características fuertemente tipificadas (con verificación estricta de tipos). Un lenguaje disponible comercialmente y más interesante es Eiffel [Meyer, 1988] de Interactive Software Engineering, Inc. Además de encapsulamiento y herencia, Eiffel incorpora muchas características orientadas a objetos, tales como *tipos paramétricos* y *pre* y *post-condiciones* para métodos. Otro lenguaje fuertemente tipificado que soporta conceptos orientados a objetos (abstracción, objetos, tipos paramétricos) es Ada, aunque la versión Ada-83 presenta el inconveniente de *no soportar herencia*.
4. *Extensiones orientadas a objetos de LISP.* Existen diferentes versiones de LISP, aunque la más conocida y notable es CLOS (Common List Object System). CLOS es un lenguaje OO que introduce notables mejoras y tiene garantizada larga vida, especialmente desde la creación del comité X3J13 de ANSI para la estandarización del lenguaje.

4.1.1. Estado actual de los lenguajes orientados a objetos en la década de los noventa

La década de los ochenta lanzó la orientación a objetos como base de la futura ingeniería de software orientada a objetos de la década de los noventa. En 1982 se predijo que la programación orientada a objetos sería en la década de los

ochenta lo que la programación estructurada fue en los setenta [Rentsch, 1982] La profecía se cumplió y la década de los ochenta se consagró como el origen de la explosión de la orientación a objetos que se produciría en la década de los noventa.

Sin duda, el desarrollo de conferencias internacionales sobre orientación a objetos y, en especial, OOSPLA (**O**bject-**O**riented **P**rogramming **S**ystems and **LA**nguages) han sido los detonantes de la explosión de la OO en la década de los noventa. La primera conferencia se celebró en el año 1986.

Otros hitos que han influido considerablemente en el enorme desarrollo de los LPOO han sido la aparición de diferentes publicaciones periódicas exclusivamente dedicadas a orientación a objetos. En 1988 apareció la primera revista de prestigio: *The Journal of Object-Oriented Programming*

En la década de los noventa, los lenguajes, técnicas, interfaces gráficos y bases de datos se están haciendo muy populares Sin duda, los noventa será la década de la proliferación de tecnologías y lenguajes orientados a objetos. Microsoft, IBM, Borland, Sun, AT&T, Digitalk, Symantec y otras grandes compañías están lanzando productos orientados a objetos de modo continuo y progresivo.

Los lenguajes más implantados en la actualidad son Smalltalk y Eiffel, junto con C++, Object Pascal (Turbo/Borland Pascal especialmente), Visual BASIC y Object Visual como lenguajes híbridos

C++ es, sin lugar a dudas, el lenguaje más popular, aunque Smalltalk está ganando adeptos día a día. Tanto C++ como Smalltalk han sido implementados en diferentes plataformas: DOS, UNIX, OS/2, Windows e incluso en sistemas grandes; su importancia reside en la gran cantidad de desarrolladores y vendedores que comercializan estos lenguajes.

4.2. CLASIFICACION DE LENGUAJES ORIENTADOS A OBJETOS

Como ya se ha comentado anteriormente, los lenguajes de programación orientados a objetos son: Eiffel, Lisp, Prolog, Simula, Smalltalk, C++, Object Pascal, etc Se entienden por **lenguajes orientados a objetos** aquellos que soportan las características de orientación a objetos Otros lenguajes de programación, tales como Ada, C, Cliper, pueden implementar algunas características orientadas a objetos, utilizando ciertas técnicas de programación, pero no se consideran orientados a objetos

Los principales lenguajes de programación utilizados actualmente para sistemas de tiempo real son C y Ada. Ada fue diseñado específicamente para la implementación de sistemas en tiempo real, especialmente empotrados. Aunque Ada (Ada-83) no cumple las propiedades importantes de un LPOO (por ejemplo herencia y ligadura dinámica), soporta un enfoque de diseño orientado a objetos y se le conoce usualmente como *basado en objetos* [Wegner, 87]. Ada-95, ya estandarizado por ISO y ANSI, soporta herencia y ligadura dinámica; en consecuencia, aunque todavía con restricciones, se considera orientado a objetos.

4.2.1. Taxonomía de lenguajes orientados a objetos

Una taxonomía de lenguajes de programación con propiedades de orientación a objetos fue creada por Wegner[2] La clasificación incluye los siguientes grupos:

1. **Basado en objetos**. Un lenguaje de programación es basado en objetos si su sintaxis y semántica soportan la creación de objetos que tienen las propiedades descritas en los capítulos anteriores
2. **Basado en clases**. Si un lenguaje de programación es basado en objetos y soporta además la creación de clases, se considera basado en clases.
3. **Orientación a objetos**. Un lenguaje de programación orientado a objetos es un lenguaje basado en clases que soporta también herencia.

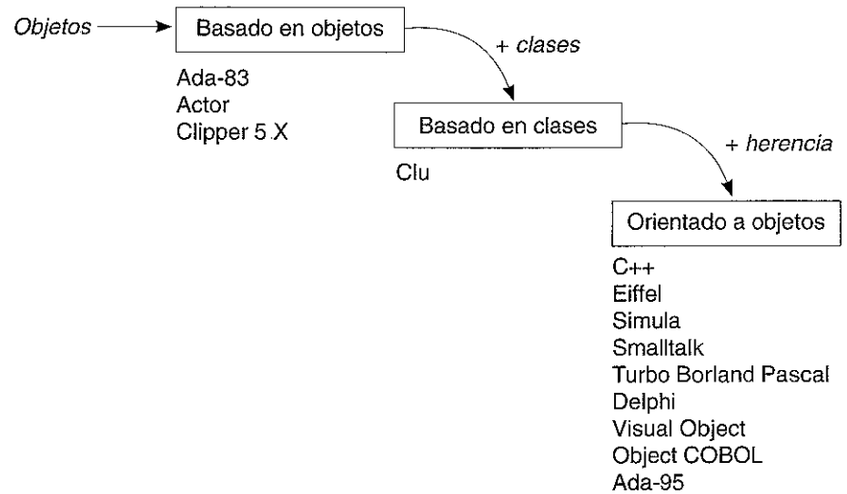


Figura 4.2. Taxonomía de lenguajes OO de Wegner.

Esta taxonomía de orientación a objetos proporciona una definición estricta de los lenguajes de programación orientados a objetos, que ha prevalecido en la época actual. Según esta taxonomía, no es suficiente que un lenguaje soporte la creación de objetos; para ser considerado *orientado a objetos*, es necesario que existan construcciones de creación de clases y que soporten herencia adecuadamente.

C++ soporta la creación de objetos y clases, así como herencia, y es por consiguiente totalmente orientado a objetos. Ada-83 soporta la creación de objetos mediante paquetes (tipos abstractos de datos). Un paquete en Ada no

[2] WEGNER, P.: *The Object-Oriented Classification Paradigm in Research Directions in Object-Oriented Programming* págs. 508-510. MIT Press, Cambridge, MA 1987

es una definición de tipos como la clase C++, y en consecuencia Ada es un lenguaje basado en objetos Ada-95 soporta, además de las propiedades de Ada-83, clases y herencia, y se puede considerar orientado a objetos.

De acuerdo a la taxonomía de Wegner, se podría actualizar la clasificación de los lenguajes pensando en la segunda década de los noventa:

Lenguajes basados en objetos	Ada-83, Actor, Clipper 5.2, Visual Basic 4.
Lenguajes basados en clases	Clu
Lenguajes orientados a objetos	C++, Objective-C, Object Pascal, Delphi, Visual Object, Object COBOL, Oveion Eiffel, Smalltalk, Simula, Prolog++, CLOS, Ada-95

4.2.2. Características de los lenguajes orientados a objetos

Además de las características citadas anteriormente de objetos, clases y herencia, los LPOO deberán tener algunas o todas las características que se citan a continuación:

1. **Tipificación estricta (fuerte).** *Tipificación* es el proceso de declarar el tipo de información que puede contener una variable. Los errores de programación relacionados con el número de parámetros, tipos de parámetros e interfaces de módulos, se detectan durante las fases de diseño e implementación, en lugar de en tiempos de ejecución.
2. **Encapsulamiento.** Es deseable que el lenguaje soporte ocultación de la información, mediante partes independientes, para la especificación y la implementación. Esta característica proporciona un diseño débilmente acoplado que cumple con rigor el principio básico de la inferencia de software: *acoplamiento débil y fuerte cohesión* entre los módulos de un programa.
3. **Compilación incremental.** Característica en el desarrollo de sistemas grandes, en los que las porciones del sistema se crean e implementan de un modo sistemático (poco a poco, etapa a etapa). Esta característica complementa la característica de tipificación estricta, que soporta partes independientes de implementación y específica.
4. **Genericidad.** Las clases parametrizadas (mediante plantillas —*templates*— o unidades genéricas) sirven para soportar un alto grado de *reusabilidad* (reutilización). Estos elementos genéricos se diseñan con parámetros formales, que se instanciarán con parámetros reales, para crear instancias de módulos que se compilan y enlazan y ejecutan posteriormente.
5. **Paso de mensajes.** El lenguaje es conveniente soporte paso bidimensional de mensajes entre módulos, lo que implicará módulos débilmente acoplados y diseños flexibles. Esto significa que se deben poder pasar señales entre módulos, sin necesidad de tener que pasar realmente ningún dato.
6. **Polimorfismo.** Los lenguajes deben permitir que existan operaciones con igual nombre, que se utilicen para manejar objetos de tipos diferentes en tiempo de ejecución. El polimorfismo se implementa, normalmente, en unión con la herencia.
7. **Excepciones.** Se deben poder detectar, informar y manejar condiciones excepcionales utilizando construcciones del lenguaje. Esta propiedad añadida al soporte de tolerancia a fallos del software permitirá una estrategia de diseño eficiente.
8. **Concurrencia.** Es conveniente que el lenguaje soporte la creación de procesos paralelos independientes del sistema operativo. Esta propiedad simplificará la transportabilidad de un sistema de tiempo real de una plataforma a otra.
9. **Persistencia.** Los objetos deben poder ser persistentes; es decir, los objetos han de poder permanecer después de la ejecución del programa.
10. **Datos compartidos.** Los módulos se deben poder comunicar mediante memoria compartida, además del paso de mensajes.

Los lenguajes de programación disponibles actualmente no cumplen todas las características citadas anteriormente. En general, los lenguajes orientados a objetos no soportan concurrencia; este es el caso de C++, aunque las versiones de C++ que siguen el futuro estándar 4.0 comienzan a incorporar propiedades de concurrencia e incluso persistencia. Asimismo, Ada-95, además de concurrencia, soporta ya herencia y polimorfismo.

4.2.3. Puros frente a híbridos

Existe un profundo debate entre los usuarios y desarrolladores de OO sobre la decisión del lenguaje a emplear. Este debate no es reciente, aunque sí es en la actualidad cuando este debate se ha acrecentado y ha comenzado a ser decisivo en el desarrollo de la OO.

Un **LPOO** *puro* es un lenguaje diseñado para soportar *únicamente* el paradigma orientado a objetos, en el que todo consta de objetos, métodos y clases. Los **LPOO** más populares son Smalltalk y Eiffel. Un **LPOO** *híbrido*, por otra parte, soporta otros paradigmas de programación (tales como el tradicional —estructurado—, funcional, etc.), además del paradigma orientado a objetos. Los lenguajes híbridos se construyen a partir de otros lenguajes existentes, tales como C o Pascal, de los cuales se derivan; es posible utilizar el LPOO de un modo no orientado a objetos y también como orientado a objetos utilizando objetos, clases y métodos.

Cada tipo de lenguaje tiene sus ventajas e inconvenientes. Los lenguajes puros, tales como Smalltalk o Eiffel, pueden ser más potentes, ya que utilizan todas las ventajas de la tecnología. Proporcionan su máxima flexibilidad para cambiar los aspectos esenciales del lenguaje. Dado que todo en el lenguaje se

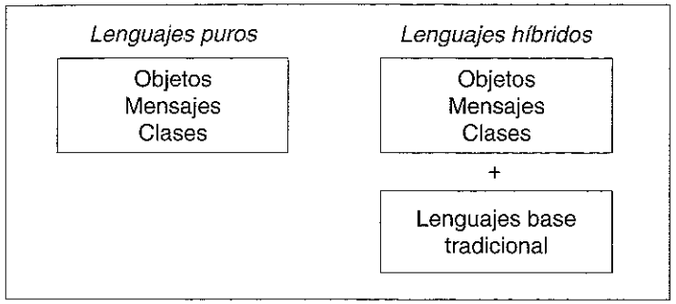


Figura 4.3. Lenguajes OO puros/híbridos.

construye sobre la base de los objetos, se pueden realizar cambios profundos si son necesarios.

Existe un precio, sin embargo, a la potencia y flexibilidad de los lenguajes puros. La mayoría de los lenguajes puros no son tan rápidos como los híbridos y son difíciles de codificar en toda clase de operaciones fundamentales. Esto les hace perder eficiencia en tiempo de ejecución.

En contraste, los lenguajes híbridos pierden con frecuencia alguna de las características de los lenguajes puros y normalmente no permiten modificar características de construcción del lenguaje base; aunque, como ya se ha comentado, son normalmente más rápidos para operaciones construidas en el lenguaje base.

Los lenguajes puros e híbridos difieren también en la facilidad de aprendizaje, si bien esta facilidad dependerá del nivel de cada persona. Así, para un programador la enseñanza de un lenguaje puro normalmente es más fácil, ya que es más sencillo: sólo ha de aprenderse un lenguaje y no dos, como en el caso de un lenguaje híbrido. Sin embargo, para un programador experimentado, el movimiento o emigración hacia un lenguaje híbrido puede ser más fácil si está *ya* familiarizado con el lenguaje base, pues entonces sólo necesita dominar las extensiones orientadas a objetos.

En la actualidad, C++ es el lenguaje más popular y utilizado; sin embargo, Smalltalk está ganando día a día en aceptación, debido esencialmente a estar soportado por las plataformas DOS, UNIX y Windows. El hecho de que en 1994 IBM lanzase versiones de Smalltalk para sus sistemas OS/2 y AIX hará crecer el número de usuarios de este lenguaje.

4.2.4. Tipificación estática frente a dinámica

Tipificación o *tipado* («typing») es el proceso de declarar cuál es el tipo de información que puede contener una variable. Por ejemplo, una variable se puede tipificar para contener un único carácter, una cadena de caracteres, un número entero o un número de coma flotante. Una vez que se ha declarado el tipo (*tipificado*), la variable está restringida a contener esa clase de datos. Cualquier intento de situar otra clase de datos en la variable producirá un mensaje de error.

Algunos lenguajes de objetos requieren que a todas las variables se les asigne un tipo antes de que se pueda utilizar. Otros lenguajes no requieren esos requisitos, permitiendo que las variables puedan tomar sus tipos adecuados en cualquier instante dado. Estas dos opciones se denominan *tipificación fuerte, estricta o estática,* y *tipificación débil, no estricta o dinámica*.

La tipificación fuerte o sistemas de tipos estáticos exige que el programador asocie explícitamente un tipo con cada nombre declarado en un programa, de modo que el compilador del lenguaje puede verificar que los nombres y expresiones compuestas de estos nombres se refieren siempre a los objetos del tipo especificado. El tipo de cada objeto se ha de determinar y comprobar antes que se ejecute el programa. Lenguajes con sistemas de tipos estáticos son los tradicionales FORTRAN, COBOL y PASCAL. La tipificación fuerte es menos flexible pero más segura, ya que el lenguaje puede realizar comprobaciones de rutinas para asegurar que los parámetros de los mensajes sean de tipo correcto.

Las ventajas considerables de un lenguaje tipificado estáticamente son que los errores relativos a tipos se *capturan* (detectan) durante la compilación, antes de que se ejecute el programa, y el programa se puede ejecutar más eficientemente, dado que no hay necesidad de hacer ninguna verificación de tipos en tiempo de ejecución.

Los lenguajes con un sistema de tipos dinámicos (lenguajes de tipificación débil) no exigen que el programador haya de especificar el tipo de objeto que puedan contener las variables cuando se escribe el programa. Cada objeto conoce su propio tipo cuando se crea durante la ejecución. Las ventajas de los lenguajes con un sistema de tipos dinámicos son que los programas son más flexibles y puede disponer de nuevos tipos de objetos que no fueron previstos cuando se escribió el programa. Esta flexibilidad se hace a costa de una pérdida de eficiencia durante la ejecución del programa, debido a la necesidad de mantener y comprobar el tipo de todos los objetos durante la ejecución.

Los lenguajes de objetos puros normalmente utilizan tipificación débil, ya que este enfoque les proporciona máxima flexibilidad, especialmente durante el desarrollo, cuando se crean nuevos tipos objetos. Los lenguajes híbridos, en contraste, tienden a construirse como lenguajes fuertemente tipificados. Es posible especificar el tipo de un argumento, de modo que haya flexibilidad en tiempo de ejecución, pero esta tarea requiere esfuerzos especiales.

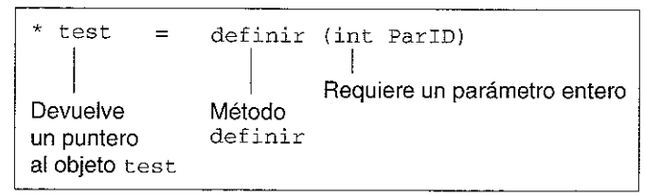


Figura 4.4. Tipificación de un mensaje.

4.2.5. Ligadura estática frente a dinámica

Ligadura es el proceso de asociar un atributo a un nombre. En el caso de funciones, el término ligadura (*binding*) se refiere a la conexión o enlace entre una llamada a la función y el código real ejecutado como resultado de la llamada

La ligadura se clasifica en dos categorías: **ligadura estática** y **ligadura dinámica**. La ligadura estática se conoce también como *ligadura temprana* o *anticipada* y se realiza en tiempo de compilación. La ligadura dinámica se conoce también como *ligadura retardada* o *postergada* y se realiza en tiempo de ejecución

Normalmente, los lenguajes funcionales y orientados a objetos presentan ligadura dinámica, al contrario que los lenguajes imperativos, que suelen presentar ligadura estática. Desde el punto de vista de la estructura del lenguaje, los intérpretes por definición realizan todos ligadura dinámica, mientras que los compiladores realizan normalmente ligadura estática.

Como ejemplo de ligadura estática, considere las declaraciones:

```
const n = 3;
var x : integer;
```

el valor 3 se enlaza estáticamente al nombre `n`, mientras que el tipo de dato `integer` se enlaza al nombre `x`. Por otra parte, consideremos la sentencia de asignación `x := 3;` en este caso, se enlaza 3 dinámicamente a `x` cuando se ejecuta la sentencia de asignación. Y la sentencia

```
new(y);
```

enlaza dinámicamente una posición de almacenamiento a `y^` y asigna a esa posición el valor de `y`.

Ejemplo de ligadura estática

Se desea procesar una determinada función, según sea un determinado carácter (código). Mediante una sentencia `switch` se podría realizar el programa correspondiente en C++:

```
#include <ctype.h>
#include <iostream.h>
//  .
static void salir()  func1(), func2();
char car;
//;
//car, contiene el código de carácter
   int codigo = toupper(car);
   switch(codigo)
   {
       case 'S' :  salir();
```



```
case 'P' :  func1();
            break;
case '*' :  func2();
            break;
default : cout <<  Código desconocido:' << car << endl;
}
```

En este caso, la función invocada en respuesta a cada orden se conoce cuando el programa se está compilando, ya que cada función se llama explícitamente por el nombre.

En el caso de propiedades orientadas a objetos, la ligadura se considera como el proceso mediante el cual se determina el receptor de un mensaje. En lenguajes que requieren ligadura estática, la identidad del receptor se debe especificar cuando se crea el programa. Si un lenguaje soporta ligadura dinámica, la identidad del receptor se puede dejar indeterminada hasta que se envía realmente el mensaje en tiempo de ejecución

La ligadura dinámica permite la implementación de una de las características más sobresalientes de la orientación a objetos: el **polimorfismo**.

4.2.6. Revisión de lenguajes orientados a objetos

Los dos lenguajes más utilizados en el desarrollo OO y más introducidos en el mercado son Eiffel, Smalltalk (en sus versiones 80 y V), C++, Actor, Objective-C, CLOS, Object Pascal (especialmente Turbo Pascal) y Visual BASIC.

El lenguaje Ada se suele considerar por muchos autores como *basado en objetos*, debido esencialmente a que soporta el concepto de abstracción de datos mediante el paquete y la genericidad mediante unidades genéricas; también es considerado basado en objetos, debido a que Grady Booch, autor del famoso método de diseño Booch, se apoyó en dicho lenguaje en su primera versión del método; la metodología HOOD también se apoya en Ada como lenguaje fundamental. La nueva versión Ada-95 ya es totalmente orientada a objetos

El lenguaje Eiffel se examina brevemente, debido a que fue diseñado para soportar totalmente la orientación a objetos y otras características de ingeniería de software. El uso de precondiciones, poscondiciones e invariantes mejoran significativamente la robustez y documentación de programas desarrollados en Eiffel.

Smalltalk, como heredero directo de Simula, es el prototipo de lenguaje de programación orientado a objetos *puro*.

Además de estos lenguajes, realizaremos una síntesis de los lenguajes Objective-C, Object Pascal y Visual BASIC. Por la importancia del lenguaje C++ y dado que es el lenguaje utilizado en este libro para la implementación de los conceptos orientados a objetos, se dedicarán capítulos específicos para enseñar la escritura y sintaxis del lenguaje, así como reglas y consejos para mejorar el estilo de programación y reglas y consejos para la depuración de programas.

4.3. ADA

Ada fue un lenguaje desarrollado a petición de DOD (Departamento de Defensa de Estados Unidos) y como fruto de un concurso para diseñar un lenguaje de programación que sirviera para reducir el coste del desarrollo del software.

Ada soporta conceptos orientados a objetos, tales como tipos abstractos de datos, sobrecarga de funciones y operadores, polimorfismo paramétrico (*genericidad*) e incluso especialización de tipos definidos por el usuario. El único concepto que no implementa Ada es la herencia, aunque es posible emular un tipo de herencia elemental. Las propiedades orientadas a objetos se implementan de la forma siguiente:

- **Encapsulamiento**, mediante el uso de paquetes
- **Ocultación de la información**, mediante tipos de datos privados y privados limitados. Instancias de estos tipos de datos sólo se pueden manipular por los subprogramas especificados en el paquete que define los tipos de datos privados; un tipo privado de Ada es un tipo abstracto de datos.
- **Sobrecarga** de operadores, funciones y procedimientos.
- **Genericidad,** mediante paquetes y subprogramas genéricos
- **Herencia,** aunque Ada no soporta la propiedad de la herencia, es posible emular la pseudoherencia mediante la definición de tipos de datos derivados, aunque no es posible tener propiedades adicionales de datos o modificación de las propiedades de datos existentes

4.3.1. Abstracción de datos en Ada

En Ada los tipos abstractos de datos se implementan mediante **paquetes**; si bien los tipos abstractos de datos sirven además para una gran variedad de aplicaciones.

Un paquete Ada consta de dos partes:

- **Especificación del paquete,** que declara los nombres de los tipos, subprogramas y objetos que son visibles a los usuarios del paquete.
- **Cuerpo del paquete,** que contiene la implementación de los componentes que son ocultos a los clientes del paquete.

Los formatos para definir la especificación y el cuerpo de un paquete son:

```
package <nombre_paquete> is                    especificación del paquete
      <elementos_declarativos>
   [private
      <elementos_declarativos>]
end <nombre_paquete> ;

package body <nombre_paquete> is               cuerpo del paquete
...
end <nombre_paquete> ;
```


Un tipo abstracto de dato para representar números complejos con diferentes tipos de operaciónes en Ada es:

```
package Tipo_Complejo is
   type complejo is private
   function "+" (izda, dcha:complejo) return complejo;
   function "-" (izda, dcha:complejo) return complejo;
   function "*" (izda, dcha:complejo) return complejo;
   function "/" (izda, dcha:complejo) return complejo;
   function "-' (dcha:complejo) return complejo;
private
   type complejo is record
       re:float := 0.0;
       im:float := 0.0;
       end record;
end Tipo_Complejo;
```

Los tipos privados permiten que se definan nuevos tipos de datos, mientras se oculta su implementación real. Al igual que otros tipos, los tipos privados se declaran dentro de una especificación de paquete. Un tipo privado se declara en la parte visible de un paquete, pero la definición real de su estructura interna se especifica en la parte privada de la especificación del paquete. La estructura del tipo definido en la parte privada es accesible sólo dentro del cuerpo del paquete.

Las operaciones permitidas en tipos privados son asignación, pruebas para igualdad y desigualdad, y cualquier operación definida explícitamente dentro de la especificación del paquete.

Ada proporciona también una definición de un tipo privado más limitado, denominado *privado limitado*. La diferencia principal es que el tipo privado limitado no hereda automáticamente operadores para asignación y pruebas de igualdad y desigualdad. Una instancia de un tipo privado limitado sólo se puede copiar en, o compararlo con, otra instancia si las operaciones necesarias han sido especificadas e implementadas por el paquete que declara el tipo de dato privado limitado. Ada no permite que se sobrecargue el operador de asignación, de modo que una implementación de usuario de esta operación para un tipo de dato privado limitado se ha de especificar como un procedimiento con nombre.

4.3.2. Genericidad en Ada

Ada soporta genericidad. Las unidades genéricas son subprogramas o paquetes parametrizados que permiten al usuario desarrollar código reutilizable. El formato de una unidad genérica es:

```
generic
  <declaración_genérica_de_parámetro>
  {<especificación_subprograma> |
   <especificación_paquete> }
```

La instanciación de una unidad genérica crea una copia de la unidad genérica. Los parámetros genéricos se corresponden con los parámetros reales durante la instanciación. Se especifican tipos reales para todos los tipos formales genéricos. Una variable o una constante se especifica para todos los objetos genéricos formales. Un nombre de función o subprograma se da a cualquier subprograma genérico. Así, por ejemplo, para instanciar un subprograma genérico `Intercambio`, que intercambia entre sí el contenido de unas variables:

```
procedure Intercambio_INTEGER is new Intercambio(INTEGER);
```

También se puede instanciar un procedimiento llamado `Intercambio_Empleado`, que intercambia dos registros `empleado`:

```
procedure Intercambio_Empleado is new Intercambio(Empleado);
```

4.3.3. Soporte de herencia en Ada-83

Ada-83 no soporta herencia, pero sí un tipo de pseudo-herencia, de modo que una declaración de tipo de dato puede especificar que el tipo de dato se deriva de un tipo de dato ya existente. Así, a partir del tipo `complejo`, definido anteriormente, se puede declarar otro tipo `Tipo_Complejo`, del modo siguiente:

```
with Tipo_Complejo;
   package Ejemplo is
   type otro_tipo_complejo is new Tipo_Complejo.Complejo;

end Ejemplo;
```

Un tipo derivado Ada hereda todas las operaciones declaradas para el tipo base y éstas se pueden redefinir para el nuevo tipo y especificadas operaciones adicionales. Sin embargo, si el tipo base es privado, el único modo de implementar cualquier operación adicional o redefinida es a través del uso de operaciones del tipo base; no existe en Ada el concepto de acceso protegido de C++.

No es posible especificar nuevas propiedades de datos para un tipo derivado en Ada, y en consecuencia, todos los tipos derivados de un tipo base tienen la misma representación.

El soporte de herencia de Ada no se puede utilizar para capturar abstracciones generalizadas, como un tipo de dato base abstracto (*clase abstracta en C++*), de las que se puedan derivar tipos de datos abstractos. *Ada tampoco soporta ninguna forma de ligadura dinámica ni de polimorfismo.*

4.3.4. Soporte Ada para orientación a objetos

Las grandes limitaciones de método de herencia de Ada y su ausencia de soporte de polimorfismo no permiten que Ada pueda ser utilizado como un lenguaje de programación orientado a objetos. A lo máximo, Ada se puede considerar

como un *lenguaje basado en objetos*, en base a que soporta todo el concepto de tipo abstracto de datos. La Figura 4.5 muestra las características de OO de Ada, así como sus carencias.

Ada-83 *soporta*	Genericidad
	Sobrecarga (no totalmente)
	Verificación estática de tipos
	Encapsulamiento
	Ocultación de la información
Ada-83 *no soporta*	Herencia
	Polimorfismo
	Extensibilidad
Ada-95 *soporta*	Herencia
	Polimorfismo
	Extensibilidad

Figura 4.5.

Un método acreditado de diseño basado en Ada se denomina ***Hierarchical Object Oriented Design*** (HOOD), por lo que puede inducir a confusión entre descomposición jerárquica de objetos, única posibilidad con Ada, y las verdaderas jerarquías de clases que soportan abstracciones mediante generalizaciones y especializaciones.

El proyecto Ada 9x, cuyo borrador definitivo se conoce como Ada-95, ha definido un nuevo estándar de Ada publicado en 1995, tanto por ISO como por ANSI, que proporciona soporte completo orientado a objetos, aunque con ciertas restricciones todavía sobre C++, pero también con notables ventajas, como es el caso de la concurrencia.

4.4. EIFFEL

Eiffel[3] es un lenguaje orientado a objetos puro desarrollado por Bertran Meyer de Interactive Software Environments Inc. Su diseño se inspiró en Simula, pero también muestra la influencia de Smalltalk y Ada. Además incorpora modernas técnicas de ingeniería de software, que lo hacen idóneo para construcción de software para grandes aplicaciones, permitiendo desarrollar aplicaciones robustas, exactas, transportables y eficientes.

Eiffel es un lenguaje compilado tipificado fuertemente, que soporta la mayoría de los conceptos orientados a objetos descritos en los capítulos anteriores, tales como abstracción de datos mediante clases, genericidad, herencia (simple y múltiple), ligadura dinámica y polimorfismo. También proporciona una amplia biblioteca de clases y soporta recolección de basura (*garbage collection*). El entorno Eiffel está especialmente concebido para áreas de ingeniería de software, tales como bases de datos o inteligencia artificial, y tiene un soporte muy limitado para objetos persistentes y concurrencia, aunque ambas carac-

[3] MEYER, Bertrand: *Object-Oriented Software Construction*. Prentice-Hall, 1988.

terísticas han sido prometidas por sus diseñadores [Meyer, 1988] y en trabajos posteriores. Las características de Eiffel se resumen en la Tabla 4.1.

Tabla 4.1. Características de Eiffel.

Ocultación de la información	Sí.
Herencia	Sí (simple y múltiple).
Verificación/ligadura de tipos	Temprana.
Polimorfismo	Sí.
Recolección de basura	Sí.
Persistencia	Pseudo-persistencia.
Concurrencia	Prometida.
Genericidad	Sí.
Biblioteca de objetos	Pocas.

Eiffel soporta conceptos de ingeniería de software tales como precondiciones, poscondiciones e invariantes de clases. Estas características y las propias de orientación a objetos lo hacen muy útil para construir, soportar y mantener proyectos de software grandes.

4.4.1. La biblioteca de clases Eiffel

El entorno de programación Eiffel proporciona una biblioteca de clases predefinidas muy grande. Las clases predefinidas van desde clases de estructura de datos, apoyo y de núcleo (*Kernel*), hasta clases gráficas que soportan puntos, rectángulos, etc.

La biblioteca de clases de gráficos avanzados soporta el sistema XWindow, lo que permite al programador construir interfaces de usuario sofisticados.

4.4.2. El entorno de programación Eiffel

El compilador Eiffel, un hojeador de clases (*browser*), un editor y las herramientas «*flat*» y «*short*» son las partes fundamentales del entorno de programación Eiffel.

El compilador Eiffel tiene una característica de recompilación automática. Recompila sólo aquellas clases que estén afectadas por un cambio. El hojeador de clases permite el examen de clases en el contexto de sus jerarquías (superclases y subclases), pudiendo visualizar las relaciones existentes entre clases.

La gestión de memoria en Eiffel se realiza por el entorno de programación. El espacio de objetos se asigna cuando se crean objetos. Eiffel proporciona recolección automática de basura. Se comprueba cuando un objeto se vuelve obsoleto y libera su espacio asociado. Eiffel soporta también un mecanismo de manejo de excepciones.

4.4.3. El lenguaje Eiffel

El único criterio de estructuración en Eiffel es la clase. Una *clase* es una unidad simple y no se separa en una parte interfaz y en una parte implementación. Una clase en Eiffel define los elementos básicos de sus objetos instancia: variables de instancia, métodos de instancia y un método de clase. Las variables de instancias se llaman también *atributos*, los métodos de instancia se llaman también *rutinas* y los atributos y rutinas se llaman colectivamente *características*.

La clase `Persona` se representa con un código similar a:

```
class Persona
creation
   crea
feature
   nombre:STRWG;
   conyuge: Persona;
feature {PERSONA} casado (n:PERSONA) is
do
   conyuge :=n
end;
feature
crea (s:STRING) is do
   nombre :=s;
   end
   casado_con (p: PERSONA) is
   require
      existe : p/=void;
      solteros: (conyuge=void and y.conyuge=void)
   do
      casado(p);
      p.casado (actual)
   ensure
      boda_valida:conyuge=y and actual=p.conyuge
   end
end
```

La clase `Estudiante` que se deriva de `Persona` podrá ser:

```
class Estudiante
   inherit
      Persona
         rename imprimir as imprimirpersona
         redefine imprimir
      end;
   creation
      hacer
   feature {NONE}
      es:STDFILES;
   feature
      hacer x (n=STRING) is
   do
```

```
        nombrepersona := n;
    end
    imprimir is
    do
        es.putstring ("Estudiante: );
        imprimirpersona;
    end;
end
```

Una de las características más sobresalientes de Eiffel es soportar el concepto de *contrato* entre el proveedor (servidor) de una clase y el usuario de una clase. El proveedor o implementador de una clase es capaz de especificar, como parte de la definición de la clase en Eiffel, bajo qué condiciones los objetos de la clase se comportan correctamente. En Eiffel, estas condiciones se expresan, de la siguiente forma:

- *Precondiciones*, que se deben cumplir *antes* de que se invoque un método
- *Poscondiciones*, que se garantizan han de cumplirse *después* de que un método se ha ejecutado
- *Invariantes*, que siempre son verdaderas para todas las ruptancias de una clase

El contrato entre el proveedor y el usuario de una clase espera que:

- El *usuario* de una clase asegurará que las precondiciones de un método se cumplan antes de que el método se invoque.
- El *proveedor* de la clase garantizará que las poscondiciones se cumplan después que se ha aplicado el método

4.5. SMALLTALK

Smalltalk es el primer lenguaje creado con tecnología de objetos puros. Fue desarrollado al principio de los setenta por Alan Kay y Adele Golberdg en el Software Concepts Group de Xerox Palo Alto Research Center. En la actualidad Smalltalk comienza a imponerse como uno de los primeros lenguajes de programación orientados a objetos.

La primera versión comercial de Smalltalk se lanzó en 1983 con el nombre de Smalltalk-80. En un principio sólo estuvo disponible para potentes estaciones de trabajo gráficas. Smalltalk, en el sentido más estricto, es un entorno de programación interactivo y requiere gran cantidad de memoria. En la actualidad las versiones de Smalltalk corren bajo entornos Windows, OS/2 y UNIX.

En esencia, el entorno de programación Smalltalk tiene tres componentes: *el lenguaje básico Smalltalk, una colección de clases*, que se utilizan para implementar el sistema Smalltalk completo, y *el entorno real de programación*, que permite a un programador introducir, comprobar y ejecutar aplicaciones Smalltalk.

4.5.1. El lenguaje Smalltalk

El bloque fundamental básico de un programa Smalltalk es la clase. La clase contiene la descripción de variables de instancia y métodos. El *método* es el equivalente al término *función miembro de* ++. Al igual que en C++, los métodos contienen el código que definen cómo responde un objeto a un mensaje. La forma básica de una sentencia Smalltalk es un «mensaje enviar»:

receptor mensaje : argumento

que expresa cómo un mensaje se envía a un receptor con un argumento.

Smalltalk es un lenguaje totalmente dinámico. Los métodos se buscan cuando un objeto recibe un mensaje. La ligadura estática no es una opción en Smalltalk. La Tabla 4.2 relaciona los términos equivalentes entre C++ y Smalltalk.

Tabla 4.2. C++ frente a Smalltalk.

Smalltalk	C++
Clase	Clase
Superclase	Clase base
Subclase	Clase derivada
Variable de clase	Campo miembro estático
Variable de instancia	Campo miembro
Método de una clase	Función miembro estática
Método instancia	Función miembro

La Tabla 4.3 contiene las características más sobresalientes de Smalltalk.

Tabla 4.3. Características de Smalltalk.

Polimorfismo	*Sí*
Ligadura/Verificación de tipos	*Tardía*
Ocultación de la información	*Sí*
Concurrencia	*Pobre*
Herencia simple	*Sí*
Herencia múltiple	*No*
Recolección de basura	*Sí*
Genericidad	*No*
Persistencia	*No*

Smalltalk no es un lenguaje híbrido, como C++, sino un lenguaje puro, que no permite un estilo de programación convencional. Es el lenguaje de programación más dinámico de los conocidos y dispone de un entorno de programación excelente, aunque tiene el inconveniente de tener una velocidad de ejecución pequeña. Otro inconveniente de Smalltalk es ser un sistema de un solo usuario.

Por el contrario, el desarrollo de aplicaciones en Smalltalk tiende a ser muy rápido. Además, dado que el código de Smalltalk se compila incrementalmente, las nuevas definiciones de clases y objetos se vuelven eficaces tan pronto como se introducen. El alto grado de interactividad del entorno Smalltalk lo hace altamente productivo para desarrollos rápidos.

4.5.2. La jerarquía de clases Smalltalk

El entorno de programación incluye una amplia biblioteca de varios centenares de clases. Para gestionar esta gran lista de clases, Smalltalk las agrupa en categorías. Las categorías de clases implementan muchas estructuras de datos básicos: colecciones, conjuntos, bolsas, diccionarios, etc. Además, existen clases predefinidas que realizan entradas y salidas y ayudan a desarrollar componentes de interfaces de usuarios; algunas clases de esta categoría son: `Punto`, `Rectángulo`, `Forma`, `Vista`

La Figura 4.6 muestra una parte de la jerarquía de clases de Smalltalk.

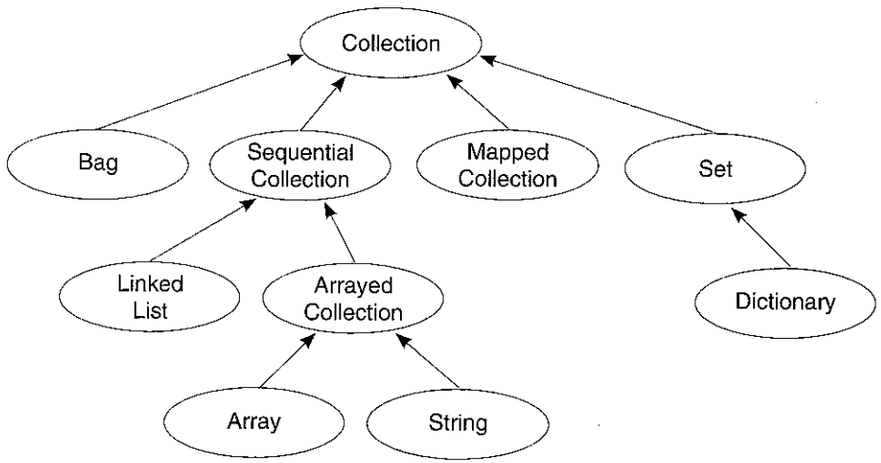


Figura 4.6. Jerarquía *Taylor* de clases de Smalltalk.

4.6. OTROS LENGUAJES DE PROGRAMACION ORIENTADOS A OBJETOS

Hoy día se puede aventurar que existen muy pocos lenguajes comercializados que no tengan versiones orientadas a objetos, tal vez sea FORTRAN el único lenguaje que no soporta estas características

La mayoría de los lenguajes estructurados modernos presentan una versión que es híbrida. Los casos más sobresalientes son:

Turbo Pascal (Object Pascal)	*versión híbrida de Pascal*
Visual Object	*versión híbrida de Clipper*
Delphi	*versión híbrida de Turbo Pascal*
Object COBOL	*versión híbrida de COBOL*
Visual BASIC 4 (*no incorpora propiedades de herencia ni de polimorfismo*)	
C++	*versión híbrida de C*

El lenguaje C++ se estudia y analiza en profundidad en la segunda y tercera parte de este libro, y es la base fundamental de todo el desarrollo del mismo. Los Apéndices A, B, C, D y E muestran unas guías rápidas de los lenguajes C++, Turbo/Borland Pascal 7, Delphi, Ada-95 y Java.

RESUMEN

El auge de las tecnologías de orientación a objetos se ha debido fundamentalmente a la existencia y posterior popularidad de numerosos lenguajes de programación orientados a objetos

En la década de los sesenta, los diseñadores del lenguaje Simula introdujeron el concepto de *objeto*. Conceptualmente, un objeto contenía tantos datos como operaciones que manipulaban esos datos. Simula incorporó también la noción de *clases* que se utilizaban para describir la estructura y comportamiento de un conjunto de objetos. Otra característica importante soportada por Simula fue la herencia de clases.

Durante la década de los setenta y principio de los ochenta, los conceptos de orientación a objetos de Simula se pasaron a Smalltalk, uno de los lenguajes orientados a objetos más influyentes en el desarrollo de las tecnologías de objetos. Este lenguaje incorporaba muchas de las características orientadas a objetos de Simula, incluyendo clases y herencia. Pero Smalltalk añadió una característica muy notable: la incorporación de un entorno de programación completo y un interfaz de usuario interactivo basado en menús.

Smalltalk es muy rico en conceptos orientados a objetos. En Smalltalk todo es un objeto, incluyendo clases y tipos base.

La taxonomía de lenguajes de Wegner clasificaba éstos en basados en objetos, basados en clases y orientados a objetos

Los lenguajes basados en objetos más populares son Ada, Modula-2, Clipper 5-2. Los lenguajes basados en clases se agrupan en torno a Cla. Los lenguajes orientados a objetos se clasifican en dos grandes bloques: *puros* e *híbridos*.

Lenguajes orientados a objetos puros son Simula, Smalltalk y Eiffel, y lenguajes orientados a objetos híbridos son Object Pascal (Turbo Pascal y Borland Pascal), Objective-C, Object COBOL, Overon, C++, Delphi, etc.

El lenguaje OO híbrido por excelencia es C++, aunque en los últimos años están apareciendo otros lenguajes híbridos que comienzan a competir con cierto éxito, tales como Object COBOL, Visual Object, Visual BASIC 4, Delphi, etcétera.

En los lenguajes OO puros todo es un objeto, incluso las definiciones de tipos, mientras que en los lenguajes híbridos no todo necesita ser un objeto

Las grandes ventajas de los lenguajes híbridos son:

- Período de aprendizaje (los programadores ya experimentados en versiones no orientadas a objetos pueden comenzar con facilidad a utilizar las versiones OO).
- Se pueden utilizar las enormes cantidades de código puente existente (código heredado) C++ se diseñó de modo que pudiera compilar código estándar C con pocas o ninguna modificación; de igual forma, Delphi se ha diseñado para compilar programas anteriores de Turbo/Borland Pascal con cambios mínimos

EJERCICIOS

4.1. Describa y justifique los objetos que obtiene de cada uno de estos casos:

a) Los habitantes de Europa y sus direcciones de correo.
b) Los clientes de un banco que tienen una caja fuerte alquilada.
c) Las direcciones de correo electrónico de una universidad.
d) Los empleados de una empresa y sus claves de acceso a sistemas de seguridad

4.2. ¿Cuáles serían los objetos que han de considerarse en los siguientes sistemas?

a) Un programa para maquetar una revista.
b) Un contestador telefónico.
c) Un sistema de control de ascensores.
d) Un sistema de suscripción a una revista.

4.3. Deducir los objetos necesarios para diseñar un programa de computadora que permita jugar a diferentes juegos de cartas.

4.4. Dibujar un diagrama de objetos que represente la estructura de un coche (carro). Indicar las posibles relaciones de asociación, generalización y agregación.

4.5. Dibujar diagramas de objetos que representen la jerarquía de objetos del modelo `Figura`.

CAPITULO **5**

MODELADO DE OBJETOS: RELACIONES

CONTENIDO

La orientación a objetos (POO) intenta modelar aplicaciones del mundo real tan fielmente como sea posible, apoyándose para ello en una gran facilidad de reutilización y extensibilidad del software. El potente concepto OO que proporciona todas estas características es la **herencia**.

A través de la herencia, los diseñadores pueden construir módulos de software (tales como clases). Las nuevas clases pueden heredar tanto el comportamiento (*behaviour*, operaciones, métodos, etc.) como la representación (variables de instancia, atributos, etc.) a partir de las clases existentes.

La herencia es la característica que diferencia esencialmente la programación orientada a objetos de la programación tradicional, debido fundamentalmente a que permite extender y reutilizar el código existente sin tener que rescribir el código.

En este capítulo analizaremos las relaciones fundamentales entre clases: el concepto de herencia y su modo de implementación en un lenguaje orientado a objetos, así como las otras relaciones entre clases, *agregación* y *asociación*.

5.1. RELACIONES ENTRE CLASES

Las relaciones entre clases juegan un papel importante en el modelo de objetos. Las clases, al igual que los objetos, no existen de modo aislado. Por esta razón existirán relaciones entre clases y entre objetos.

Las relaciones entre entidades, clave del modelo relacional de datos, se expresan utilizando verbos a partir de frases del lenguaje ordinario, tales como *vive-en, estudia-en, trabaja-para*.

Las relaciones entre clases, como indica Booch[1], se deben a dos razones: Primera, una relación de clases puede indicar algún tipo de compartición (por ejemplo, margaritas y rosas son ambas tipos de flores). Segunda, una relación entre clases puede indicar algún tipo de conexión semántica; por ejemplo, las rosas rojas y amarillas son más parecidas entre sí que las margaritas y rosas.

Los tres grandes tipos de relaciones entre clases son:

- *Generalización/especialización* **(es-un)**.
- *Agregación* **(todo/parte)**.
- *Asociación*.

La primera de ellas es la *generalización*, que representa una relación «*un tipo de*». Por ejemplo, una rosa **es un** tipo de flor, significando que una rosa es una subclase especializada de la clase más general, flor. Este tipo de relación se conoce como relación **es-un** (*is-a*). La segunda es *agregación*, que representa «*una parte de*» la relación. Por consiguiente, un pétalo no es una clase de flor; es una parte de una flor. Esta relación también se conoce como relación **tiene** (*has*). La tercera relación es la *asociación*, que representa conexión semántica entre clases no relacionadas. Así, por ejemplo, rosas y velas son clases independientes, aunque ambas representan cosas que se pueden utilizar para decorar una mesa para cenar.

Las relaciones se expresan frecuentemente utilizando verbos o frases verbales del lenguaje natural, tales como *vive-en, estudia-en, es-responsable-de*.

5.2. RELACION DE GENERALIZACION/ESPECIALIZACION (*is-a/es-un*)

Booch[2], para mostrar las semejanzas y diferencias entre clases, utiliza las siguientes clases de objetos: flores, margaritas, rosas rojas, rosas amarillas y pétalos. Se puede constatar que:

- Una margarita *es un* tipo (una clase) de flor.
- Una rosa *es un* tipo (diferente) de flor.

[1] BOOCH, Grady: *Object-Oriented Analysis and Design with Applications* Benjamin/Cummings, 2ª edición, 1994.

[2] BOOCH, Grady: *Object-Oriented Design with Applications* Benjamin/Cummings, 1991, págs. 96-100.

- Las rosas rojas y amarillas son *tipos de* rosas.
- Un pétalo *es una parte* de ambos tipos de flores.

Como Booch afirma, las clases y objetos no pueden existir aislados, y en consecuencia existirán entre ellos relaciones. Las relaciones entre clases pueden indicar alguna forma de compartición, así como algún tipo de conexión semántica. Por ejemplo, las margaritas y las rosas son ambas tipos de flores, significando que ambas tienen pétalos coloreados brillantemente, ambas emiten fragancia, etc. La conexión semántica se materializa en el hecho de que las rosas rojas y las margaritas y las rosas están más estrechamente relacionadas entre sí que lo están los pétalos y las flores.

Las clases se pueden organizar en estructuras jerárquicas. La *herencia* es una relación entre clases donde una clase comparte la estructura o comportamiento, definida en una (*herencia simple*) o más clases (*herencia múltiple*). Se denomina *superclase* a la clase de la cual heredan otras clases. De modo similar, una clase que hereda de una o más clases se denomina *subclase*. Una subclase heredará atributos de una superclase más elevada en el árbol jerárquico. La herencia, por consiguiente, define un «tipo» de jerarquía entre clases, en las que una subclase hereda de una o más superclases.

La Figura 5.1 ilustra una jerarquía de clases `Animal`, con dos subclases que heredan de `Animal`, `Mamíferos` y `Reptiles`.

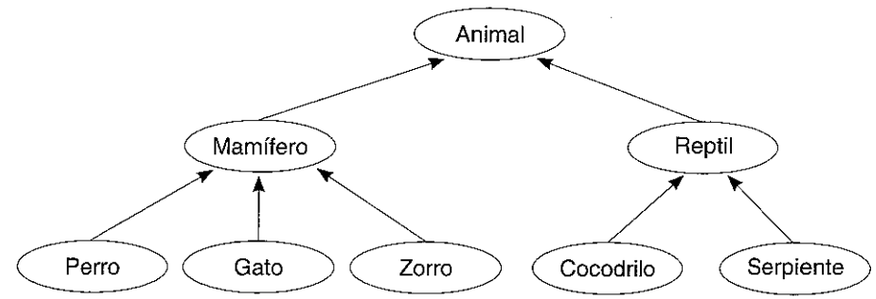


Figura 5.1. Jerarquía de clases.

Herencia es la propiedad por la cual instancias de una clase hija (o subclase) puede acceder tanto a datos como a comportamientos (métodos) asociados con una clase padre (o superclase). La herencia siempre es transitiva, de modo que una clase puede heredar características de superclases de nivel superior. Esto es, si la clase `perro` es una subclase de la clase `mamífero` y de `animal`.

Una vez que una jerarquía se ha establecido es fácil extenderla. Para describir un nuevo concepto no es necesario describir todos sus atributos. Basta describir sus diferencias a partir de un concepto de una jerarquía existente. La herencia significa que el comportamiento y los datos asociados con las clases hija son siempre una extensión (esto es, conjunto estrictamente más grande) de las propiedades asociadas con las clases padres. Una subclase debe tener todas las propiedades de la clase padre y otras. El proceso de definir nuevos tipos

y reutilizar código anteriormente desarrollado en las definiciones de la clase base se denomina *programación por herencia*. Las clases que heredan propiedades de una clase base pueden, a su vez, servir como clases base de otras clases. Esta jerarquía de tipos normalmente toma la estructura de árbol, conocido como *jerarquía de clases* o *jerarquía de tipos*.

La jerarquía de clases es un mecanismo muy eficiente, ya que se pueden utilizar definiciones de variables y métodos en más de una subclase sin duplicar sus definiciones. Por ejemplo, consideremos un sistema que representa varias clases de vehículos manejados por humanos. Este sistema contendrá una clase genérica de vehículos, con subclases para todos los tipos especializados. La clase `vehículo` contendrá los métodos y variables que fueran propios de todos los vehículos, es decir, número de matrícula, número de pasajeros, capacidad del depósito de combustible. La subclase, a su vez, contendrá métodos y variables adicionales que serán específicos a casos individuales.

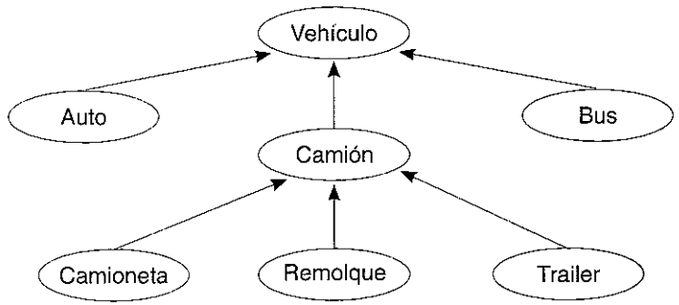


Figura 5.2. Subclases de la clase `Vehículo`.

La flexibilidad y eficiencia de la herencia no es gratuita; se emplea tiempo en buscar una jerarquía de clases para encontrar un método o variable, de modo que un programa orientado a objetos puede correr más lentamente que su correspondiente convencional. Sin embargo, los diseñadores de lenguajes han desarrollado técnicas para eliminar esta penalización en velocidad en la mayoría de los casos, permitiendo a las clases enlazar directamente con sus métodos y variables heredados de modo que no se requiera realmente ninguna búsqueda.

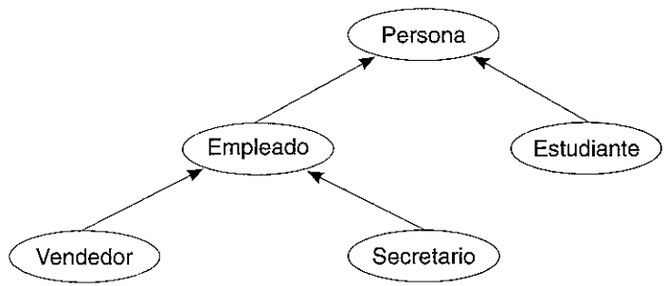


Figura 5.3. Una jerarquía `Persona`.

5.2.1. Jerarquías de generalización/especialización

Las clases con propiedades comunes se organizan en superclases.

Una **superclase** representa una *generalización* de las subclases. De igual modo, una subclase de una clase dada representa una *especialización* de la clase superior (Fig. 5.4). La clase derivada *es-un* tipo de clase de la clase base o superclase.

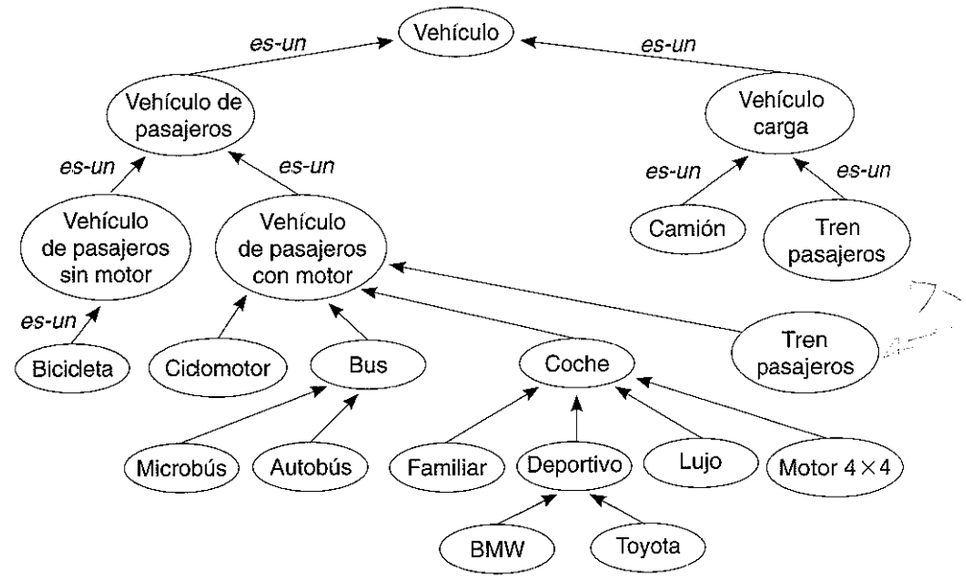


Figura 5.4. Relación de *generalización*.

Una superclase representa una *generalización* de las subclases. Una subclase de la clase dada representa una *especialización* de la clase ascendente (Figura 5.5).

En la *modelización* o *modelado* orientado a objetos es útil introducir clases en un cierto nivel que puede no existir en la realidad, pero que son construcciones conceptuales útiles. Estas clases se conocen como **clases abstractas** y su propiedad fundamental es que no se pueden crear instancias de ellas. Ejemplos de clases abstractas son `VEHICULO DE PASAJEROS` y `VEHICULO DE MERCANCIAS`. Por otra parte, de las subclases de estas clases abstractas, que corresponden a los objetos del mundo real, se pueden crear instancias directamente por sí mismas. Por ejemplo, de `BMW` se pueden obtener, por ejemplo, dos instancias, `Coche1` y `Coche2`.

La generalización, en esencia, es una abstracción en que un conjunto de objetos de propiedades similares se representa mediante un objeto genérico. El método usual para construir relaciones entre clases es definir generalizaciones buscando propiedades y funciones de un grupo de tipos de objetos similares, que se agrupan juntos para formar un nuevo tipo genérico. Consideremos el

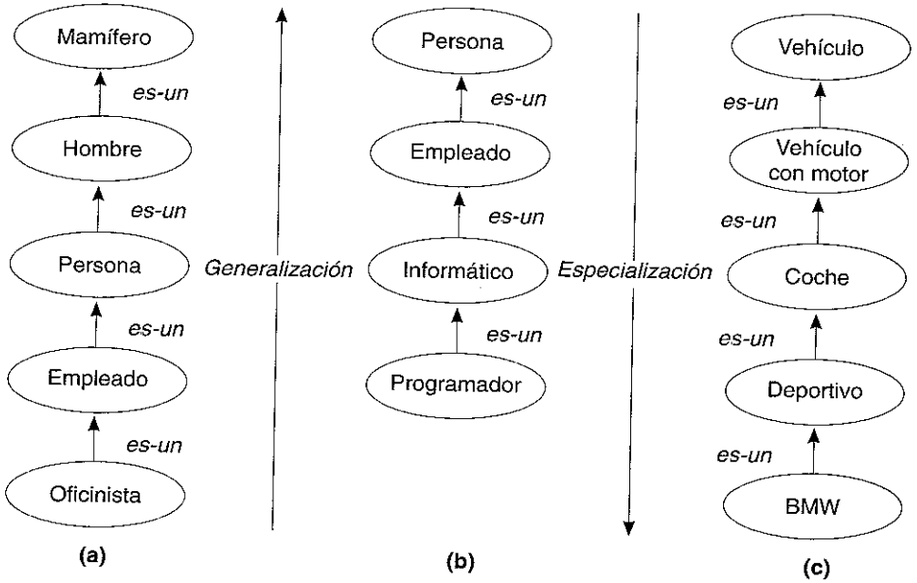


Figura 5.5. Relaciones de jerarquía *es-un (is-a)*: **(a)**, **(c)** generalización; **(b)** especialización.

caso de empleados de una compañía que pueden tener propiedades comunes (nombre, número de empleado, dirección, etc) y funciones comunes (calcular_nómina), aunque dichos empleados pueden ser muy diferentes en atención a su trabajo: oficinistas, gerentes, programadores, ingenieros, etc. En este caso, lo normal será crear un objeto genérico o superclase `empleado`, que definirá una clase de empleados individuales. Por ejemplo, `analistas`, `programadores` y `operadores` se pueden generalizar en la clase `informático`. Un programador determinado (`Mortimer`) será miembro de las clases `programador`, `informático` y `empleado`; sin embargo, los atributos significativos de este programador variarán de una clase a otra

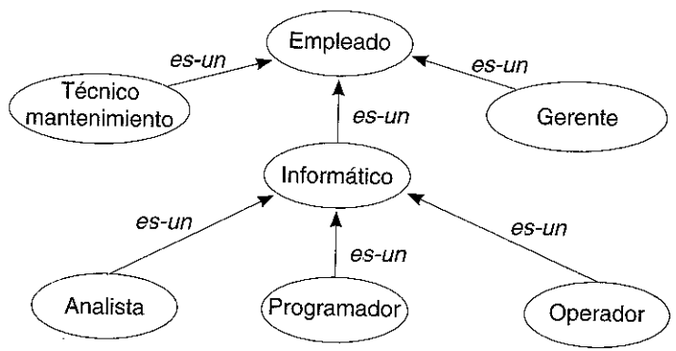


Figura 5.6. Una jerarquía de generalización de empleados.

La jerarquía de generalización/especialización tiene dos características fundamentales y notables. Primero, un tipo objeto no desciende más que de un tipo objeto genérico; segundo, los descendientes inmediatos de cualquier nodo no necesitan ser objetos de clases exclusivas mutuamente. Por ejemplo, los `gerentes` y los `informáticos` no tienen porqué ser exclusivos mutuamente, pero pueden ser tratados como dos objetos distintos; es el tipo de relación que se denomina *generalización múltiple*.

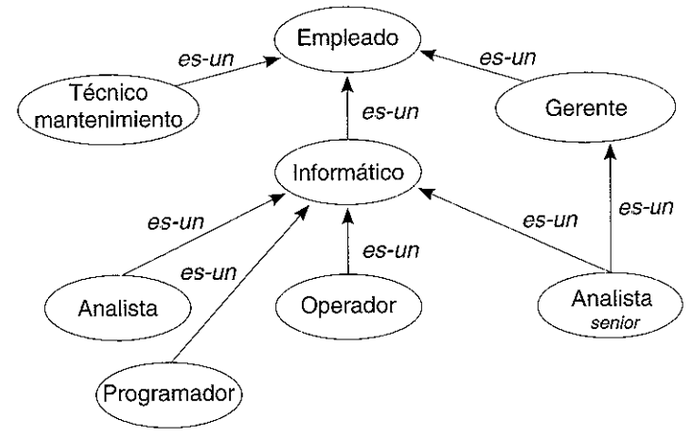


Figura 5.7. Una jerarquía de generalización múltiple.

5.3. RELACION DE AGREGACION *(has-a/tiene-un)*

Una *agregación* es una relación que representa a los objetos compuestos. Un objeto es *compuesto* si se compone a su vez de otros objetos. Una casa se compone de habitaciones, tejados, suelos, puertas, ventanas, etc. Una habitación, a su vez, se compone de paredes, techos, suelo, ventanas y puertas (Figura 5.8).

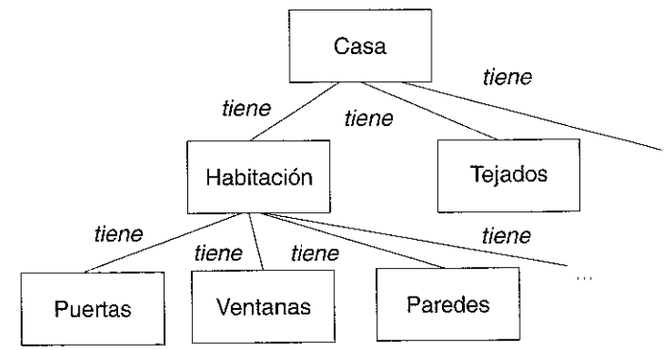


Figura 5.8. Un objeto compuesto (*casa*) y sus componentes.

La agregación de objetos permite describir modelos del mundo real que se componen de otros modelos, que a su vez se componen de otros modelos. La agregación es un concepto que se utiliza para expresar tipos de relaciones entre objetos **parte-de** (*part-of*) o **tiene-un** (*has-a*) El objeto componente, también a veces denominado *continente* o *contenedor*, es un objeto agregado que se compone de múltiples objetos.

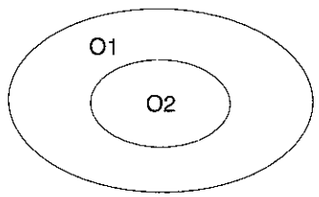
La agregación es una forma específica de asociación y no un comportamiento independiente, que añade significados o connotaciones semánticas en ciertos casos. Dos objetos forman un agregado, o existe entre ellos una relación de agregación, si existe entre ellos una relación **todo-parte, continente-contenido** (*whole-part*) Si dos objetos se consideran normalmente como independientes, sus relaciones se consideran normalmente una asociación Rumbaugh *et al*[3], en la obra ya citada, sugiere las siguientes pruebas para determinar si una relación es una agregación:

- ¿Se utiliza la frase *parte-de* (*tiene-un, consta-de* ..) para describir la relación?
- Las operaciones del todo, ¿se aplican automáticamente a sus partes?
- Los valores de los atributos, ¿se propagan del todo (completo) a todas o algunas partes?
- ¿Existe una asimetría intrínseca a la asociación en la que una clase de objetos se subordina a la otra?

Si se responde afirmativamente a cualquiera de estas preguntas, se tiene una agregación

La agregación puede ser de dos tipos: por *contenido físico*[4], o por *contenido por referencia* o *conceptual*

La *agregación por contenido físico* (Fig 5.9) o *por valor* implica que un objeto contenido no existe independientemente del objeto contenedor La vida de ambos objetos está íntimamente relacionada.



Cuando se crea un sintagma de la clase (un objeto) `O1`, se crea una instancia de la clase `O2` (otro objeto) Cuando se destruye un objeto del contenedor `O1`, se destruye otro objeto de `O2`.

El objeto agregado `COCHE`[5] se compone de un `MOTOR`, una `TRANSMISION`,

[3] En la obra *Object-Oriented Modeling and Design*, de JAMES RUMBAUGH *et al* se muestra uno de los mejores estudios existentes sobre modelos de objetos

[4] GRADY BOOCH define estos dos términos en su obra, ya citada, *Object-Oriented Analysis and Design* 2.ª edición

[5] En Latinoamérica se utiliza como acepción usual de *automóvil* la palabra **carro**

un `CHASIS`, etc., que son a su vez *parte-de* `COCHE`, representa un ejemplo de una agregación con contenido físico (Fig 5.9)

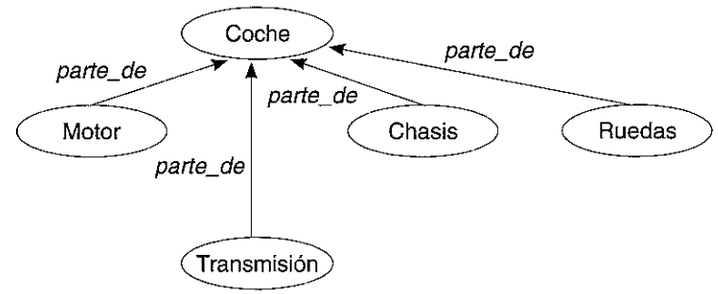


Figura 5.9. Agregación con contenido físico.

La *agregación no implica siempre contenido físico*, como en el caso de `COCHE`, sino que puede implicar simplemente relaciones conceptuales. Así, un `EDIFICIO` puede componerse de `DIVISIONES`, que a su vez constan de `OFICINAS`, o bien una `COMPAÑIA` consta de varios `DEPARTAMENTOS` y cada `DEPARTAMENTO` consta de varias `SECCIONES`.

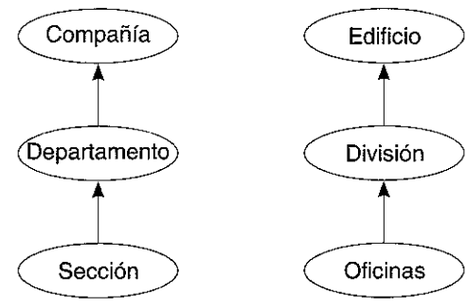


Figura 5.10. Agregación sin contenido físico.

En la *agregación por referencia, conceptual* o *sin contenido físico* existe fuerte dependencia entre objetos de las clases continente/contenido y no están acoplados entre ellos Eso significa que se pueden crear y destruir instancias de clase independientemente

5.3.1. Agregación frente a generalización

La agregación es diferente de la generalización La generalización relaciona las clases constituyendo un modo de estructurar la descripción de un objeto único Con la generalización, un objeto es simultáneamente una instancia de la superclase o clase base y una instancia de la subclase Una superclase se compone de propiedades que describen un objeto (relaciones *es-un, un-tipo-de*).

Una agregación relaciona instancias de objetos: un objeto es parte de otro objeto. Las jerarquías de agregación se componen de ocurrencias de objetos que a su vez son parte de un objeto contenedor (relación *parte-de, tiene-un*).

relación de generalización	***es-un***	***un-tipo-de***
relación de agregación	***parte-de***	***tiene-un***

Las jerarquías complejas de objetos suelen constar de relaciones de agregación y de generalización. Este es el caso de los dispositivos Avión que incluyen en su jerarquía ambos tipos de relaciones.

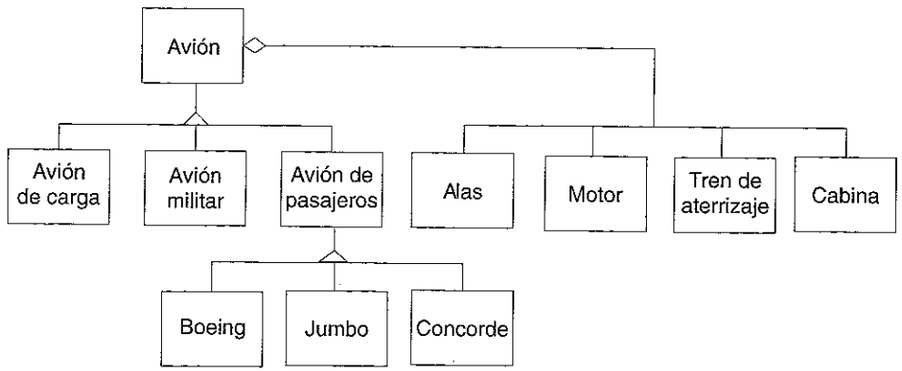


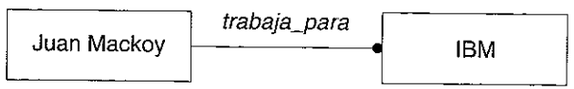
Figura 5.11. Jerarquía compleja de agregaciones y generalizaciones.

La relación de agregación se implementa mediante objetos compuestos.

5.4. RELACION DE ASOCIACION

Una asociación representa una dependencia semántica entre clases e implica la dirección de esta dependencia. En general, las asociaciones son bidireccionales, aunque pudiesen ser unidireccionales si así se indica expresamente.

Una relación de asociación define una relación de *pertenencia*. Para encontrar relaciones de asociación es preciso buscar frases tales como *«pertenece a»*, *«es miembro de»*, *«está asociado con»*, *«trabaja para»*, etc.

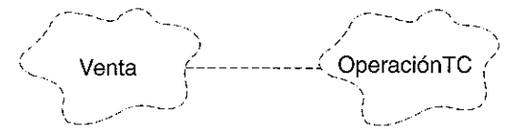


Las asociaciones pueden ser unitarias, binarias, ternarias o de cualquier otro orden, aunque la mayoría serán binarias.

Una propiedad importante intrínseca a la relación de asociación o multiplicidad es la *cardinalidad* o *multiplicidad*. La **multiplicidad** es la propiedad que expresa el número de instancias de una clase que se asocian o conectan con instancias de la clase asociada. Esta propiedad ya fue introducida por Chen para definir el modelo entidad-relación (E/R). Existen tres tipos de multiplicidad o cardinalidad:

- Una a una
- Una a muchas
- Muchas a muchas

Una relación *una-a-una* implica una relación estrecha entre objetos. Por ejemplo, una relación entre una clase `Venta` de un producto y la clase `OperaciónTC` que representa la operación o pago de la venta mediante una tarjeta de crédito.



Cada venta se corresponde con una operación de una tarjeta de crédito y a la inversa. Otro ejemplo es la relación entre `PAIS` y `CAPITAL`. Un país tiene una capital y sólo una, y una ciudad que es capital de un país sólo pertenece a un país.

Una relación *una a muchas* se puede ver entre las clases `PAIS` y `CIUDAD`. Un país tiene muchas ciudades, mientras que una ciudad sólo pertenece a un país.

Otro ejemplo se da entre la clase `LINEA`, `CIRCUNFERENCIA`, o en general `FIGURA` y la clase `PUNTO`. Cualquier figura puede tener muchos (infinitos) puntos, y un punto se asocia a una figura.

Las clases `Venta` y `Artículo` se enlazan también por una relación *una-muchas*. Una venta puede constar de muchos artículos.

Por último, la multiplicidad *muchas-a-muchas* implica que una instancia de una clase puede corresponder con muchas instancias de otra clase, y viceversa. Las clases `Estudiante` y `Asignatura` pueden estar relacionadas con asociaciones de multiplicidad *muchas-a-muchas*. Un estudiante puede estar matriculado en muchas asignaturas, y en una asignatura determinada pueden estar matriculados muchos alumnos.

Las relaciones de multiplicidad o cardinalidad se indican de muy diversas formas, según sea la metodología de análisis y diseño orientada a objetos que se utilice. Las notaciones más usuales son:

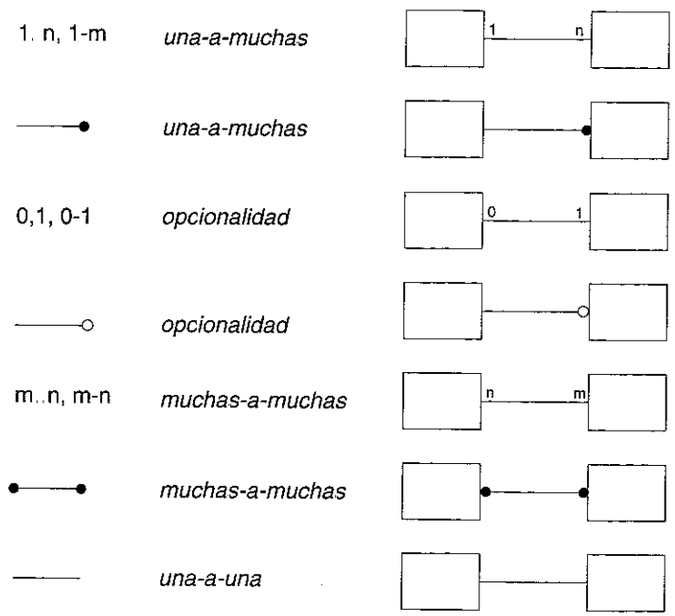


Figura 5.12. Relaciones de cardinalidad (multiplicidad).

5.4.1. Otros ejemplos de cardinalidad

Una asociación *una-a-una* es la que mantienen objetos de la clase `Persona` y de la clase Número de la Seguridad Social (`NSS`). Cada persona tiene un único número de la Seguridad Social y dado un número de la Seguridad Social se corresponde con una única persona. Obsérvese la relación bidireccional entre objetos de estas dos clases.

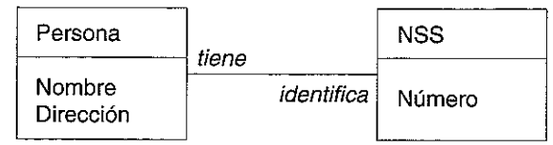


Figura 5.13. Asociación *una-a-una*.

El rol (papel) representado en la línea (una persona *tiene-un* NSS) se lee de izquierda a derecha; sin embargo, la relación inversa (un NSS *identifica* a una persona) está también implicada y se lee de derecha a izquierda.

Una asociación *una-a-muchas* se ilustra en la Figura 5.14 entre una `Persona` y la `Compañía` (empresa) para la que trabaja, suponiendo que una persona *no está* pluriempleada y sólo trabaja para una compañía.

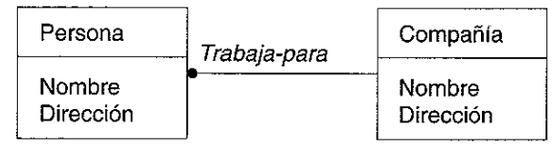


Figura 5.14. Asociación *una-a-muchas*.

Una asociación *muchas-a-muchas* entre la clase `Ventas` y los `Productos` que se compran en cada operación de venta.

Figura 5.15. Asociación *muchas-a-muchas*.

Las asociaciones se implementan mediante punteros o referencias a objetos de las clases

5.5. HERENCIA: JERARQUIA DE CLASES

Una de las herramientas disponibles en lenguajes orientados a objetos más poderosos es la **herencia** Esta construcción permite modelar, del modo más preciso, la realidad que se desea emular en su programa, abstrayendo el comportamiento común entre objetos similares a través de un mecanismo de generalización. Este mecanismo, como ya se ha comentado, proporciona un gran detalle de descripción, que comienza en clases globales y se va extendiendo a través de subclases específicas y especializadas.

En otras palabras, la herencia permite crear muchas clases que son similares entre sí, sin tener que rescribir cada vez las partes que son similares; esta propiedad permite combinar varias clases en una de ellas o modificar una clase existente sin modificar realmente el código original. *La herencia es el corazón de la programación orientada a objetos y constituye el bloque fundamental de construcción para reutilizar el código.* El segundo bloque de construcción es el *polimorfismo*. La herencia es una técnica natural y útil para organizar programas.

La **herencia**, en esencia, es una relación entre clases, en donde una clase comparte la estructura o comportamiento definida en una clase (*herencia simple*) o varias clases (*herencia múltiple*).

La clase superior en la jerarquía se denomina **superclase** (*clase base* en C++); las clases que heredan los miembros de la superclase (incluyendo funciones y datos) se denominan **subclase** (*clase derivada* en C++). Las clases derivadas pueden también modificar miembros de la clase base o añadir nuevos miembros Este proceso puede continuar, de modo que una clase derivada servirá normalmente como una clase base, a partir de la cual se definen otras clase derivadas Por consiguiente, se crean jerarquías de clases, en las que cada clase puede servir como un padre o raíz de un nuevo conjunto de clases. La Figura 5 16 representa un ejemplo de una jerarquía de clases

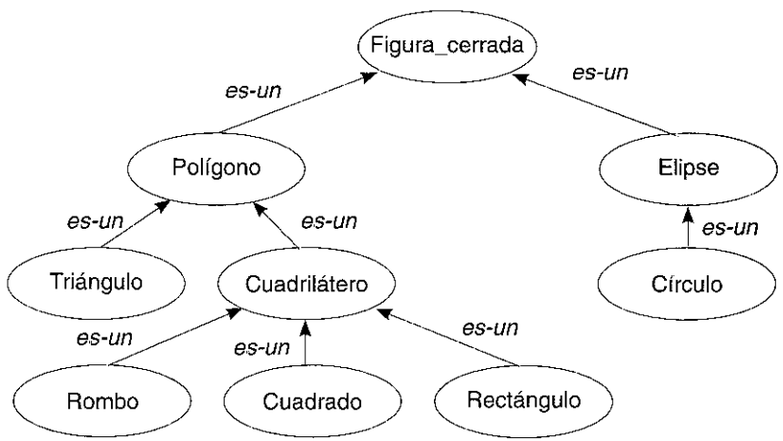


Figura 5.16. Relaciones entre clases (*un tipo de*, o *es un/una...*).

La herencia es la propiedad que implementa una relación de generalización/especialización (**es-un**; *is-a*), en la que una subclase hereda de una o más superclases

La herencia significa que el comportamiento de los datos asociados con las clases hijas son siempre una extensión (es decir, estrictamente hablando, un conjunto más grande) de las propiedades asociadas con las clases padres Una subclase debe tener todas las propiedades de la clase padre y otras. Por otro lado, dado que una clase hija es una forma más especializada de la clase padre, es también, en un cierto sentido, una contracción del tipo padre.

La herencia significa que las subclases heredan la estructura y el comportamiento de sus superclases. La mayoría de los lenguajes de la programación orientados a objetos permiten que los métodos de una superclase se puedan heredar, así como excluir, y también se permite añadir y redefinir métodos en una subclase

> Las superclases se denominan *clases base* o *clases padre*. Las subclases se denominan también *clases derivadas* o *clases hija*.

La herencia se puede utilizar para ayudar a escribir código reutilizable y representar las relaciones entre tipos y subtipos (clases y subclases)

Existen dos tipos de herencia: *herencia simple* y *herencia múltiple*.

5.5.1. Herencia simple

El caso más simple de herencia es la **herencia simple**, en la que una clase *sólo* se deriva de otra clase. Así, por ejemplo, en un caso típico de zoología, un gato o un perro *es un* animal, y en un caso típico de botánica, un clavel *es una* flor. Un gato posee determinadas características de la categoría o clase animal; por otra parte, un gato difiere en determinadas características de la categoría animal. En nuestros ejemplos, «Animal» o «Flor» son las superclases o clases bases, y «Gato» o «Rosa» son las subclases, clases derivadas o clases extendidas o ampliadas.

Otro ejemplo simple puede ser la clase `Profesor` como un tipo de clase derivado de `Persona`

La clase `Profesor` es un tipo de (*es-un*) `Persona`, al que añade sus propias características Las clases se organizan en una estructura lógica denominada *jerarquía de clases* La Figura 5 18 muestra una jerarquía de clases que contiene en su nivel más alto la clase `Persona` y dos subclases o clases derivadas, `Estudiante` y `Profesor` Las subclases *heredan* características de sus superclases Las características pueden ser variables de instancias (campos miembro) y/o métodos (funciones miembro).

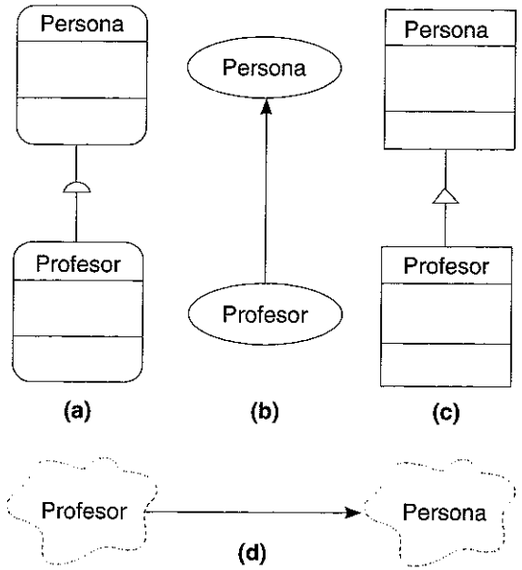


Figura 5.17. Herencia simple: **(a)** notación de Yourdon; **(b)** notación genérica; **(c)** notación OMT; **(d)** notación de Booch.

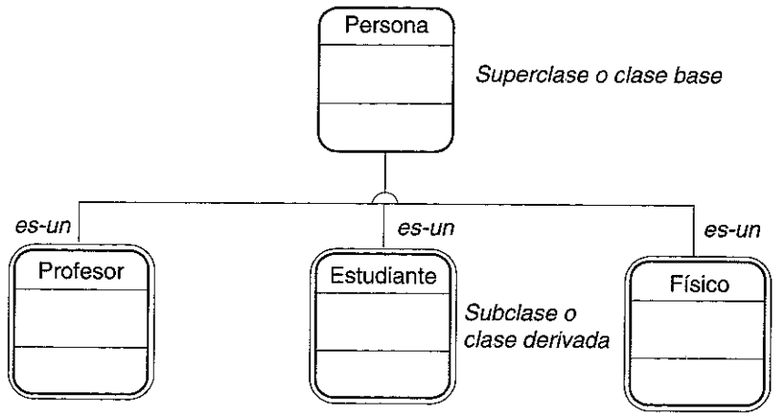


Figura 5.18. Jerarquía de clases: `Persona`, `Profesor` y `Estudiante`.

5.5.2. Herencia múltiple

Hasta este momento sólo se ha utilizado la herencia *simple*: cada subclase o clase derivada tiene una y sólo una superclase o clase base. Supongamos una jerarquía de clases `Persona`, donde sólo existen relaciones de herencia simple.

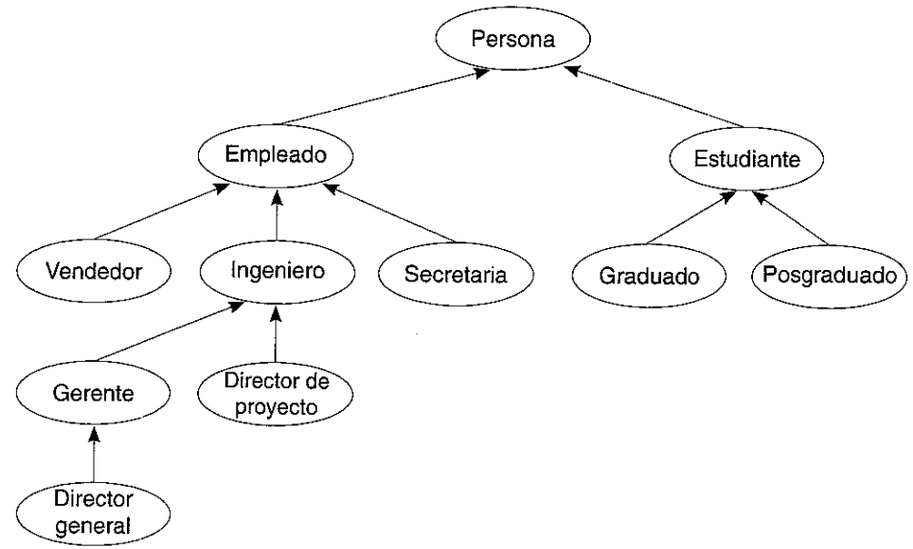


Figura 5.19. Jerarquía de clase `Persona` (*herencia simple*).

La herencia simple no puede expresar relaciones múltiples, por ejemplo aquellas personas que sean a la vez empleados y estudiantes. En realidad, existen numerosos ejemplos en la vida diaria de relaciones de herencia múltiple: un `Fabricante de motocicletas japonesas` tiene propiedades (variables instancia y métodos) que pertenecen a `Compañía japonesa` y a `Fabricante de motocicletas`; una ventana `VentanaTextoBordeada` permite editar texto en una ventana con bordes que hereda de `VentanaTexto` y de `VentanaConBordes`; por último, `Robocop` actúa tanto de `robot` como de `policía`.

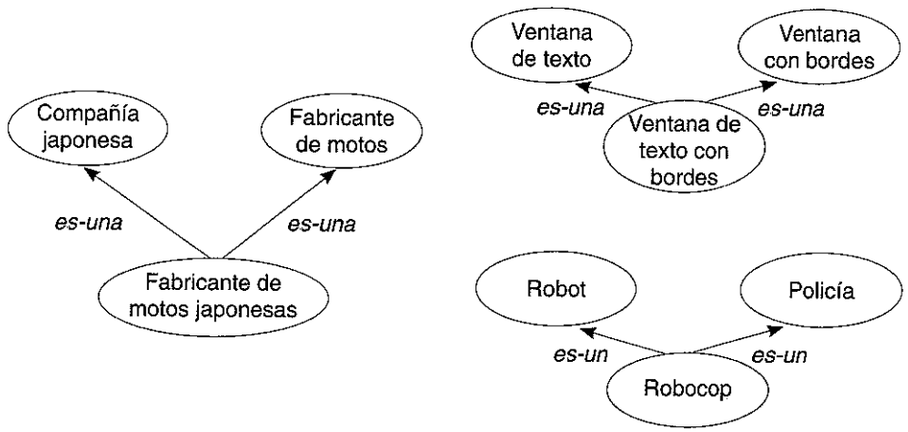


Figura 5.20. Jerarquía de herencia múltiple.

El mecanismo que permite a una clase heredar de más de una clase se llama *herencia múltiple*; se dice entonces que una clase es una *extensión* de dos o más clases. Con herencia múltiple se pueden combinar diferentes clases existentes para producir combinaciones de clases que utilizan cada una de sus múltiples superclases. La representación gráfica en este caso se suele hacer con un grafo dirigido no simétrico, ya que una clase puede tener más de una predecesora inmediata.

La Figura 5.21 proporciona un ejemplo de la jerarquía de clase `Persona` utilizando herencia múltiple. Como en ella se ilustra, el `GerenteVentas` hereda de `Gerente` y `Vendedor`; de modo similar, `EstudianteTrabajador` hereda de `Estudiante`; `DirectorDeProyectos` hereda de `Gerente` y de `Ingeniero`; por último, `SecretarioTécnico` hereda de `Secretario` y de `Ingeniero`.

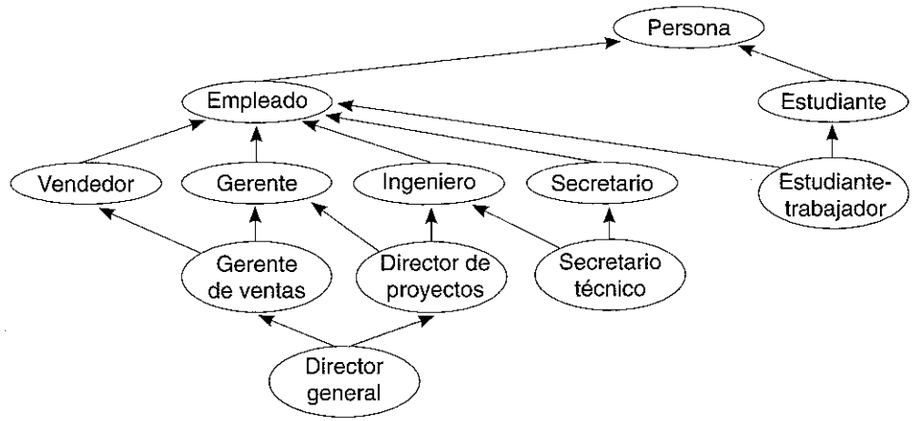


Figura 5.21. Jerarquía de clases `Persona` con herencia múltiple.

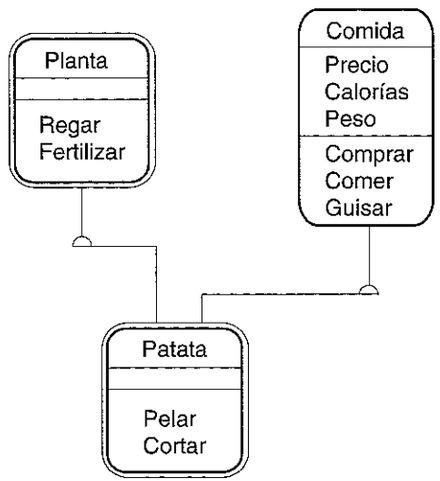


Figura 5.22. Jerarquía de clases `Patata` con herencia múltiple.

Otro ejemplo de herencia múltiple se puede ver en la Figura 5.22, en donde `Comida` y `Planta` son dos clases que actúan como superclases de `Patata`, que es un alimento y a la vez una planta.

La herencia múltiple es una herramienta muy potente, pero es fácil abusar de ella y caer en graves errores; por el contrario, utilizada con precaución, es una ayuda valiosísima en el desarrollo orientado a objetos. De cualquier forma, la herencia múltiple ha sido y sigue siendo tema de debate entre expertos de programación y de lenguajes orientados a objetos.

5.5.2.1. Ventajas de la herencia múltiple

La herencia múltiple es útil en muchas situaciones. Puede ayudar, fundamentalmente, a modelar objetos en su dominio del problema. Una aplicación muy frecuente de la herencia múltiple se suele dar cuando se crea una nueva clase a partir del comportamiento de dos o más clases, incluso aun cuando fueran desarrolladas independientemente unas de otras. Un uso muy común es añadir *persistencia*[6] a los objetos.

Otra ventaja apreciable es su alto grado de flexibilidad, simplicidad y elegancia en la definición de nuevas clases que se crean a partir de clases existentes.

La herencia múltiple favorece claramente la reutilización, por la razón importante que permite más libertad en la definición de nuevas clases a partir de las existentes. Esta construcción permite crear jerarquías de clases completas más fácilmente, sin restringir las relaciones en las jerarquías de clases o casos singulares. En este sentido, la herencia múltiple favorece un enfoque más flexible para el diseño de aplicaciones.

Además, la herencia múltiple facilita el cambio de la implementación de una clase, mientras deja inalterado su interfaz. De hecho, puede cambiar simplemente la parte de una clase que corresponda a una de sus clases base, heredando de una clase base diferente que realiza las mismas funcionalidades en diferentes formas, obteniendo, en consecuencia, una implementación diferente de la misma abstracción.

Sin embargo, hemos de reconocer que el uso generalizado de plantillas (*templates*) ha reducido significativamente las ventajas de la herencia múltiple.

5.5.2.2. Inconvenientes de la herencia múltiple

Existen problemas asociados al uso de la herencia múltiple. Con frecuencia se produce confusión y comportamiento impredecible, debido al uso de la

[6] Los *objetos persistentes* —en contraposición a los *objetos transitorios*— son aquellos que permanecen activos entre ejecuciones. El tiempo de vida de un objeto es la duración de la ejecución del programa; una vez que el programa termina su ejecución, todos los objetos que estaban activos se vuelven inaccesibles. Los objetos persistentes son aquellos que al almacenarse en disco permanecen intactos entre ejecuciones. La *persistencia* o mejor el soporte de objetos persistentes es fundamental en los sistemas de gestión o administración de bases de datos orientadas a objetos. Para ampliar conceptos sobre objetos persistentes se puede estudiar a Booch, Khosafian y Abnous, Graham y Mary Loomies, entre otros.

herencia a partir de clases con métodos que tienen los mismos nombres pero significados diferentes; es decir, se produce ambigüedad. Todo lenguaje que soporte herencia múltiple ha de tener reglas propias para resolver esta ambigüedad. Otro inconveniente de la herencia múltiple es el aumento de tiempo auxiliar que se añade a los programas.

La herencia múltiple está soportada por C++, CLOS y Objective-C. Sin embargo, Object-Pascal, Turbo Pascal 5.5/6/7 y Smalltalk no soportan esta propiedad, aunque algunas versiones de Smalltalk también permiten esta característica.

5.5.2.3. Diseño de clases con herencia múltiple

El diseño de una estructura de clases adecuada que implica herencia, especialmente herencia múltiple, es una tarea difícil. Suele ser un proceso interactivo e incremental. Dos problemas se suelen presentar cuando se manipula herencia múltiple: ¿cómo resolver las colisiones de nombre de diferentes superclases? Y ¿cómo manipular herencia repetida o herencia de ascendientes comunes?

Las *colisiones* de *nombres* se producen cuando dos o más superclases diferentes tienen el mismo nombre para algún elemento de sus interfaces, tales como variables instancia y métodos. Y la *herencia repetida* se produce cuando una clase es ascendiente de otra clase por más de un camino. Supongamos que las clases `empleado` y `estudiante` tienen las estructuras de variables instancias que se muestran en la Figura 5.23.

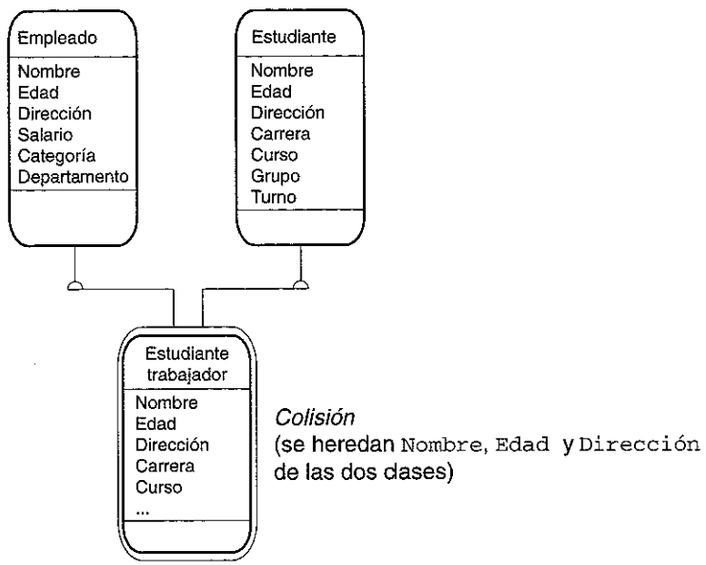


Figura 5.23. Colisión de nombres de atributos.

La clase `EstudianteTrabajador` hereda de `Empleado` y `Estudiante`. Una variable instancia de `Estudiante` puede contener las variables `{Nombre, Edad, Dirección, Carrera, Curso, Grupo, Turno}` y una variable instancia de `Empleado` contendrá las variables `{Nombre, Edad, Dirección, Salario, Categoría, Departamento}`.

Booch considera que existen básicamente tres métodos para resolver las colisiones o choque de nombres:

1. La semántica del lenguaje puede considerar una colisión de nombres como ilegal y rechaza la compilación de la clase. Este es el método utilizado por Smalltalk y Eiffel. Sin embargo, en Eiffel es posible renombrar elementos de modo que no exista ambigüedad.
2. La semántica del lenguaje puede considerar el mismo nombre introducido por clases diferentes con referencia al mismo elemento conflictivo. Es el método de CLOS.
3. La semántica del lenguaje puede permitir el choque o conflicto, pero requiere que todas las referencias de nombres califiquen la fuente de su declaración. Es el método utilizado por C++.

5.6. HERENCIA REPETIDA

El otro problema grave que se produce en el uso de la herencia múltiple es la *herencia repetida*. Este tipo de herencia se produce cuando una clase hereda de dos o más superclases que a su vez heredan de la misma superclase. Esta situación se presenta, por ejemplo, en el caso de la Figura 5.24, en la que `EstudianteTrabajador` es una subclase de `Empleado` y `Estudiante`, que a su vez son subclases de `Persona`. Suponiendo que `Persona` tiene los atributos `Nombre, Edad, Dirección` que se reciben por herencia de `EstudianteTrabajador`. En este caso, los atributos anteriores se repetirán en la última subclase.

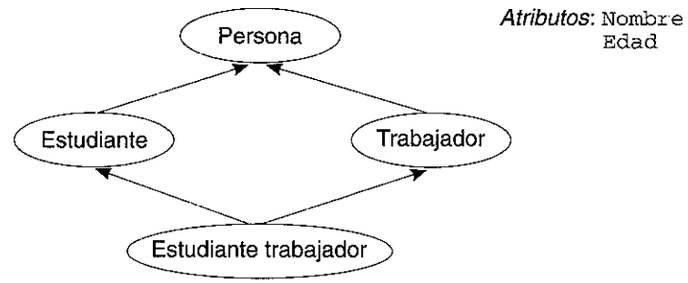


Figura 5.24. Herencia repetida.

La mayoría de los lenguajes de programación no permiten la duplicación estática de la superclase, pero eso no se producirá siempre, y así se puede dar el caso de que el compilador duplique la clase que se hereda dos o más veces. En

la Figura 5.25 se muestran grafos de herencia repetida, con lo que se crea una copia o dos copias (instancias), según el caso.

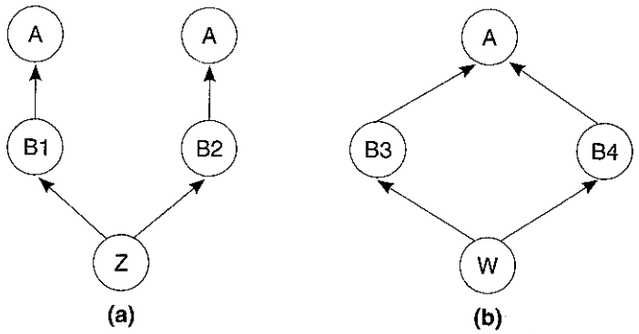


Figura 5.25. Grafos de herencia repetida: **(a)** creación de dos copias de la superclase; **(b)** creación de una sola copia de la superclase.

Estas dos clases diferentes de herencia repetida corresponden a dos significados muy diferentes de esta construcción, como se ilustra en los ejemplos siguientes.

El primer ejemplo es una clase que describe una `Persona` (con atributos de datos de la persona y otra información). Se crean dos nuevas clases a partir de la clase `Persona` por herencia: la clase `Profesor` de universidad (se añaden datos adicionales relativos a la universidad donde imparte docencia, asignaturas y cursos a su cargo); la otra clase que se crea por herencia es `Autor` de libros (con algunos nuevos atributos, tales como títulos de libros publicados, años de publicación y editoriales). En esta situación se pueden considerar profesores que son autores de libros, y cuya estructura jerárquica se muestra en la Figura 5.26. En este caso es más correcto utilizar el caso de la herencia múltiple de la Figura 5.25b, ya que un `ProfesorAutor` es *sólo* una persona (¡y no dos!), y en consecuencia sus datos privados se han de duplicar.

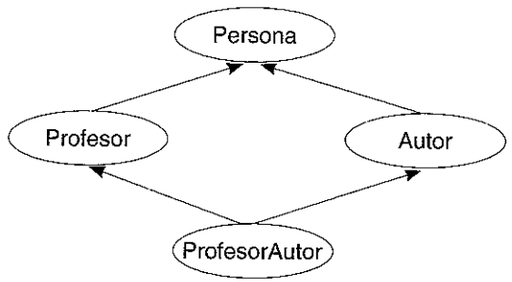


Figura 5.26. Grafo de herencia repetida (una copia de la superclase).

Por el contrario, si consideramos una clase que describa a una pareja o matrimonio que se derive de dos clases, `Hombre` y `Mujer`, respectivamente, la clase `Persona` debe estar presente dos veces, ya que el matrimonio lo constituyen dos personas y no una. La Figura 5.27 muestra otro ejemplo de herencia repetida.

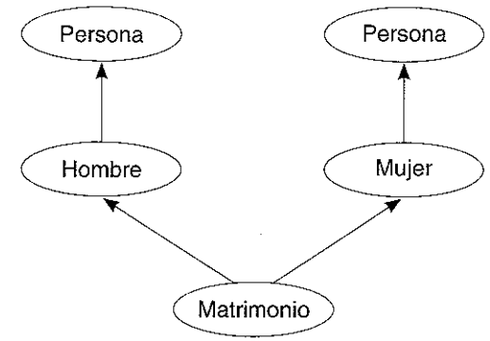


Figura 5.26. Grafo de herencia repetida (dos copias de la superclase).

El grafo de la Figura 5.28 muestra cómo la clase D tiene dos copias, instancias o subobjetos de la clase A: una copia A de W y otra copia —llamada *virtual* en C++— compartida por B y C.

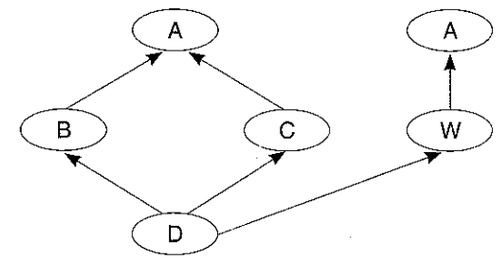


Figura 5.28. Otro caso de herencia repetida.

Existen tres métodos para tratar el problema de la herencia repetida:

1. Tratar la ocurrencia de herencia repetida como ilegal. Este es el enfoque de Smalltalk y Eiffel (aunque Eiffel también permite el cambio de nombre para evitar las ambigüedades).
2. Se permite la duplicación de superclases, pero requiere el uso de nombres cualificados totalmente para referirse a los miembros de una copia específica. Este es el método empleado por C++.

3. Se puede tratar referencias a la misma clase como si fueran la misma clase. Este es el método empleado por C++ cuando se introducen superclases repetidas como clases base virtual. Una *clase base virtual* existe cuando una subclase nombra a otra clase como su superclase y marca la superclase como virtual, para indicar que es una clase compartida.

RESUMEN

Los modelos de objetos describen la estructura de datos estática de los objetos, clases y sus relaciones entre sí. Una clase de objetos describe un grupo de objetos con atributos, operaciones y semántica comunes. Un atributo es una propiedad de los objetos de una clase; una operación es una acción que se puede aplicar a objetos de una clase.

Las relaciones entre clases pueden ser: generalización/especialización, agregación y asociación.

Las asociaciones establecen relaciones entre objetos y clases. Un enlace conecta dos o más objetos. La multiplicidad especifica cuántas instancias de una clase se pueden relacionar con cada instancia de otra clase.

Una *agregación* es una relación en la que un objeto de una clase se compone de una serie de objetos de diferentes clases; así, un motor de un coche (carro) se compone de bujías, válvulas, cilindros, etc.

El término *generalización* es útil para construir modelos conceptuales de datos e implementación. Durante el modelo conceptual, la generalización permite al desarrollador organizar clases en una estructura jerárquica basada en sus semejanzas y diferencias. Durante la implementación, la herencia facilita la reutilización de código. El término *generalización* se refiere a las relaciones entre clases; el término *herencia* se refiere al mecanismo de obtener atributos y operaciones utilizando la estructura de generalización. La generalización proporciona los medios para redefinir una superclase en una o más subclases. La superclase contiene características comunes a todas las clases; las subclases contienen características específicas de cada clase. La herencia puede producirse a través de un número arbitrario de niveles, en donde cada nivel representa un aspecto de un objeto.

La herencia entre clases puede ser *simple* y *múltiple*.

EJERCICIOS

5.1. Crear una clase `Pila` con la siguiente estructura:

miembros dato (creación protegida)	array de enteros cima de tipo entero	`pila` `cima`
funciones miembro:	`meter` datos en la pila `sacar` datos en la pila	

Esta clase tiene el inconveniente de no detectar desbordamientos positivos o negativos de la pila. En consecuencia, se decide diseñar una nueva clase `PilaDer` derivada de `Pila`, con las mismas funciones miembro `meter` y `sacar`, pero que adviertan al usuario con mensajes de «Pila vacía» o «Pila llena» cuando se intente sacar un elemento de una pila vacía o meter un elemento en una pila llena. Escribir programa que gestione la pila con ambas clases.

5.2. Se dispone de la clase `obj_geom`:

```
#include <iostream,h>

class obj_geom {
protected:
  float xC, yC;
public:
  obj_geom(float x =0, float y =0) { xC =x; yC = y; }
  void vercentro () { cout << xC << "" << yC << end;}
};
```

Diseñar clases `círculo` y `cuadrado` derivadas del `obj_geom` que permitan calcular sus áreas. Una vez diseñadas todas las clases, escribir un programa que cree un objeto de cada clase, visualice los centros de cada figura y a continuación calcule y visualice las áreas de cada figura.

5.3. Una editorial de libros y discos desea crear fichas que almacenen el título y el precio (de tipo `float`) de cada publicación. Crear la correspondiente clase (denominada Publicación) que implemente los datos anteriores. A partir de esta clase, diseñar dos clases derivadas: `Libro`, con el número de páginas (tipo `int`), año de publicación (tipo `int`) y precio (tipo `float`); y `disco`, con duración en minutos (tipo `float`) y precio (tipo `int`). Cada una de las tres clases tendrá una función y otra función `mostrar()`, para visualizar sus datos.

Escribir un programa que cree instancias de las clases `Libro` y `disco`, solicite datos del usuario y a continuación los visualice.

5.4. Se dispone de la clase `publicación` del ejercicio 5.3 y se desea crear una nueva clase base llamada `ventas` que contenga un array con las ventas del último semestre de una determinada publicación. Esta clase debe tener funciones miembros `Leer()` y `mostrar()` que obtenga y visualice las citadas ventas. Modificar las clases `Libro` y `disco`, lea y visualice las publicaciones.

5.5. El departamento de informática de un hospital está realizando un nuevo registro de datos del personal, pacientes y proveedores del hospital y desea realizar la jerarquía de clases siguientes:

Escribir las clases correspondientes de acuerdo a las siguientes estructuras:

Persona	*Paciente*	*Empleado*
nombre	nombre	nombre
dirección	dirección	dirección
ciudad	ciudad	ciudad
Leer( )	código_diagnóstico	código_empleado
visualizar( )	teléfono	horas_extras
	fecha de nacimiento	compañía de seguros
	Leer( )	Leer( )
	visualizar( )	visualizar( )
	enviar_factura	enviar_salario

Proveedor	*Plantilla*	*Eventual*
nombre	datos de empleado	datos de empleado
dirección	salario sumal	honorarios/hora
ciudad	años de antiguedad	pagar_salario( )
código vendedor	pagar_salario	
saldo		
fax		
teléfono		
descuentos		
Leer( )		
visualizar		
pagar_factura( )		

5.6. Diseñar una clase `nombre` que contenga tres miembros datos (nombre, primer apellido y segundo apellido), un constructor y dos funciones miembro `Leer_nombre()` que obtiene valores para los miembros dato de la clase `mostrar()` que ofrece la visualización del nombre complemento. Diseñar otra clase derivada `dirección` que tome la información de nombre y añada calle, ciudad, provincia y código postal. Esta clase debe tener acceso a las funciones públicas de la clase base y tres funciones miembro `nueva_dirección`, `nuevo_nombre()` y `mostrar()`. Escribir un programa que cree un objeto de `dirección`, lea datos y visualice la información.

5.7. Diseñar una jerarquía de clases: `Círculo`, `Cilindro` y `CilindroHueco`. En esencia, se puede decir que un objeto cilindro es un objeto círculo con una altura, y un cilindro hueco es un cilindro con un espacio hueco dentro de él. La clase `Círculo` debe tener un dato `Radio` (tipo `double`) y unas funciones miembro `LeerRadio`, `Area` y `Circunferencia`, que obtienen el valor del radio y calculan el área del círculo y la longitud de la circunferencia. Escribir un programa que permita crear objetos `Círculo`, `Cilindro` y `CilindroHueco` y calcule la longitud de la circunferencia y las áreas del círculo, del cilindro y del cilindro hueco.

Fórmulas

Círculo	Longitud	$2 \cdot \pi \cdot r$
	Area	$\pi \cdot r^2$
Cilindro	Area	$2 \cdot \pi \cdot r \cdot h + 2 \cdot \pi \cdot r^2$
	Volumen	$\pi \cdot r^2 \cdot h$
Cilindro hueco	Longitud	$2 \cdot \pi \cdot (r^2 - r^2 interno) + 2 \cdot \pi \cdot r \cdot h + 2 \cdot \pi \cdot h \ interno$
	Volumen	$\pi \cdot (r^2 - r^2 interno) \cdot h$

Nota: r = radio del cilindro y radio externo cilindro hueco.
r interno = radio interno del cilindro hueco.