Tema 1

Programación clásica y Programación Orientada a Objetos

Índice de contenidos

1. Programación clásica y crisis de software

A finales de los años 60, en una conferencia organizada por la Comisión Científica de la OTAN se reconoció abiertamente que había un problema en el desarrollo de los sistemas software. Las aplicaciones que se construían, o bien no llegaban nunca a completarse con éxito, o bien, además de ser muy costosas, y de estar mal estructuradas eran prácticamente imposibles de mantener, como consecuencia la mayoría de las aplicaciones eran poco fiables. Esta situación fue denominada *crisis del software*.

Las causas de esta crisis tienen el origen en la **complejidad intrínseca de las aplicaciones software**, que deriva fundamentalmente de cuatro elementos que se pueden identificar en aplicaciones de mediana y gran envergadura:

- L**a complejidad de gestionar el proceso de desarrollo**: Normalmente el equipo de desarrollo no suele tener claro desde el primer momento qué se espera de la aplicación, ya que los usuarios no siempre consiguen transmitir sus necesidades y expectativas. Esto obliga a realizar multitud de cambios en los requisitos durante el desarrollo, lo cual dificulta considerablemente este proceso. Si la captura de requisitos fuera eficiente y el usuario los pudiera verificar mediante prototipos en fases tempranas del desarrollo, esta dificultad y el coste respectivo se reducirían.

- **La complejidad del dominio y de la solución** hace que haya que descomponer la aplicación en gran cantidad de módulos y, por lo tanto, que se requieran grandes equipos de desarrollo. Hay que tener en cuenta que hablamos de aplicaciones que modelan el mundo real. Esto comporta la dificultad de gestionar un equipo de desarrollo con muchos miembros, manteniendo una unidad e integridad del diseño de la aplicación.

- **Ausencia de estándares**: casi siempre que se construye una aplicación, se identifican partes que se podrían resolver de manera parecida a como se hizo en aplicaciones anteriores, pero la ausencia de estándares en la industria del software hace que aquello que ya se ha implementado sea difícilmente reutilizable.

- **Caracterizar el comportamiento de sistemas discretos es difícil.** Estos sistemas pueden tener un número muy grande de estados. Por este motivo, las pruebas no se pueden considerar nunca completas, ya que es prácticamente imposible simular todas las situaciones por las que puede pasar el sistema.

Esta complejidad, unida a la falta de una metodología a la hora de desarrollar las aplicaciones, hacía que los desarrollos basados en las estructuras de datos clásicas presentaran los siguientes **inconvenientes:**

- **Limitaciones en el modelado de problemas no estructurados**, ya que el método utilizado para construir las aplicaciones está basado en el hecho de descomponer la aplicación en una jerarquía de módulos funcionales ideados para transformar unas entradas determinadas en salidas bien definidas. Este tipo de

diseños no facilitan resolver problemas complejos desestructurados. El enfoque tradicional, se trata de un enfoque más apropiado para resolver problemas estructurados, en los que se puede describir el código según algoritmos de transformación de datos.

- **Dificultad a la hora de reutilizar el código**, ya que los módulos se caracterizaban por la transformación de unos datos en concreto, cosa que los hacía muy dependiente de la aplicación original.

- **Mantenimiento difícil y costoso**, ya que los módulos estaban muy orientados a tratar los datos y normalmente, durante su desarrollo, no se tenían en cuenta posibles cambios futuros.

- **Reducción de la calidad de las aplicaciones**, medida en flexibilidad, eficiencia, fiabilidad y robustez, criterios que trataremos en el módulo siguiente. La calidad se reduce a medida que se hacen modificaciones, ya que éstas complican cada vez más el diseño original, con lo cual se resta flexibilidad al desarrollo y cada vez resulta más complejo introducir nuevos cambios.

2. La orientación a objetos

La orientación a objetos se podría definir como el **conjunto de disciplinas que desarrollan y modelan sistemas software a partir de componentes (objetos)**, lo que facilita la construcción de sistemas complejos.

Podemos decir que la orientación a objetos es una nueva forma de resolver problemas de estructuración y representación más ajustada al mundo real y, por lo tanto, más próxima al proceso mental de las personas.

La orientación a objetos está basada **en tres aspectos organizativos próximos al razonamiento humano**, ya que establecen **conexiones** entre los siguientes elementos:

- **Un objeto y sus características**; por ejemplo, un coche tiene un color, un número de puertas, un tipo de motor, etc.

- **Un objeto y otros objetos con los que se asocia o interactúa** (que al mismo tiempo se relacionan con otros objetos); por ejemplo, el coche tiene ruedas (con unas características propias), un motor, etc.

- **Un objeto general y otros objetos que comparten sus mismas propiedades** pero más específicos; por ejemplo, un coche puede ser deportivo o familiar.

La **programación orientada a objetos es una técnica** para construir aplicaciones más antigua de lo que puede parecer. Si bien pasó desapercibida hasta principios de la década de los noventa, su **origen se remonta a 1967**, cuando dos noruegos, Ole-Johan Dahl y Kristen Nygaard, idearon los conceptos básicos de la programación orientada a objetos tal y como se conoce hoy.

El atractivo de la **orientación a objetos** es que **proporciona conceptos y herramientas con las cuales se modela, de forma a aproximada a como entendemos la realidad, es decir basada en objetos con propiedades.**

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 llevada a cabo de los requisitos del sistema (definidos en términos del usuario) a la especificación del sistema (definido en términos del la computadora) se reduce considerablemente.*

Teniendo en cuenta todo lo que hemos explicado hasta ahora, podemos decir que la orientación a objetos va más allá de la implementación de aplicaciones. Podemos decir que define toda una filosofía para analizar y diseñar sistemas dinámicos.

Antes de empezar a implementar cualquier problema, es necesario examinar qué requisitos existen y desde la perspectiva de la programación orientada a objetos usar clases y objetos que aproximen el dominio del problema. Una vez obtenido el modelo del problema, será necesario refinar y adaptar el modelo al lenguaje de programación que se utilizará, dado que no todos los lenguajes implementan los conceptos de la programación orientada a objetos de la misma manera.

A continuación, mostraremos un ejemplo que nos permitirá comprobar las diferencias entre el desarrollo orientado a objetos y el desarrollo procedimental.

Problema

Tenemos que hacer una aplicación para gestionar las nóminas de los profesores de una facultad y la dotación económica que recibe cada departamento.
En la facultad, cada profesor imparte docencia en una asignatura, cobra de acuerdo con las horas impartidas y pertenece a un departamento.
Según la carga docente, los departamentos tienen una dotación económica u otra. Para nuestra aplicación, necesitaremos saber los datos que modelan los departamentos (nombre, director, secretario, dirección postal, los profesores que hay asignados, etc.) y una operación que nos diga cuál es la dotación económica de los departamentos (para evaluarla, necesitaremos saber el número de horas de docencia de todos sus profesores).
De los profesores, aparte de los datos identificadores (el nombre y los apellidos, el despacho, el correo electrónico, etc.), necesitaremos poder consultar el nombre de las asignaturas que imparten y las horas de docencia que realizan en estas asignaturas.
De las asignaturas, habrá que saber el nombre y las horas de docencia.

Solución con el desarrollo clásico

Por una parte, tendríamos que definir las estructuras de datos que necesitaríamos en nuestra aplicación (departamento, profesor y asignatura) y, para cada una de éstas, deberíamos definir una estructura de datos para almacenarlas todas.
Después necesitaríamos una estructura de datos que nos permitiera representar las relaciones entre los departamentos y los profesores, y otra para representar la relación entre los profesores y las asignaturas.
Finalmente, tendríamos que implementar las operaciones que nos permitieran realizar las funcionalidades pedidas, y también otras operaciones que nos sirviesen para "navegar" entre las relaciones almacenadas.

Solución con el desarrollo orientado a objetos

En este caso, lo que tendríamos que hacer es identificar cada entidad (departamento, profesor y asignatura) y utilizar tres módulos diferentes en los que se definirían las propiedades de cada entidad y se implementarían las funcionalidades por separado.
Por ejemplo, en la operación del departamento de cálculo de las horas lectivas (para calcular la asignación), en vez de tener que navegar por toda la estructura de datos para saber si un profesor es o no del departamento, puesto que cada departamento ya está
relacionado con sus profesores, sólo haría falta pedirles las horas lectivas y sumarlas.

Una vez definidas estas entidades en nuestra aplicación de gestión, tendríamos tantos objetos de tipo `Profesor` como profesores haya, cada uno de los cuales con los datos que lo definen, y tantos objetos de tipo `Departamento` como departamentos haya en el sistema.
En este caso, no sería necesaria una estructura que representara las relaciones, ya que los objetos ya están relacionados entre sí.

A la vista del ejemplo anterior, podríamos decir que la orientación a objetos permite **agrupar las funcionalidades referentes a un mismo tipo de objeto en una misma entidad** y, por lo tanto, localizar posibles problemas es mucho más rápido y hacer modificaciones es más sencillo.

Algunos argumentos que podríamos esgrimir de forma rápida para usar la programación orientada a objetos son la mejora de la calidad de las aplicaciones, la escalabilidad y adaptabilidad de éstas, y también la disminución del tiempo y el coste de desarrollo. Gran parte de estas ventajas provienen de otra fundamental: el concepto de reutilización del código, que veremos en el apartado siguiente.

3. Reutilización del código

Cuando se construye un automóvil, o un dispositivo electrónico, se ensamblan una serie de piezas independientes, Existen piezas que se diseñan una vez y se reutilizan para distintos modelos, en vez de fabricarlos cada vez que se necesita construir un nuevo circuito u automóvil. En la construcción de software surge la cuestión. ***¿Por qué no se utilizan componentes ya construidos para formar parte de otros sistemas?***

Las técnicas orientadas a objetos proporcionan un mecanismo para construir *componentes de software reutilizables* que posteriormente podrán ser interconectados entre sí para formar grandes proyectos de software. Esta propiedad de poder utilizar elementos de software programados y probados previamente durante la construcción de aplicaciones nuevas se denomina **reutilización** o **reusabilidad**.

Podemos resumir las **ventajas** principales que se obtienen de reutilizar un código de la siguiente forma:

- **Disminución del esfuerzo de mantenimiento**. Ya que se utilizan componentes previamente probados en otros sistemas. Además puesto que todo el código que afecta a una funcionalidad concreta está en un mismo módulo, las tareas de mantenimiento se basan en modificaciones específicas y lo que es má importante están acotadas.

- **Mayor velocidad en el desarrollo de aplicaciones**, favorecida principalmente por el hecho de poder reutilizar componentes totalmente funcionales .

- **Aumento de la fiabilidad de los programas**. El módulo reutilizable habrá superado pruebas de funcionamiento previas, con lo que su fiabilidad habrá sido comprobada.

- **Abaratamiento de costes** favorecido por el hecho de no tener querealizar nuevos desarrollos, ya que se pueden aprovechar de proyectos anteriores.

Para ejemplarizar mejor el concepto de reutilización de código, podemos hablar de las **interfaces gráficas de usuario.** Esta colección de elementos que utilizamos cuando desarrollamos aplicaciones gráficas (menús, botones, cuadros de texto, etc.) no los programamos cada vez que realizamos un proyecto, sino que los tomamos de un repositorio que los creadores del sistema operativo o el lenguaje de programación nos ofrecen. Si los tuviéramos que implementar nosotros, aparte de que deberíamos tener un gran conocimiento sobre este tema, necesitaríamos mucho más tiempo, ya que estos elementos los utilizamos muchas veces en nuestro programa.

El desarrollo de los elementos de las interfaces gráficas ha sido realizado por un grupo de programadores una única vez, y todos nosotros sacamos provecho del mismo. Por lo tanto, podemos decir que, en este caso, aprovechamos los siguientes aspectos de la programación orientada a objetos:

- **Eficiencia** de nuestra aplicación para tratar la interfaz gráfica, ya que suponemos que estos elementos son lo más eficientes posible.

- **Fiabilidad** de la interfaz gráfica, ya que estos elementos están muy probados y, por lo tanto, no presentarán un comportamiento inesperado.

- **Consistencia** de la interfaz de usuario, dado que todos los elementos tienen unas características parecidas.

- **Más velocidad de desarrollo y menos costes**. Como hemos comentado, estos elementos ya nos vienen dados y, por lo tanto, ahorramos tiempo y dinero.

- **Eliminación de los costes de mantenimiento**, ya que estos elementos, puesto que ya están hechos, no necesitan –ni pueden– ser modificados.

4. Programación y abstracción

El proceso de abstracción es común a todas las ciencias, y como vemos forma parte del proceso de razonamiento humano.Las personas normalmente comprenden el mundo construyendo modelos de partes del mismo: un modelo mental es una vista simplificada de cómo funcionan las cosas. Los modelos abstraen las características básicas o esenciales para un propósito particular y desechan las irrelevantes. A este proceso se denomina **abstracción**.

Veamos la definición de abstracción del Diccionario de la Real Academia Española (D.R.A.E).

abstracción.

(Del lat. *abstractĭo, -ōnis*).

1. f. Acción y efecto de abstraer o abstraerse.

abstraer.

(Del lat. abstrahĕre).

1. tr. Separar por medio de una operación intelectual las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción.

2. intr. Prescindir, hacer caso omiso. Abstraer DE examinar la naturaleza de las cosas. U. t. c. prnl.

La **abstracción** es esencial para el funcionamiento de la mente humana normal y es una herramienta muy potente para tratar la complejidad.

En general **un programa** de ordenador no **es** más que **una descripción abstracta** de un procedimiento o fenómeno que sucede en el mundo real. 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 leguaje de programación para describir de un modo abstracto el comportamiento físico de la computadora que se está utilizando (Ej usando números decimales en lugar de números binarios, variables en lugar de celdas de memoria direccionadas directamente).

Como describe **Wulf:** << *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>>.

Algunas reglas interesantes para realizar abstracciones en programación orienta a objetos son las siguientes:

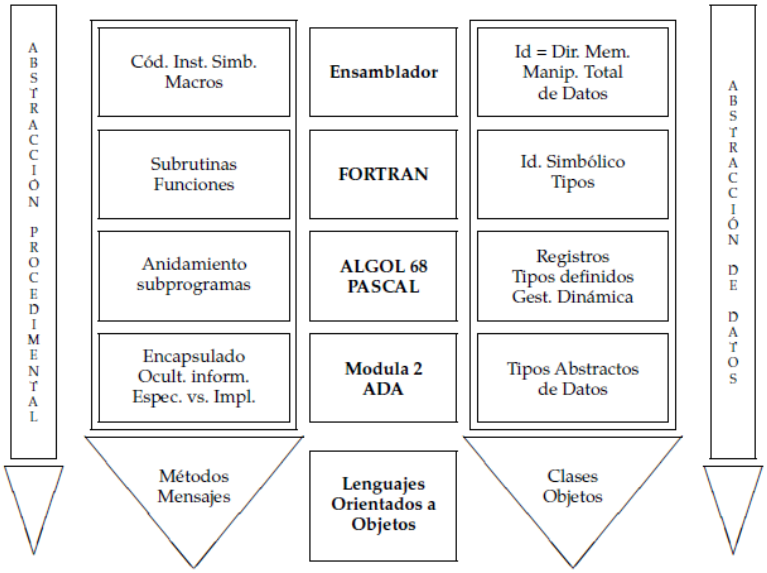
- En primer lugar se debe identificar los objetos principales.
- Tener en cuenta que la abstracción es un proceso iterativo de ensayo-y-error, con aproximaciones y refinamientos sucesivos.
- Posteriormente se debe definir propiedades y métodos esenciales de cada objeto. En definitiva características relevantes o esenciales en el dominio del problema.
- Si existen objetos semejantes ,clasificar los objetos según sus semejanzas/diferencias
- Por último se debe establecer cómo se relacionan entre sí, cómo se comunican (eventos).

En la **década de los cincuenta**, uno de los pocos mecanismos de abstracción para ordenadores era el **lenguaje ensamblador y lenguaje máquina**. Posteriormente los lenguajes de programación de alto nivel ofrecieron un nuevo nivel de abstracción.

Los diferentes paradigmas de programación han aumentado su nivel de abstracción, comenzando desde los lenguajes máquina, los más próximos al ordenador y más lejanos a la comprensión humana; pasando por los lenguajes de comandos, los imperativos, la orientación a objetos (OO).

La abstracción ofrecida por los lenguajes de programación se puede dividir en dos categorías: **abstracción de datos** (pertenecientes a los datos) y **abstracción de control** (perteneciente a las estructuras de control)

El siguiente diagrama se puede observar la evolución de los lenguajes de programación.



Existe como vemos una evolución constante en el nivel de abstracción apreciable a lo largo de la historia y evolución de los lenguajes de programación.

5. Lenguajes de programación orientados a objetos

El primer lenguaje orientado a objetos fue SIMULA 67. Fue creado por científicos noruegos (Nygaard y Gahl) que, después de probar otros lenguajes de tercera generación, decidieron crear un lenguaje que les permitiera realizar todo lo que los otros no les permitían a causa de las limitaciones que tenían. SIMULA 67 fue el primer lenguaje de programación en el que se introdujeron los conceptos de **clase** y **objeto.**

Más tarde, a principios de los años setenta, en los laboratorios de Xerox, un equipo de desarrollo intentó implementar un prototipo de ordenador para niños, conocido como **DynaBook**, que utilizaba por primera vez el concepto d**e interfaz gráfica de usuario**. Enseguida los desarrolladores relacionaron las ideas de la programación orientada a objetos con las piezas del ordenador nuevo, ya que se podía definir perfectamente el funcionamiento de cada elemento gráfico como un objeto, porque tenía un comportamiento muy definido y respondía a interacciones muy concretas.

Para implementar el proyecto DynaBook, elaboraron un lenguaje de programación nuevo denominado **Smalltalk**, que estaba fuertemente influido por SIMULA 67. Este lenguaje **introducía el concepto de herencia de manera explícita**. **La herencia** es el mecanismo que permite que un objeto comparta las características y el comportamiento de otro objeto, del cual se dice que hereda.

El concepto de herencia es muy valioso para reutilizar código, ya que determinados objetos, aunque no son idénticos, sí que tienen características comunes. Por ejemplo, toda persona tiene un nombre y una fecha de nacimiento. Estas propiedades se utilizarán tanto si dentro de nuestra aplicación la persona es una estudiante como si es un profesor. En próximos temas, lo estudiaremos más detalladamente.

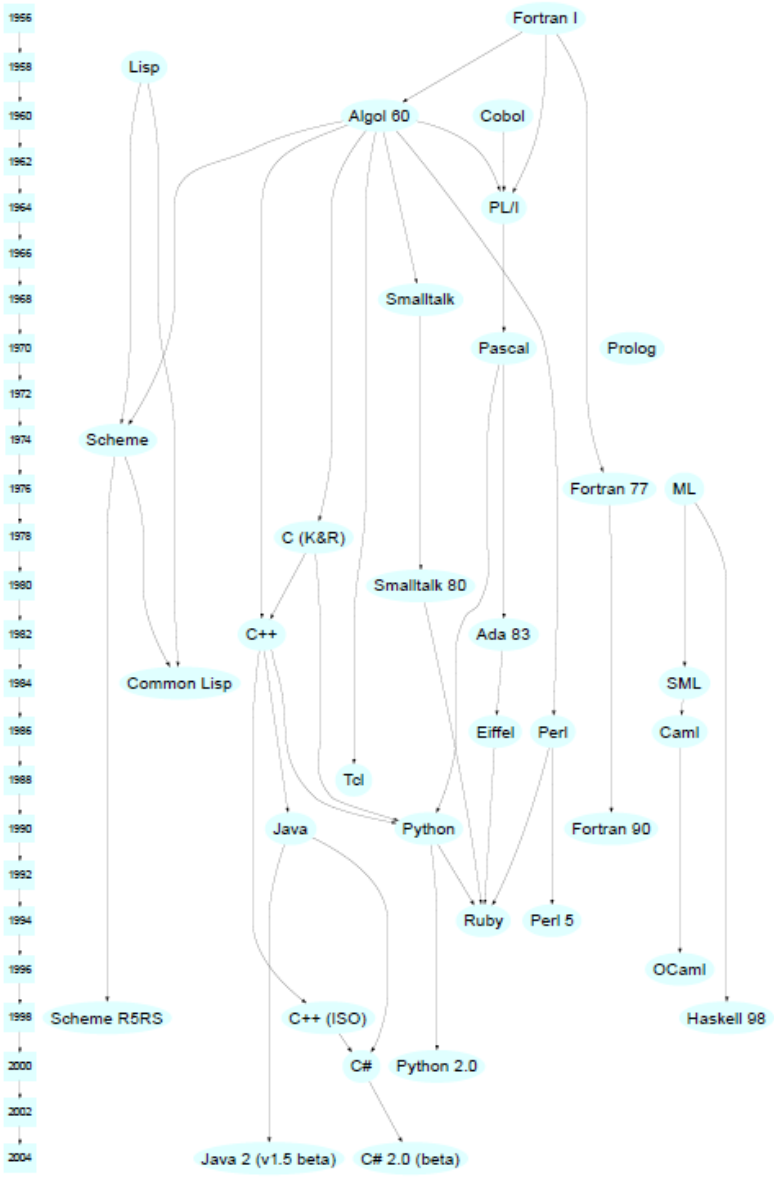
A partir de este momento, los lenguajes de programación orientados a objetos van incorporando características nuevas que los hacen evolucionar hasta los lenguajes de programación que conocemos hoy en día.

En la década de los ochenta, **Bjarne Stroustrup**, empleado de AT&T Labs., amplió el lenguaje de programación C para adaptarlo a la programación orientada a objetos. Inicialmente se denominó **C with classes**, pero finalmente se llamó **C++**. Este lenguaje está basado en el SIMULA 67, con respecto a la orientación a objetos, y en el C, en cuanto a la sintaxis.

Desde que aparecieron los lenguajes orientados a objetos, éstos han ido aportando funcionalidades nuevas a la programación orientada a objetos en versiones sucesivas, como las clases abstractas, los métodos static o la genericidad. Veremos todos estos conceptos en los próximos módulos.

Actualmente hay muchos lenguajes de programación orientada a objetos, unos descendientes de lenguajes procedimentales que se han adaptado al paradigma de la programación orientada a objetos (POO) y otros que se han creado desde el principio pensando en este paradigma. Por poner algunos ejemplos, aparte de C++, SIMULA 67 y Smalltalk, que ya hemos mencionado antes, podemos mencionar Perl 5, Python, Java, C#, Delphi, Eiffel, etc.

El siguiente gráfico muestra la evolución y la relación existente entre distintos lenguajes de programación.



5.1.Clasificación de los lenguajes de programación

Los lenguajes de programación han tenido grandes transformaciones durante su historia. Hay muchas clasificaciones que, según diferentes criterios, así lo muestran. Una de las más conocidas es la clasificación de **Wegner**, que **divide los lenguajes de programación** de alto nivel **de acuerdo con el orden de aparición y las características comunes**. Concretamente, la clasificación consta de lenguajes de primera generación, de segunda generación, de tercera generación y la generación de los lenguajes actuales.

5.1.1. Lenguajes de primera generación

Lenguajes que se desarrollaron entre 1954 y 1958. Con éstos se produjo un salto en la abstracción a la hora de programar, ya que anteriormente se utilizaba el lenguaje ensamblador, que requería un buen conocimiento de la máquina en la que se ejecutaría el código. Algunos ejemplos de estos lenguajes son **Fortran I, Algol 58 o IPL V**: todos estaban basados en codificar de la mejor manera expresiones matemáticas y, por lo tanto, su dominio de aplicación estaba centrado en el cálculo y las aplicaciones científicas y de ingeniería.

Los programas que se hacían con estos lenguajes se basaban en subprogramas que compartían los datos al ser ejecutados.

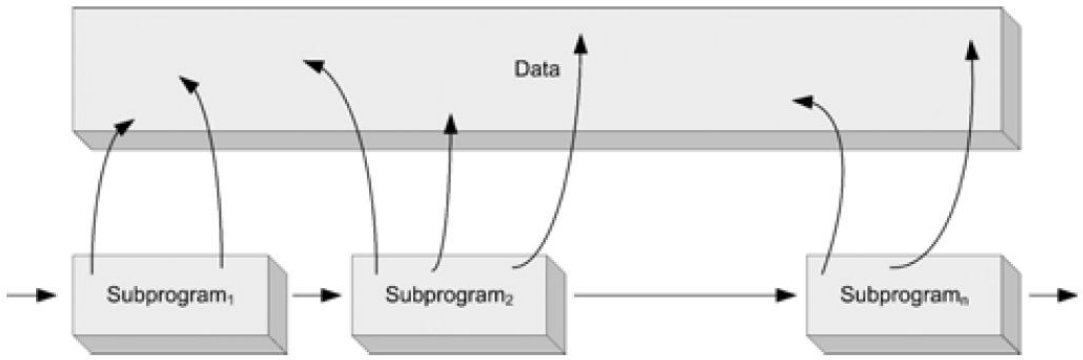


Figura 1. Lenguajes de primera generación

Esta situación presentaba problemas serios porque cualquier error en el funcionamiento de un subprograma se propagaba al resto de la aplicación. Aparte, de que cuando el programa era demasiado grande, cualquier cambio se traducía en una tarea muy costosa que no siempre se conseguía llevar a cabo sin complicar todavía más el diseño original.

5.1.2. Lenguajes de segunda generación

Aparecieron a finales de la década de los cincuenta y a principios de los sesenta. En estos lenguajes, se empezó a poner énfasis en la **abstracción algorítmica** y la programación se extendió poco a poco a otros dominios del mundo real. Algunos de los lenguajes pertenecientes a esta generación son Cobol, Algol 60 o Fortran II. En el

terreno de la estructuración de los programas, la segunda generación **empieza a introducir el concepto de procedimientos** dentro de los subprogramas, con lo cual **aparecen** los conceptos de **paso de parámetros** y de **visibilidad de las variables.**

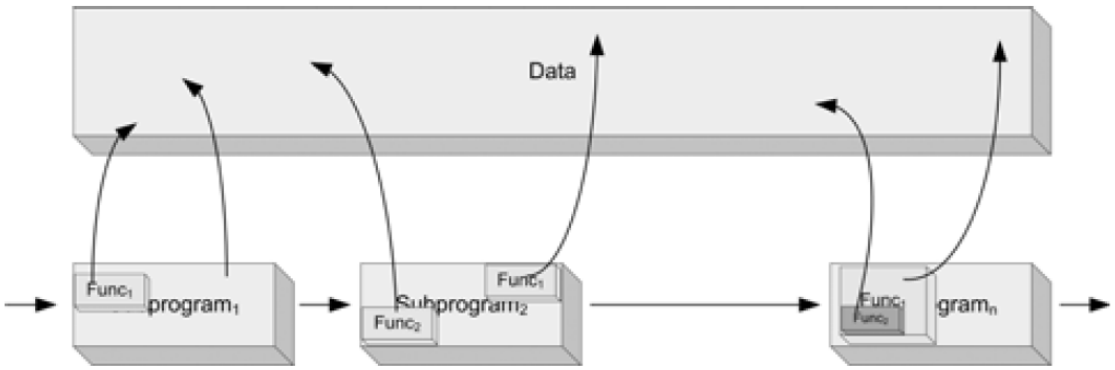


Figura 2. Lenguajes de segunda generación

5.1.3. Lenguajes de tercera generación

En esta época, los costes del hardware continúan abaratándose, lo cual hace que las aplicaciones informáticas lleguen cada vez más a ámbitos más distintos. Esta diversificación en los campos de trabajo comporta que se tenga que trabajar con **tipos de datos** diferentes de los puramente matemáticos, y cambiantes de una aplicación a otra. Por este motivo, en esta época **aparece** el concepto de **tipo abstracto de datos** que permite al programador especificar un tipo adecuado para cada problema y dotarlo de significado mediante un conjunto de operaciones sobre este tipo de dato.

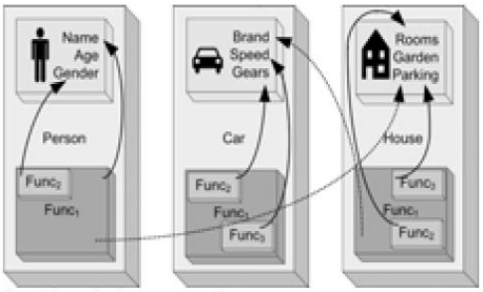


Figura 3. Lenguajes de tercera generación

Este nuevo concepto resuelve algunos problemas de la programación de aplicaciones grandes, ya que de esta manera distintos desarrolladores pueden implementar módulos diferentes que se pueden compilar por separado y, finalmente, combinarlos. A pesar de todo, durante esta época el concepto de abstracción de los datos no fue demasiado utilizado como tal, sino que simplemente permitía agrupar distintas funcionalidades en módulos diferentes. Otro concepto que no existía era el de la **visibilidad de los datos**. Por lo tanto, nada impedía que diferentes módulos modificasen directamente los datos de otros módulos.

5.1.4. Lenguajes de cuarta generación

Estos son lenguajes de programación que no son de propósito general, como los descritos anteriormente. Han sido diseñados para algún propósito específico. Entre

otros, están Sculptor (lenguaje de dirigido por dominio) o lenguajes de especificación como SQL.

Aunque no existe consenso sobre lo que es un *lenguaje de cuarta generación* (4GL).

Una característica relevante podría ser el hecho de necesitar menos líneas de código debido a su poder de abstracción, y son lenguajes en los que no se indican detalles de implementación, simplemente se especifica lo que se desea obtener , dejando al interprete el implementar como hacerlo

Los 4GL se apoyan en unas herramientas de más alto nivel de abstracción denominadas herramientas de cuarta generación. A groso modo los 4GL podrían abarcar:

- Lenguajes de presentación, como lenguajes de consultas y generadores de informes.
- Lenguajes especializados, como hojas de cálculo y lenguajes de bases de datos.
- Generadores de aplicaciones que definen, insertan, actualizan y obtienen datos de la base de datos.
- Lenguajes de muy alto nivel que se utilizan para generar el código de la aplicación.

6. Lenguajes Orientados a Objetos

Se denominan lenguajes orientados a objetos todos aquellos lenguajes de programación que exhiben en mayor o menor medida una serie de características que veremos a continuación, pero **su rasgo esencial es que utilizan el concepto de clase de objetos como mecanismo de representación de abstracciones y de organización del código.**

6.1. Características básicas de los lenguajes de programación orientados a objetos

- **Tipificación estricta de datos**

Tipificar es el proceso de declarar el tipo de información que puede contener una variable. Este hecho implica que si dos expresiones están relacionadas, tanto si se trata de una asignación como si se trata de cualquiera de las operaciones que se pueden hacer sobre expresiones, tienen que coincidir en tipo. Si no lo hacen, el compilador genera un error en tiempo de compilación.

- **Encapsulamiento**

Es un **mecanismo** que permite **agrupar** datos y operaciones relacionadas en una misma entidad (clase). Esta propiedad facilita que aparezcan otras características de la programación orientada a objetos como la reutilización.

El encapsulamiento está relacionado con la ocultación de la información y con la separación de una implementación y su especificación. Proporcionando un acoplamiento más débil.

De esta manera, y debido a la ocultación de la información, los usuarios de una clase disponen de unos métodos que permiten consultar y modificar el comportamiento de la clase, pero no tienen acceso directo a los datos internos.

Para aclarar los conceptos de encapsulamiento y ocultación, podemos pensar en la clase *Date(existente en Java)*. Una fecha se puede descomponer en día, mes y año, y puede tener unos métodos para acceder al día, al mes y al año y otros para modificar la fecha que almacena.

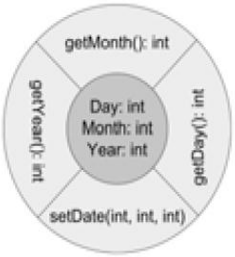


Figura 4. Representación de la clase Date

Cualquier objeto que intente consultar o establecer la fecha de este objeto, lo tendrá que hacer mediante las operaciones destinadas a esta finalidad y no podrá acceder directamente a los atributos que representan el día, el mes y el año.

La ventaja de esta situación es que si en cualquier momento tenemos que cambiar la representación interna de los datos almacenados, no será necesario modificar ninguna de las aplicaciones que utilizan objetos de tipo Date, ya que, a pesar de cambiarse su estructura interna, no se modifica su interfaz.

Otra de las ventajas del encapsulamiento es la posibilidad de ocultar operaciones internas de la clase que no deben ser visibles por objetos externos a ésta. Por ejemplo, podríamos tener un método que, dada una fecha, comprobase si ésta es correcta y que sólo sea accesible internamente.

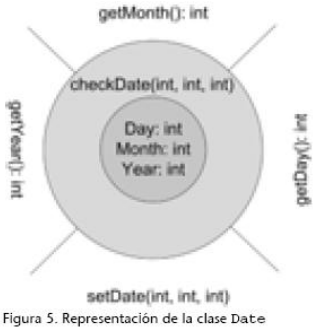


Figura 5. Representación de la clase Date

Esta operación oculta puede servir para asegurarnos de que no insertamos fechas erróneas o para realizar cualquier otra operación que se utiliza internamente pero que no se quiere que se ejecute desde otras partes del programa.

- **Genericidad**

Esta propiedad permite la definición de módulos de programa (clases) con un alto grado de reusabilidad. La genericidad es básica para reutilizar código. Estos módulos se diseñan con parámetros formales genéricos, qué se instanciarán posteriormente con parámetros reales específicos. Esta permite definir métodos que tienen como parámetros elementos de cualquier tipo.

Un ejemplo de genericidad lo tendríamos si quisiéramos definir un objeto que fuera un vector o una lista que pudiera recibir elementos de cualquier tipo. Esto tiene sentido porque el comportamiento de este objeto siempre es el mismo (obtener, borrar, insertar, etc.), independientemente del tipo u objeto contenido. De esta manera, lo podríamos usar una vez como vector de enteros, otra como vector de caracteres, etc.

- **Herencia**

Propiedad que nos permite definir una clase según otra u otras de manera que la clase que hereda tenga el mismo comportamiento y las mismas características que la clase de la cual hereda, más las características y el comportamiento de la propia clase.

Por ejemplo, podríamos definir una clase Persona con los atributos y métodos correspondientes y dos clases más, Estudiante y Profesor, que hereden de la clase Persona. Las tres clases tendrían una parte común, y las clases Estudiante y Profesor añadirían otros métodos y atributos específicos para cada una de éstas.

- **Polimorfismo**

El polimorfismo está estrechamente vinculado con la herencia. Se puede definir como la propiedad por la cual se pueden realizar tareas diferentes invocando la misma operación, y esta se realizará según el tipo de objeto sobre el cual se invoca.

Continuamos con el ejemplo anterior de los Estudiantes y los Profesores. Si tuviéramos un método denominado *obtenerCreditos*, en el caso de un estudiante, debería devolver el número de créditos de los que éste está matriculado, y en el caso de un profesor, nos tendría que devolver el número de créditos de las asignaturas que imparte. Las tareas necesarias para devolver el resultado según el tipo de objeto serían diferentes.

Éstas son las características principales de los lenguajes de programación orientada a objetos, aunque hay lenguajes que incorporan otras características propias adicionales, como la gestión de objetos en memoria (de C# y Java) o la gestión de excepciones.

7. Bibliografía

- Object -Oriented Construction 2nd .Bertrand Meyers .ISE California
- Java 2.Curso de programación. Fco Javier Ceballos.Ra-Ma

- Programación orientada a objetos. Joan Arnedo Moreno. Daniel Riera I Terrén.www,uoc.edu
- Programming Languages.1976 Wegner
- Programación Orientada a Objetos .Luis Joyanes Aguilar.Mc Graw Hill