2 Tecnología Orientada a Objetos

Dado que un aspecto primordial de este libro es el software orientado a objetos es entonces necesario comprender que significa esta tecnología. Comenzamos discutiendo brevemente cuales son los mitos y cuales las realidades con esta tecnología. Continuamos describiendo los aspectos básicos que distinguen a la programación orientada a objetos con respecto a la manera tradicional de programación. El resto del capítulo describirá la motivación, y conceptos detrás de esta tecnología junto con una breve reseña de los más importantes lenguajes orientados a objetos.

2.1 Mitos y Realidades

La orientación a objetos es un buen ejemplo de cómo un "pequeño detalle" puede significar tan crítico, algo similar a la famosa frase de Neil Armstrong cuando pisó la luna: "Un pequeño paso para un hombre, un gran paso para la humanidad". Sin exagerar con la similitud analicemos qué significa este pequeño paso tecnológico que tanto ha significado para el desarrollo de software.

2.1.1 Programación Tradicional

En la programación tradicional, conocida como *estructurada*, es separar los datos del programa de las funciones que los manipulan. El programa o aplicación completa, consiste de múltiples datos y múltiples funciones, como se muestra en la Figura 2.1.

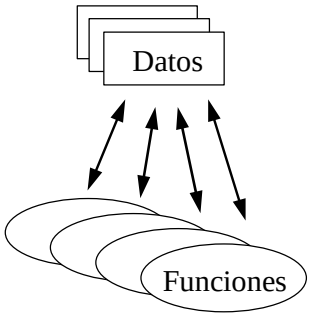

Figura 2.1 Programación estructural: datos y funciones globales.

Esta forma de programar tiene sus orígenes en la arquitectura "von Neumann" de las primeras computadoras modernas. La arquitectura básica es la misma utilizada en la actualidad a nivel comercial en las PCs y se basa de manera simplificada en una unidad central de procesamiento (CPU) y una memoria donde se carga el programa o aplicación que debe ejecutarse. (El disco duro guarda a largo plazo la aplicación para que ésta no se pierda pero no juega un papel primordial cuando la aplicación se ejecuta.) La memoria en sí se divide en una sección donde se guardan las funciones del programa, correspondiente al código que controla la lógica de la aplicación, y otra sección de datos donde se guarda la información que quiere manipularse. Dada esta separación entre funciones y datos en la memoria lo más lógico siempre ha sido utilizar una programación que se ajustara a ello dando origen a un gran número de lenguajes basados en esta estructuración.

Esta manera de programar tiene dos problemas principales. El primer problema es obligar a un programador a pensar como la máquina, en lugar de lo opuesto. El segundo problema es que toda la información presente es conocida y potencialmente utilizada por todas las funciones del programa y si se hiciera algún cambio en la estructura de alguno de los datos (se consideran todos como "globales"), potencialmente habría que modificar todas las funciones del programa para que éstas pudieran utilizar la nueva estructura.

¿Que tan problemático pudiese ser esto? Pues que mejor ejemplo que el problema del año 2000 donde un dato tan insignificante como la fecha, que al cambiarse de dos a cuatro dígitos resultó en costos mundiales de cerca de $1 trillón de dólares. Lo que empeoró las cosas fue que todos estos programas tenían miles de funciones donde cada una de ellas requería de la fecha para funcionar correctamente, cómo en el caso de aplicaciones bancarias y nóminas de compañías.

2.1.2 Programación Orientada a Objetos

¿Cómo puede ayudarnos la orientación a objetos a solucionar los dos problemas principales de la programación tradicional? La respuesta es que la orientación nos ayuda a mejorar radicalmente ambas situaciones gracias a que la unidad básica de programación es el *objeto*. A nivel organizacional el concepto del objeto nos acerca más a la manera de pensar de la gente al agregar un nivel de abstracción adicional donde internamente la estructura del programa se ajusta a la arquitectura de la máquina. En relación al segundo problema, los datos globales desaparecen, asignando a cada objeto sus propios datos y funciones locales, resultando en un programa o aplicación definido exclusivamente en término de objetos y sus relaciones entre sí, como se muestra en la Figura 2.2.

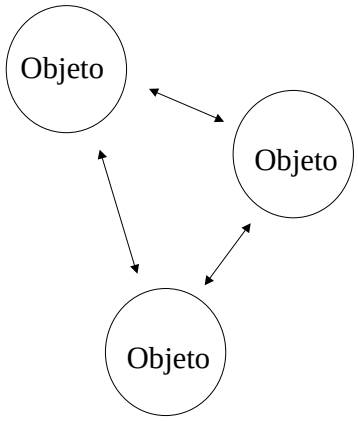

Figura 2.2 Programación orientada a objetos: objetos globales.

Obviamente debemos tener datos y funciones para que un programa tenga sentido, pero estos son guardados en cada objeto de manera independiente, como se muestra en la Figura 2.3.

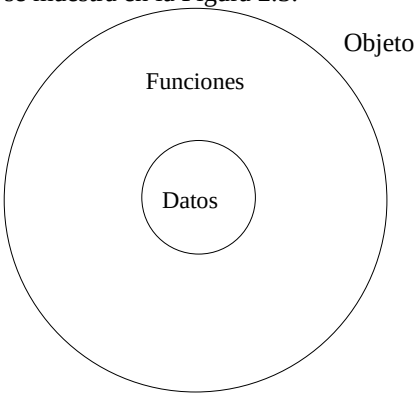

Figura 2.3 Programación orientada a objetos: objetos globales que contienen datos y funciones locales.

Nótese en el diagrama, que los datos están ubicados en el centro del objeto (un concepto puramente ilustrativo) resaltando el efecto de que un cambio en la estructura de uno de estos datos sólo afecta a las funciones del mismo objeto pero no al resto de la aplicación. Todo lo relacionado al detalle de los objetos junto con sus datos y funciones será descrito en el Capítulo 4.

2.1.3 El Problema del Año 2000 Revisado

¿Cuáles hubieran sido las consecuencias del problema del año 2000 si todas esas aplicaciones hubiesen sido programadas mediante la programación orientada a objetos. La fecha como tal no hubiese sido un dato sino un objeto y aunque el objeto “Fecha” hubiese contenido originalmente dos en lugar de cuatro dígitos, el resto de la aplicación se relacionaría únicamente con el objeto “Fecha” como se muestra en la Figura 2.4.

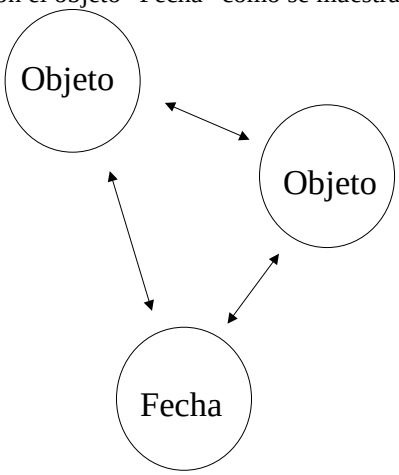

Figura 2.4 El objeto "Fecha" como ejemplo de un objeto.

Llegando el año 2000 donde se reconoce la deficiencia de los dos dígitos se habría cambiado la estructura interna de los datos del objeto “Fecha” a cuatro dígitos solamente afectando las funciones internas encargadas de manipular los datos internos del objeto “Fecha”, como se muestra en la Figura 2.5.

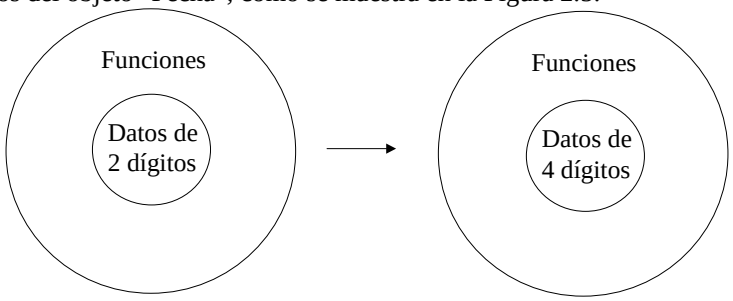

Figura 2.5 Extensión de la estructura de dato de "Fecha" de 2 a 4 dígitos.

El resto de la aplicación nunca se hubiera enterado y el diagrama mostrado en la Figure 2.4 hubiese sido idéntico. ¡Como consecuencia el mundo se hubiera ahorrado $1 trillón de dólares! Un cambio insignificante para un programa pero repercusiones brutales para la humanidad. Por supuesto que aún con tecnología orientado a objetos, un mal diseño no hubiese solucionado el problema, ¡aunque el desafío para lograr malos diseños es mayor!

2.2 Programación Orientada a Objetos

El software orientado a objetos apoya ciertos aspectos que mejoran la robustez de los sistemas, este software requiere de ciertas características mínimas para considerarse orientado a objetos y finalmente debe integrarse como parte de un lenguaje de programación. Estos temas son descritos a continuación.

2.2.1 Aspectos que Mejoran la Robustez de los Sistemas

Existen razones un poco más técnicas que motivan a la orientación a objetos, como son la *abstracción*, *modularidad*, *extensiblidad* y *reutilización*.

?? **Abstracción.** Una de las consideraciones más importantes para tratar el problema de la complejidad del software es el concepto de *abstracción*. La idea básica de la abstracción es reducir el nivel de primitivas o representaciones básicas necesarias para producir un sistema de software. De manera sencilla esto se logra mediante el uso de lenguajes de programación que contengan estructuras de datos de alto nivel. En otras palabras, la pregunta opuesta sería: ¿por qué no programar en código binario, o sea 0s y 1s ? La respuesta es que ninguna persona sería capaz de comprender una aplicación al verse el código y por otro lado requeriría de programas extremadamente extensos para representar la aplicación completa dada la simplicidad de la primitiva básica. Los sistemas de software construidos con lenguajes de programación de más alto nivel reducen el número total de líneas de código por lo tanto reducen su complejidad. Con la programación orientada a objetos se definen dos niveles de abstracción. El nivel más alto, el de los objetos, es utilizado para describir la aplicación mientras que el nivel más bajo, el de los datos y las funciones, es utilizado para describir sus detalles. Este nivel inferior corresponde al único nivel de la programación tradicional. Esto refleja que la complejidad se maneja de mejor manera con la tecnología orientada a objetos. En general cuanto más podamos simplificar las tareas de desarrollo mejor será el manejo de la complejidad. Por otro lado el objeto como estructura básica sirve para separar el “que” de una aplicación del “como”, o sea sus detalles, al contrario de la programación tradicional donde el “que” y el “como” se resuelven a la vez.

?? **Modularidad.** Otro aspecto importante de una aplicación es su *modularidad*. La modularidad de un sistema depende de sus abstracciones básicas, lo cual permite dividir el sistema en componentes separados. Al tener abstracciones de mayor nivel la modularidad de los componentes también es de mayor nivel reduciendo el número final de componentes lo cual a su vez simplifica su manipulación y mantenimiento. Con la orientación a objetos, la modularidad del sistema se da en base a objetos, un nivel más alto que los datos y funciones tradicionales. El número final de módulos, o sea objetos, es menor que el número original de datos y funciones. Esto reduce la complejidad de la aplicación ya que el programador piensa en menos componentes a la vez descartando detalles innecesarios.

?? **Extensibilidad.** En general, los sistemas de software tienden a ser modificados y ampliados durante el transcurso de su vida. Como se mencionó en el Capítulo 1, la “Ley de Lehman” dice que todo programa que se use se modificará. O sea, si un programa no se modifica es porque nadie lo quiere usar, por lo cual uno se pregunta: ¿que tan larga es la vida de un sistema? En otras palabras, ¿cuándo se vuelve más costoso mantener un sistema de software que desarrollar uno nuevo? La *extensibilidad* tiene como objetivo permitir cambios en el

sistema de manera modular afectando lo mínimo posible el resto del sistema. Con la orientación a objetos, los cambios se dan a dos niveles: modificación externa e interna de los objetos. Los cambios internos a los objetos afectan principalmente al propio objeto, mientras que los cambios externos a los objetos afectarán de mayor forma al resto del sistema. Dada la reducción en el número de entidades básicas en un sistema mediante abstracciones de nivel más alto, se logra un desarrollo de sistemas más estables con menor complejidad, y por lo tanto más fácilmente extensibles.

?? **Reutilización.** Una de las maneras de reducir la complejidad del software es mediante la *reutilización* o *reuso* de partes existentes. La pregunta que uno se hace es: ¿cuánto puedo reutilizar del código y sistemas ya existentes? El reuso de código reduce el tiempo del diseño, la codificación, y el costo del sistema al amortizar el esfuerzo sobre varios diseños. El reducir el tamaño del código también simplifica su entendimiento, aumentando la probabilidad de que el código sea correcto. Mediante el reuso de código se puede aprovechar componentes genéricos para estructurar bibliotecas reutilizables, y así lograr una estandarización y simplificación de aplicaciones por medio de componentes genéricos prefabricados. Tradicionalmente, los componentes o librerías de software han existido por muchos años como procedimientos y funciones, particularmente para aplicaciones numéricas y estadísticas. Y aunque el reuso es posible en lenguajes convencionales, los lenguajes orientados a objetos aumentan substancialmente las posibilidades de tal reuso, gracias a la modularidad de los sistemas. En particular, lenguajes como Java ofrecen componentes de estructuras de datos básicas como colas, pilas, listas, árboles, junto con aquellas de más alto nivel, utilizadas por ejemplo para la construcción de interfaces de usuario facilitando el desarrollo de nuevas aplicaciones. La problemática mayor de la reutilización radica en que para construir componentes genéricos, sencillos, con interfaces bien definidas y que puedan utilizarse en varias áreas de aplicación el esfuerzo es mucho mayor que para construir componentes que serán utilizados en una aplicación. Con la orientación a objetos, el *objeto* es la unidad de reuso más pequeña, pudiéndose aprovechar definiciones similares de objetos dentro de la misma aplicación o incluso en distintas aplicaciones. Al agrupar objetos similares se puede lograr reutilización de componentes de más alto nivel. Por otro lado, se puede aprovechar objetos con estructuras de datos y funciones similares, definiendo una sola vez los aspectos comunes y *especializándolos* en objetos adicionales. A un nivel más amplio existen los *marco de aplicación* ("frameworks") donde una aplicación genérica en un dominio particular se especializa para diferentes ambientes, algo promovido con diferente éxito por compañías como SAP y PeopleSoft. Al definir una aplicación en términos suficientemente abstractos o generales, se puede en teoría especializar su comportamiento sin tener que hacer ningún cambio en la estructura básica de los componentes y de la propia aplicación. Esto extendería de manera radical la utilidad de la aplicación. Esto sería el elixir de la ingeniería de software, lograr crear nuevas aplicaciones sin escribir una sola línea de código, solamente integrando componentes ya existentes, como en la construcción de casas o puentes prefrabricados. Dado que es difícil lograr grandes niveles de reutilización sin contar con niveles intermedios, se ha realizado un esfuerzo muy importante conocido como "Patrones de Diseño" ("Design Patterns"), algo que discutiremos con mayor detalle en el capítulo de diseño.

2.2.2 Características Mínimas de los Lenguajes Orientados a Objetos

En la sección anterior mencionamos la motivación detrás de la orientación a objetos: lograr mayor productividad en el desarrollo de software y mejorar la calidad de éste mediante niveles más altos de abstracción, apoyo a la modularidad, extensiblidad y reutilización de código.

En esta sección describimos los conceptos básicos que hacen que un lenguaje sea considerado efectivamente orientado a objetos. En general, cuatro aspectos deben existir en tal lenguaje: *encapsulamiento*, *clasificación*, *generalización* y *polimorfismo*. En esta sección únicamente introducimos los conceptos, los cuales serán descritos en mucho mayor detalle y ejemplos en el Capítulo 4.

?? **Encapsulación.** *Encapsulación* o *encapsulamiento* es la separación de las propiedades externas de un objeto, o sea su interface, correspondiente a la interface de sus funciones, de los detalles de implementación internos del objeto, o sea sus datos y la implementación de sus funciones, como se muestra en la Figura 2.5. Esta separación es muy importante. Si nos referimos al diagrama de la Figura 2.2, realmente el conocimiento de un objeto por otros objetos en la misma aplicación es exclusivamente en base a la interface de dichos objetos. Todo el detalle, al estar encapsulado, es desconocido por el resto de la aplicación, limitando el impacto de cualquier cambio en la implementación del objeto, ya que los cambios a las propiedades internas del objeto no afectan su interacción externa. Obviamente cualquier cambio en la propia interface del objeto afectaría potencialmente a todo el resto de la aplicación. Sin embargo el porcentaje de código dedicado a las interfaces es por lo general "muchísimo" menor que el porcentaje total de líneas de código utilizados para datos e implementación de funciones. De tal manera se reduce la complejidad del sistema protegiendo los objetos contra posibles errores, y permitiendo lograr de mejor manera extensiones futuras en la implementación de los objetos.

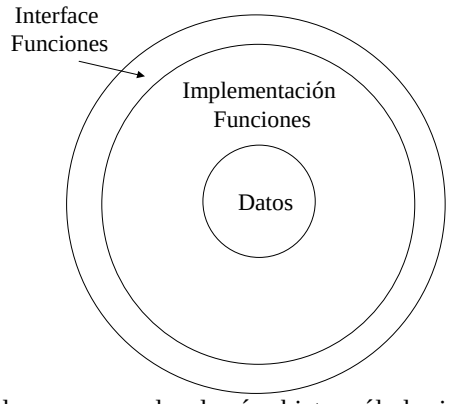

Figura 2.5 Un objeto da a conocer a los demás objetos sólo las interfaces de sus funciones.

?? **Clasificación.** En todo programación orientada a objetos la *clasificación* es un aspecto fundamental, donde objetos que contienen estructuras similares, correspondiente a tipos de datos y funciones similares, se clasifican como pertenecientes a la misma *clase* de objeto. Nótese de que hablamos de *tipos* de datos similares, dado que los valores de los datos aún pueden cambiar en objetos de clase similar. ¡Si todos los valores de los datos tuvieran que ser también iguales entonces todos los objetos de una misma clase serían idénticos, algo que limitaría el alcance de la clasificación además de ser muy aburrido!

?? **Generalización.** Si tomamos como base la clasificación, y consideramos que no sólo los objetos similares pueden clasificarse, sino también las propias clases de los objetos, entonces se define la *generalización* o *especialización* de clases. Mediante la generalización, clases de objetos con estructura y comportamiento similar se reutilizan en la definición de las nuevas clases. Estas nuevas clases se consideran clases más especializadas o *subclases* mientras que las originales se consideran clases más generales o *superclases*. El mecanismo para describir jerarquías de generalización de clases se conoce como *herencia*, un término muy utilizado en la orientación a objetos, se dice que una subclase hereda de una superclase. La herencia puede ser *sencilla*, donde una subclase hereda de una sola superclase directa, o *múltiple*, donde una subclase hereda de múltiples superclases directas. La herencia es también una forma de reutilización de código, ya que se aprovechan descripciones de clases de objetos para luego definir clases de objetos parecidos.

?? **Polimorfismo.** Quizás el concepto más complicado de explicar y en cierta manera el más poderoso es el *polimorfismo*. De manera simplificada, mediante el polimorfismo se definen funciones con el mismo nombre e interfaz en distintas clases de objetos, pero bajo implementaciones distintas. ¿Para qué sirve entonces el polimorfismo? Sin adelantarme a las explicaciones más detalladas que vendrán en el Capítulo 4, el polimorfismo es útil para extender la funcionalidad existente en los objetos del sistema, a nuevos objetos aún desconocidos en ese momento. Es como definir un estándar de interfaces para los objetos la cual debe ser seguida por todos los existentes y nuevos. Haciendo una analogía, todo el tiempo aparecen nuevas aplicaciones de software que pueden ejecutarse en sistemas ya existentes. Para que todo funcione correctamente el diseñador del nuevo software debe mantener un estándar en las interfaces de sus funciones que sea ya conocida y aceptada aunque la implementación de las funciones sea obviamente distinta. Nuevamente, esto es un ejemplo de cómo necesidades actuales en los sistemas pueden ser apoyado de mejor manera mediante nueva tecnología que ayude a mejorar los diseños aunque no garantiza el resultado final.

2.2.3 Lenguajes de Programación

Los lenguajes orientados a objetos varían en su apoyo a los conceptos de orientación a objetos y en los ambientes de desarrollo que incorporan. Por lo general, cada lenguaje, aunque orientado a objetos, tiene un diseño particular teniendo aspectos comunes entre si. El usuario debe considerar los distintos aspectos y tomar una decisión de cual es el lenguaje más apropiado para su aplicación. En general, el lenguaje que utilizaremos en este libro es Java por tres motivos principales: su integración con el Web, sus buenas características como lenguaje de programación y su gran aceptación en el mercado que lo hacen uno de los más utilizados en la actualidad.

No sería completa una descripción de la programación orientada a objetos sin mencionar algunos de los lenguajes de programación más importantes. Considerando que existen lenguajes de programación orientados a objetos ya desde hace varias décadas sería bueno revisar brevemente la historia de estos lenguajes, como se muestra en la Tabla 2.1, en orden cronológico. (Nótese que a partir de la década de los 80 la gran mayoría son orientados a objetos.)

Año	Lenguaje	Descripción	¿OO?
1957	FORTRAN	“FORmula TRANslator” fue el primer lenguaje de alto nivel y aún sigue	No

		siendo el más utilizado para cálculos numéricos. Fue diseñado originalmente por John Backus entre 1954 y 1957. La versión actual es FORTRAN-90.	
1959	LISP	Lisp fue diseñado por McCarthy entre 1956 y 1961. Existen diferentes extensiones, conocidas hoy en día como "CommonLisp". El lenguaje se utiliza principalmente para aplicaciones en Inteligencia Artificial. En un esfuerzo por hacer de LISP un lenguaje más moderno, éste se extendió en 1988 con orientación a objetos dando lugar a CLOS ("Common LISP Object System").	No
1959	COBOL	"COmputer Business Oriented Language" fue un lenguaje diseñado a partir de 1959 por un grupo de profesionales conocidos como CODASYL ("COnference on DAta SYstems Languages") y fue creado para aplicaciones principalmente financieras. Este lenguaje es el principal culpable del problema del milenio. La versión más reciente es COBOL-97 conteniendo incluso extensiones de orientación a objetos.	No
1960	ALGOL	"ALGOrithmic Language" fue desarrollado por J. Backus y P. Naur entre 1958 y 1960. Se le considera el primer lenguaje de propósito general para aplicaciones tanto industriales como científicas. La última versión fue Algol68.	No
1962	SIMULA	El primer sistema con objetos fue B1000 en 1961, seguido por Sketchpad en 1962, conteniendo "clones" e instancias. Sin embargo, se le atribuye como el primer lenguaje orientado a objetos conteniendo objetos y clases, a Simula I, diseñado por Ole Dahl y Kristen Nygaard del Centro de Computación de Noruega (NCC Oslo) en 1962. El lenguaje, implementado por primera vez en 1964, fue diseñado como una extensión a Algol 60 para la simulación de eventos discretos. En 1967, se introdujo el lenguaje de propósito más general Simula67 con un número mayor de tipos de datos además de apoyo a objetos. Simula se estandarizó en 1977.	Sí
1962	PL/I	"Programming Language 1" fue un lenguaje bastante complejo inventado en IBM a partir de 1962 para su famosos "System/360". PL/I quería ser "el" lenguaje para los sistemas grandes y aplicaciones. Fue utilizado principalmente en la década de los 80s.	No
1962	APL	"A Programming Language" fue diseñado por Ken Iverson a partir de 1962 y utilizado por IBM. El objetivo principal era programar matemáticamente. Incluía letras griegas, siendo un lenguaje extremadamente compacto. La versión actual es APL2.	No
1964	BASIC	Este famoso lenguaje fue inventado por los profesores John G. Kemeny y Thomas E. Kurtz de la Universidad de Dartmouth, Estados Unidos. El primer programa de BASIC fue ejecutado el 1 de Mayo de 1964. Los dialectos más modernos incluyen, a partir de 1991, VisualBasic diseñado por Microsoft (¡reminicencias de su primer negocio en 1975 vendiendo interpretadores de Basic!)	No
1968	Pascal	Este famosos lenguaje fue diseñado por Niklaus Wirth del Instituto Tecnológico Federal de Zurich entre 1968 y 1971. Pascal evolucionó el diseño de Algol siendo por muchos años "el" lenguaje para la enseñanza de la introducción a la programación en las diversas universidades. Las versiones más reconocidas posteriormente fueron promovidas por la compañía Borland con TurboPascal, y luego ObjectPascal en su ambiente de desarrollo Delphi, actualmente muy utilizado.	No
1972	Smalltalk	Smalltalk diseñado por Alan Kay (quien había diseñado y construido entre 1967 y 1968 la primera computadora personal basada en programación orientada a objetos, llamada FLEX) y otros en XEROX PARC. La primera versión fue conocida como Smalltalk 72, cuyas raíces fueron Simula 67. Siguió Smalltalk 76, una versión totalmente orientada a objetos. En 1980, Smalltalk 80, fue la primera versión comercial del lenguaje, incluyendo un	Sí

		ambiente de programación orientado a objetos uniforme. El lenguaje Smalltalk ha influido sobre otros lenguajes como C++ y Java, y aunque no ha tenido el grado de éxito de estos dos últimos, quizás por lo tardío en volverse gratis junto a razones de eficiencia, este lenguaje tiene un gran número de seguidores en la actualidad, los cuales consideran a Smalltalk como el "mejor" lenguaje que existe.	
1972	Prolog	"PROgramming in LOGic" fue el progenitor de la programación lógica. Fue diseñado por Robert A Kowalski de la Universidad de Edinburgo, Reino Unido, y Alain Colmerauer de la Universidad de Aix-Marseille, Francia.	No
1972	C	C fue diseñado por Ritchie y Thompson entre 1969 y 1973, en paralelo con los primeros desarrollos del sistema operativo Unix. Otra etapa del desarrollo fue hecha por Kernighan y Ritchie entre 1977 y 1979, cuando la portabilidad de Unix era demostrada. En esa época se escribió el famoso libro "*The C Programming Language*" [Kernighan y Ritchie, 1978]. Es uno de los lenguajes de mayor utilización en la actualidad.	No
1977	CLU	"CLUster" es un lenguaje diseñado por Barbara Liskov del MIT entre 1974 y 1977. El lenguaje utiliza conceptos básicos de la orientación a objetos aunque no es propiamente considerado como tal.	No
1980	Modula	La versión original se conoció como Modula-2 desarrollada por Niklaus Wirth diseñada a mediado de los 70s como descendiente director de Pascal. El lenguaje incluía concurrencia y ciertos aspectos de la orientación a objetos. La última versión conocida como Modula-3 fue diseñada por Luca Cardelli. Dada su simpleza se desconoce por qué la falta de éxito en la utilización de este lenguaje.	Sí
1983	Ada	El lenguaje fue diseñado a partir de 1977 por el Departamento de Defensa de Estados Unidos, teniendo como autor principal a Jean Ichibah, para apoyar programación de gran escala y promover la robustez del software. Su nombre fue en honor de Lady Ada Lovelace (1815-1852), una amiga y confidente de Charles Babbage, considerado el padre de la computación por su trabajo teórico hace un siglo y medio. Aunque la versión original no era orientada a objetos, la versión actual Ada 1995 sí lo es. Existe otra versión no orientada a objetos conocida como Ada 83 o Ada Clásica.	Sí
1983	Objective-C	El lenguaje fue diseñado por Brad Cox como una extensión a C pero con orientación a objetos. El lenguaje ofrecía muchos aspectos de diseño de Smalltalk-80 como su misma naturaleza sin tipos, aunque incorporaba datos sencillos de C, como enteros y reales. Su popularidad inicial vino a raíz de su utilización en la computadora NeXT, incluyendo una interfaz de construccion como parte del ambiente NeXTSTEP, conocido luego como OpenStep, y actualmente adquirido por Apple como base para su nuevo sistema operativo MacOS X.	Sí
1983	Beta	Beta, desarrollado por Madsen en la Universidad de Aarhus en Dinamarca es otro lenguaje orientado a objetos inspirado por Simula con sintaxis similar a Pascal y algo parecido a C.	Sí
1984	ML	"Standard" ML ("Meta Language") representa una familia de lenguajes funcionales propuestas originalmente en 1983 y diseñadas entre 1984 y 1988 por Milner y Tofte. La versión actual, "Standard ML '97" es una revisión modesta y simplificada del languaje.	No
1985	C++	C++ diseñado por Bjarne Stoustrup, AT&T Bell Labs, entre 1982 y 1985 es uno de los lenguajes de programación más populares actualmente. El lenguaje se agrega aspectos de orientación a objetos al lenguaje de C, siendo realmente un lenguaje híbrido donde un programador puede efectivamente programar en C aunque utilizando C++. En la actualidad muchos de los seguidores de este lenguaje se han pasado a Java. La razón primordial de esto es la complejidad de C++ junto con muchos aspectos	Sí

		problemáticos y falta de estandarización bajo distintas plataformas.	
1986	Eiffel	Eiffel, honrando a la famosa torre en París, fue diseñado por Bertrand Meyer como un lenguaje orientado a objetos con una sintaxis superficialmente similar a C. Eiffel ofrece un ambiente interpretado de "bytecode" similar a Java, aunque por eficiencia este código normalmente se traduce a C para luego ser compilado en el ambiente particular. El diseño del lenguaje apoya un enfoque de ingeniería de software conocido como "Diseño por Contrato". Aunque es un lenguaje muy sencillo y poderoso nunca logró la aceptación lograda por C++ y Java, posiblemente por la falta de compiladores gratis.	Sí
1986	Self	Self diseñado por David Ungar y Randall Smith es un lenguaje cuya sintaxis es similar a Smalltalk. Un aspecto muy novedoso es la omisión de la nocion de clase en el lenguaje, donde un objeto se deriva de otro (su prototipo) por medio de copiado y refinado. Dado esto, Self es un lenguaje muy poderoso, sin embargo, es aún un proyecto de investigación requiriendo una gran cantidad de memoria y una gran máquina para ejecutar.	Sí
1988	CLOS	CLOS ("Common LISP Object System") es una extensión de CommonLisp mediante la orientación a objetos desarrollada originalmente en 1988 por David Moon (Sumbolics), Daniel Bobrow (Xerox), Gregor Kiczales (Xerox) y Richard Gabriel (Lucid), entre otros.	Sí
1990	Haskell	Haskell desarrollado por un comité (Hughes, Wadler, Peterson y otros) tiene su nombre en honor a Haskell Brooks Curry, cuyo trabajo en lógica matemática sirve como fundamento para los lenguajes funcionales. El lenguaje está altamente influenciado por Lisp aunque fue extendido con ciertos aspectos de la orientación a objetos para ser más moderno.	Sí
1992	Dylan	Dylan ('DYnamic LANguage') es un lenguaje orientado a objetos originalmente desarrollado por Apple, se parece mucho a CLOS y Scheme, aunque ha sido influenciado por Smalltalk y Self.	Sí
1995	Java	Java, diseñado por Gosling en Sun Microsystems entre 1994 y 1995 es el lenguaje orientado a objetos más utilizado en la actualidad. El lenguaje es sencillo y portátil, bastante similar a C++, aunque tomando ideas de Modula-3, Smalltalk y Objective-C, haciéndolo más robusto y seguro. Java es típicamente compilado en "bytecodes" que son luego interpretados por una máquina virtual de Java (JVM). Un aspecto primordial en el éxito del lenguaje es su integración con el Web mediante aplicaciones conocidas como "applets" que pueden ser ejecutadas desde un navegador del Web ("browser"). Otro aspecto importante es la inclusión de un gran número de paquetes y librerías que estandarizan y facilitan el desarrollo de nuevos programas.	Sí
2000	C#	Este lenguaje conocido como "C Sharp" es el último intento por parte de Microsoft de competir contra el éxito y el seguimiento que tiene Java. El lenguaje revisa muchos aspectos problemáticos de C++.	Sí

Tabla 2.1 La tabla describe los lenguajes más importantes de la historia de la computación haciendo énfasis en aquellos orientados a objetos..