# 2 Programación y Lenguajes de Programación

En la actualidad, la mayoría de nosotros utilizamos computadoras permanentemente: para mandar correos electrónicos, navegar por Internet, chatear, jugar, escribir textos. Las computadoras se usan para actividades tan disímiles como predecir las condiciones meteorológicas de la próxima semana, guardar historias clínicas, diseñar aviones, llevar la contabilidad de las empresas o controlar una fábrica. Y lo interesante aquí (y lo que hace apasionante a la programación) es que el mismo aparato sirve para realizar todas estas actividades: uno no cambia de computadora cuando se cansa de chatear y quiere jugar al solitario.

Muchos definen una computadora moderna como "una máquina que almacena y manipula información bajo el control de un programa que puede cambiar". Aparecen acá dos conceptos que son claves: por un lado se habla de una máquina que almacena información, y por el otro lado, esta máquina está controlada por un programa que puede cambiar.

Una calculadora sencilla, de esas que sólo tienen 10 teclas para los dígitos, una tecla para cada una de las 4 operaciones, un signo igual, encendido y limpiar, también es una máquina que almacena información y que está controlada por un programa. Pero lo que diferencia a esta calculadora de una computadora es que en la calculadora el programa no puede cambiar.

Un programa de computadora es un conjunto de instrucciones paso a paso que le indican a una computadora cómo realizar una tarea dada, y en cada momento uno puede elegir ejecutar un programa de acuerdo a la tarea que quiere realizar.

Las instrucciones se deben escribir en un lenguaje que nuestra computadora entienda. Los lenguajes de programación son lenguajes diseñados especialmente para dar órdenes a una computadora, de manera exacta y no ambigua. Sería muy agradable poder darle las órdenes a la computadora en castellano, pero el problema del castellano, y de las lenguas habladas en general, es su ambigüedad.

**El Mito de la Máquina Todopoderosa** Muchas veces la gente se imagina que con la computadora se puede hacer cualquier cosa, que no hay tareas imposibles de realizar. Más aún, se imaginan que si bien hubo cosas que eran imposibles de realizar hace 50 años, ya no lo son más, o no lo serán dentro de algunos años, cuando las computadoras crezcan en poder (memoria, velocidad, unidades de almacenamiento), y la computadora se

vuelva una máquina todopoderosa.

Sin embargo eso no es así: existen algunos problemas, llamados no computables que nunca podrán ser resueltos por una computadora digital, por más poderosa que ésta sea. La computabilidad es la rama de la computación que se ocupa de estudiar qué tareas son computables y qué tareas no lo son. De la mano del mito anterior, viene el mito del lenguaje todopoderoso: hay problemas que son no computables porque en realidad se utiliza algún lenguaje que no es el apropiado.

En realidad todas las computadoras pueden resolver los mismos problemas, y eso es independiente del lenguaje de programación que se use. Las soluciones a los problemas computables se pueden escribir en cualquier lenguaje de programación. Eso no significa que no haya lenguajes más adecuados que otros para la resolución de determinados problemas, pero la adecuación está relacionada con temas tales como la elegancia, la velocidad, la facilidad para describir un problema de manera simple, etc., nunca con la capacidad de resolución.

Los problemas no computables no son los únicos escollos que se le presentan a la computación. Hay otros problemas que si bien son computables demandan para su resolución un esfuerzo enorme en tiempo y en memoria. Estos problemas se llaman intratables. El análisis de algoritmos se ocupa de separar los problemas tratables de los intratables, encontrar la solución más barata para resolver un problema dado, y en el caso de los intratables, resolverlos de manera aproximada: no encontramos la verdadera solución porque no nos alcanzan los recursos para eso, pero encontramos una solución bastante buena y que nos insume muchos menos recursos (el orden de las respuestas de Google a una búsqueda es un buen ejemplo de una solución aproximada pero no necesariamente óptima).

En este trabajo veremos problemas no sólo computables sino también tratables. Y aprenderemos a medir los recursos que nos demanda una solución, y empezaremos a buscar la solución menos demandante en cada caso particular. Por ejemplo:

> Dado un número N se quiere calcular N 33. Una solución posible, por supuesto, es hacer el producto N×N×. . .×N , que involucra 32 multiplicaciones. Otra solución, mucho más eficiente es: Calcular N×N . Al resultado anterior multiplicarlo por sí mismo con lo cual ya disponemos de $N^4$. Al resultado anterior multiplicarlo por sí mismo con lo cual ya disponemos de $N^8$.

> Al resultado anterior multiplicarlo por sí mismo con lo cual ya disponemos de $N^{16}$. Al resultado anterior multiplicarlo por sí mismo con lo cual ya disponemos de $N^{32}$. Al resultado anterior multiplicarlo por N con lo cual conseguimos el resultado deseado con sólo 6 multiplicaciones. Cada una de estas soluciones representa un algoritmo, es decir un método de cálculo, diferente. Para un mismo problema puede haber algoritmos diferentes que lo resuelven, cada uno con un costo distinto en términos de recursos computacionales involucrados.

Hay muchas aplicaciones a las herramientas computacionales que se han programado y podemos programar, pero nos interesan especialmente aquellas que permitan resolver problemas concomitantes en Ciencia e Ingeniería. Muchas de estas aplicaciones caen en lo que comúnmente se llama cómputo científico. La computación científica es el campo de estudio relacionado con la construcción de modelos matemáticos, técnicas numéricas para resolver problemas científicos y de ingeniería; y su respectiva implementación computacional.

Este campo es distinto a las ciencias de la computación y el procesamiento de información, también es diferente a la teoría y experimentación, que son las formas tradicionales de la ciencia y la ingeniería. El enfoque de la computación científica es para ganar entendimiento, principalmente a través del análisis de modelos matemáticos implementados en computadoras.

Los programas de aplicación de la computación científica a menudo modelan cambios en las condiciones del mundo real, tales como el tiempo atmosférico, el flujo de aire alrededor de un avión, el movimiento de las estrellas en una galaxia, el comportamiento de un dispositivo explosivo, entre otros. Estos programas deberían crear una 'malla lógica' en la memoria de la computadora, donde cada ítem corresponda a un área en el espacio y contenga información acerca del espacio relevante para el modelo. Por ejemplo, en modelos para el tiempo atmosférico, cada ítem podría ser un kilómetro cuadrado, con la altitud del suelo, dirección actual del viento, humedad ambiental, temperatura, presión, etc. El programa debería calcular el siguiente estado probable basado en el estado actual, simulado en medidas de tiempo, resolviendo ecuaciones que describen cómo operan los sistemas mediante el uso de un algoritmo[3], y repetir el proceso para calcular el siguiente estado. Este código o programa se escribe en un lenguaje de programación que sigue

[3] Un algoritmo es un conjunto preescrito de instrucciones o reglas bien definidas, orde-

algún paradigma de programación[4], que posteriormente puede ser ejecutado por una unidad central de procesamiento -computadora-.

Así, una parte importante de la programación es el hecho de conocer y usar uno o más paradigmas de programación. Entonces iniciemos delineando lo que es un paradigma de programación. Los paradigmas difieren unos de otros, en los conceptos y la forma de abstraer los elementos involucrados en un problema, así como en los pasos que integran la solución del problema, en otras palabras, el cómputo. Tiene una estrecha relación con la formalización de determinados lenguajes al momento de definirlos -es el estilo de programación empleado-.

## 2.1 Paradigmas de Programación

Los Tipos más comunes de paradigmas de programación son[5]:

- Programación imperativa o por procedimientos: es el más usado en general, se basa en dar instrucciones a la computadora de como hacer las cosas en forma de algoritmos. La programación imperativa es la más usada y la más antigua, el ejemplo principal es el lenguaje de máquina. Ejemplos de lenguajes puros de este paradigma serían C, BASIC o Pascal.

- Programación orientada a objetos: esta basado en el imperativo, pero encapsula elementos denominados objetos que incluyen tanto variables como funciones. Esta representado por C++, C#, Java o Python entre otros, pero el más representativo sería el Smalltalk que esta completamente orientado a objetos.

- Programación dinámica: esta definido como el proceso de romper problemas en partes pequeñas para analizarlos y resolverlos de forma lo más cercana al óptimo, busca resolver problemas en $O(n)$ sin usar por tanto métodos recursivos. Este paradigma está más basado en el

nadas y finitas que permiten llevar a cabo una actividad mediante pasos sucesivos que no generen dudas a quien deba hacer dicha actividad. Dados un estado inicial y una entrada, siguiendo los pasos sucesivos se llega a un estado final y se obtiene una solución.

[4]Este representa un enfoque particular o filosofía para diseñar soluciones e implementarlas en algún lenguaje de programación.

[5]En general la mayoría son variantes de los dos tipos principales: imperativa y declarativa.

modo de realizar los algoritmos, por lo que se puede usar con cualquier lenguaje imperativo como C, C++, Java o Python.

- Programación dirigida por eventos: la programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos diseñen, por ejemplo en las interfaces gráficas de usuarios o en la Web.

- Programación declarativa: esta basado en describir el problema declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones. Hay lenguajes para la programación funcional, la programación lógica, o la combinación lógico-funcional. Unos de los primeros lenguajes funcionales fueron Lisp y Prolog.

- Programación funcional: basada en la definición los predicados y es de corte más matemático, está representado por Scheme (una variante de Lisp) o Haskell. Python también representa este paradigma.

- Programación lógica: basado en la definición de relaciones lógicas, está representado por Prolog.

- Programación con restricciones: similar a la lógica usando ecuaciones. Casi todos los lenguajes son variantes del Prolog.

- Programación multiparadigma: es el uso de dos o más paradigmas dentro de un programa. El lenguaje Lisp se considera multiparadigma. Al igual que Python, que es orientado a objetos, reflexivo, imperativo y funcional.

- Programación reactiva: este paradigma se basa en la declaración de una serie de objetos emisores de eventos asíncronos y otra serie de objetos que se «suscriben» a los primeros -es decir, quedan a la escucha de la emisión de eventos de estos- y "reaccionan" a los valores que reciben. Es muy común usar la librería Rx de Microsoft (Acrónimo de Reactive Extensions), disponible para múltiples lenguajes de programación.

El paradigma de programación más utilizado en la actualidad, es el de «orientación a objetos». El núcleo central de este paradigma es la unión de

datos y procesamiento en una entidad llamada «objeto», relacionable a su vez con otras entidades «objeto».

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

Otra parte importante de la programación tiene que ver con el algoritmo a implementar y las estructuras de datos necesarias para soportar los datos del problema, así como su eficiencia. Entonces entenderemos por eficiencia algorítmica para describir aquellas propiedades de los algoritmos que están relacionadas con la cantidad de recursos utilizados por el algoritmo. Un algoritmo debe ser analizado para determinar el uso de los recursos que realiza. La eficiencia algorítmica puede ser vista como análogo a la ingeniería de productividad de un proceso repetitivo o continuo. Con el objetivo de lograr una eficiencia máxima se quiere minimizar el uso de recursos. Sin embargo, varias medidas (e.g. complejidad temporal, complejidad espacial) no pueden ser comparadas directamente, luego, cual de los algoritmos es considerado más eficiente, depende de cual medida de eficiencia se está considerando como prioridad, e.g. la prioridad podría ser obtener la salida del algoritmo lo más rápido posible, o que minimice el uso de la memoria, o alguna otra medida particular.

Entonces, ¿cómo determinar si un algoritmo es mejor que otro?. Algunas pautas pueden ser:

- Facilidad de implementar
- Facilidad de entender
- Facilidad de modificar
- Usa menos memoria
- Menor tiempo de ejecución

**Tomando en cuenta todo lo anterior, ¿cómo se hace un programa?**

- Hay que tener claro el problema a resolver (*formalización*)

- Se debe planear cómo se quiere resolver o abordar el problema (*análisis y diseño*)
- Se elige uno o más lenguajes de programación y se escriben las instrucciones en ese lenguaje para llevar a cabo esa tarea (*codificación*)
- El texto se compila o interpreta para detectar errores sintácticos y semánticos (*depuración*)
- El programa revisado se prueba con los distintos datos de entrada (*ejecución*)
- Se evalúan los resultados, y de ser necesario se regresa a cualquiera de los pasos anteriores para completar el proceso de corrección (*validación*)

Así, para dar solución a algún problema de nuestro interés, debemos elegir el mejor algoritmo a nuestra disposición, seleccionar[6] el paradigma de programación que nos ofrezca ventajas según su eficiencia algorítmica y con ello usar el lenguaje que nos permita implementar dicho programa usando las técnicas de análisis y diseño que garanticen la calidad de un producto de Software[7].

## 2.2 Lenguaje de Programación

Entenderemos por un lenguaje de programación, a un lenguaje formal que especifica una serie de instrucciones para que una computadora produzca diversas clases de datos. Los lenguajes de programación pueden usarse para crear programas que pongan en práctica algoritmos específicos que controlen el comportamiento físico y lógico de una computadora. Esta formado por un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones. Al proceso por el cual

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

[7] La ingeniería de Software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de Software y el estudio de estos enfoques, es decir, el estudio de las aplicaciones de la ingeniería al Software. Integra matemáticas, ciencias de la computación y prácticas cuyos orígenes se encuentran en la ingeniería.

se escribe, se prueba, se depura, se compila (de ser necesario) y se mantiene el código fuente de un programa informático se le llama programación.

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

**Variables** son títulos asignados a espacios en memoria para almacenar datos específicos. Son contenedores de datos y por ello se diferencian según el tipo de dato que son capaces de almacenar. En la mayoría de lenguajes de programación se requiere especificar un tipo de variable concreto para guardar un dato específico. A continuación, un listado con los tipos de variables más comunes:

| Tipo de dato | Breve descripción |
|---|---|
| *Char* | Contienen un único carácter |
| *Int* | Contienen un número entero |
| *Float* | Contienen un número decimal |
| *String* | Contienen cadenas de texto |
| *Boolean* | Solo pueden contener[8] verdadero o falso |

**Condicionales** son estructuras de código que indican que, para que cierta parte del programa se ejecute, deben cumplirse ciertas premisas; por ejemplo: que dos valores sean iguales, que un valor exista, que un valor sea mayor que otro. Estos condicionantes por lo general solo se ejecutan una vez a lo largo del programa. Los condicionantes más conocidos y empleados en programación son:

- If: Indica una condición para que se ejecute una parte del programa.
- Else if: Siempre va precedido de un "If" e indica una condición para que se ejecute una parte del programa siempre que no cumpla la condición del if previo y sí se cumpla con la que el "else if" especifique.

[8] En el caso de variables booleanas, el cero es considerado para muchos lenguajes como el literal falso ("False"), mientras que el uno se considera verdadero ("True").

- Else: Siempre precedido de "If" y en ocasiones de "Else If". Indica que debe ejecutarse cuando no se cumplan las condiciones previas.

**Bucles** son parientes cercanos de los condicionantes, pero ejecutan constantemente un código mientras se cumpla una determinada condición. Los más frecuentes son:

- For: Ejecuta un código mientras una variable se encuentre entre 2 determinados parámetros.
- While: Ejecuta un código mientras que se cumpla la condición que solicita.

Hay que decir que a pesar de que existan distintos tipos de bucles, todos son capaces de realizar exactamente las mismas funciones. El empleo de uno u otro depende, por lo general, del gusto del programador.

**Funciones** estas se crearon para evitar tener que repetir constantemente fragmentos de código. Una función podría considerarse como una variable que encierra código dentro de sí. Por lo tanto cuando accedemos a dicha variable (la función), en realidad lo que estamos haciendo es ordenar al programa que ejecute un determinado código predefinido anteriormente.

Todos los lenguajes de programación tienen algunos elementos de formación primitivos para la descripción de los datos y de los procesos o transformaciones aplicadas a estos datos -tal como la suma de dos números o la selección de un elemento que forma parte de una colección-. Estos elementos primitivos son definidos por reglas sintácticas y semánticas que describen su estructura y significado respectivamente.

**Implementación de un Lenguaje de Programación** Es la que provee una manera de que se ejecute un programa para una determinada combinación de Software y Hardware. Existen básicamente dos maneras de implementar un lenguaje:

- Compilación: es el proceso que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación, generando un programa equivalente que la máquina será capaz interpretar. Los programas traductores que pueden realizar esta operación se llaman

compiladores. Estos, como los programas ensambladores avanzados, pueden generar muchas líneas de código de máquina por cada proposición del programa fuente.

- Interpretación: es una asignación de significados a las fórmulas bien formadas de un lenguaje formal. Como los lenguajes formales[9] pueden definirse en términos puramente sintácticos, sus fórmulas bien formadas pueden no ser más que cadenas de símbolos sin ningún significado. Una interpretación otorga significado a esas fórmulas.

Para ayudar al programador es común el uso de editores de texto o ambientes integrados de desarrollo que permiten resaltar los elementos de la sintaxis con colores diferentes para facilitar su lectura. A la forma visible de un lenguaje de programación se le conoce como sintaxis. La mayoría de los lenguajes de programación son puramente textuales, es decir, utilizan secuencias de texto que incluyen palabras, números y puntuación, de manera similar a los lenguajes naturales escritos. Por otra parte, hay algunos lenguajes de programación que son más gráficos en su naturaleza, utilizando relaciones visuales entre símbolos para especificar un programa.

La sintaxis de un lenguaje de programación describe las combinaciones posibles de los símbolos que forman un programa sintácticamente correcto. El significado que se le da a una combinación de símbolos es manejado por su semántica -ya sea formal o como parte del código duro[10] de la referencia de implementación-.

[9] En matemáticas, lógica y ciencias de la computación, un lenguaje formal es un lenguaje cuyos símbolos primitivos y reglas para unir esos símbolos están formalmente especificados. Al conjunto de los símbolos primitivos se le llama alfabeto (o vocabulario) del lenguaje, y al conjunto de las reglas se le llama gramática formal (o sintaxis). A una cadena de símbolos formada de acuerdo a la gramática se le llama fórmula bien formada (o palabra) del lenguaje. Estrictamente hablando, un lenguaje formal es idéntico al conjunto de todas sus fórmulas bien formadas. A diferencia de lo que ocurre con el alfabeto (que debe ser un conjunto finito) y con cada fórmula bien formada (que debe tener una longitud también finita), un lenguaje formal puede estar compuesto por un número infinito de fórmulas bien formadas.

[10] Hard-Code, término del mundo de la informática que hace referencia a una mala práctica en el desarrollo de Software que consiste en incrustar datos directamente (a fuego) en el código fuente del programa, en lugar de obtener esos datos de una fuente externa como un fichero de configuración o parámetros de la línea de comandos, o un archivo de recursos.

La sintaxis de los lenguajes de programación es definida generalmente utilizando una combinación de expresiones regulares -para la estructura léxica- y la Notación de Backus-Naur[11] para la estructura gramática.

**Objetivos a Cumplir en el Código Generado** Para escribir programas que proporcionen los mejores resultados, cabe tener en cuenta una serie de objetivos, entre los que destacan:

- Corrección: Un programa es correcto si hace lo que debe hacer tal y como se estableció en las fases previas a su desarrollo. Para determinar si un programa hace lo que debe, es muy importante especificar claramente qué debe hacer el programa antes de desarrollarlo y, una vez acabado, compararlo con lo que realmente hace.

- Claridad: Es muy importante que el programa sea lo más claro y legible posible, para facilitar así su desarrollo y posterior mantenimiento. Al elaborar un programa se debe intentar que su estructura sea sencilla y coherente, así como cuidar el estilo en la edición; de esta forma se ve facilitado el trabajo del programador, tanto en la fase de creación como en las fases posteriores de corrección de errores, ampliaciones, modificaciones, etc. Fases que pueden ser realizadas incluso por otro programador, con lo cual la claridad es aún más necesaria para que otros programadores puedan continuar el trabajo fácilmente.

- Eficiencia: Se trata de que el programa, además de realizar aquello para lo que fue creado -es decir, que sea correcto-, lo haga gestionando de la mejor forma posible los recursos que utiliza. Normalmente, al hablar de eficiencia de un programa, se suele hacer referencia al tiempo que tarda en realizar la tarea para la que ha sido creado y a la cantidad de memoria que necesita, pero hay otros recursos que también pueden ser de consideración al obtener la eficiencia de un programa, depen-diendo de su naturaleza -espacio en disco que utiliza, tráfico de red que genera, etc.-.

[11] La notación de Backus-Naur, también conocida por sus denominaciones inglesas Backus-Naur form (BNF), Backus-Naur formalism o Backus normal form, es un metalenguaje usado para expresar gramáticas libres de contexto: es decir, una manera formal de describir lenguajes formales.

- Portabilidad: Un programa es portable cuando tiene la capacidad de poder ejecutarse en una plataforma, ya sea Hardware o Software, diferente a aquella en la que se elaboró. La portabilidad es una característica muy deseable para un programa, ya que permite, por ejemplo, a un programa que se ha desarrollado para sistemas GNU/Linux ejecutarse también en la familia de sistemas operativos Windows. Esto permite que el programa pueda llegar a más usuarios más fácilmente.

- Integridad: Un programa tiene integridad si es posible controlar su uso, el grado con que se puede controlar el acceso al Software o a los datos a personas no autorizadas denotará su integridad.

- Facilidad de uso: Un programa tiene facilidad de uso si el esfuerzo requerido es mínimo o moderado para aprender el manejo de una aplicación, trabajar con ella, introducir datos y conseguir resultados.

- Facilidad de mantenimiento: Un programa tiene facilidad de mantenimiento si el esfuerzo requerido para localizar y reparar errores es moderado o mínimo.

- Flexibilidad: Un programa tiene flexibilidad si el esfuerzo requerido para modificar y/o añadir funcionalidades a una aplicación en funcionamiento es moderado o mínimo.

- Reusabilidad: Es el grado en que un programa o porción de este es reusable en otras aplicaciones.

- Interoperatibilidad: Un programa debe poder comunicarse con otras aplicaciones o sistemas informáticos, el esfuerzo necesario para ello medirá su interoperatibilidad.

Existe una gran variedad de lenguajes de programación y su grado de uso depende de diversos factores, entre los que destacan: C, C++, Java y Python.

## 2.3 Conceptos Transversales

En esta sección comentaremos algunos conceptos transversales a los distintos paradigmas de programación y a los lenguajes que los implementan.

**Abstracción** Entendiendo un sistema como una abstracción de la realidad, los datos son las entidades que representan cada uno de los aspectos de la realidad que son significativos para el funcionamiento del sistema. Para que tenga sentido como abstracción de la realidad, cada dato implica un determinado valor y requiere de una convención que permita representarlo sin ambigüedad y procesarlo de una manera confiable.

En la misma línea, la lógica del procesamiento de los datos es también una abstracción de los procesos que suceden en la realidad que conforma el dominio de la aplicación. El proceso de abstraerse progresivamente de los detalles y así manejar niveles de abstracción, es el que permite construir sistemas complejos.

Las unidades de Software que realizan la funcionalidad del sistema requieren también de convenciones y criterios de ordenamiento y articulación interna tanto para un funcionamiento confiable y eficiente del sistema, como para que el proceso de construcción y mantenimiento del Software sea de la forma más simple posible.

**Modularización, Encapsulamiento y Delegación** Una estrategia central de la programación es buscar la manera de organizar y distribuir la funcionalidad de un sistema complejo en unidades más pequeñas de Software con que se responsabilizan de tareas específicas y que interactúan entre ellas. Estas unidades reciben nombres diferentes según cada lenguaje de programación, como rutinas, funciones, procedimientos, métodos, predicados, subprogramas, bloques, entidades, siendo "módulos" una de las más frecuentes y que dan origen al término.

La clave de cada módulo es no conocer el funcionamiento interno de los demás módulos con los que interactúa, sino sólo su interfaz, es decir, la forma en que debe enviarle información adicional en forma de parámetros y cómo va a recibir las respuestas. Esta propiedad recibe diversos nombres, como el de encapsulamiento, ocultación de información o "caja negra". Ante la modificación de una funcionalidad en particular del sistema, en la medida que su implementación esté encapsulada en un módulo, el impacto que produce su cambio no afectará a los otros módulos que interactúan con el.

En concordancia con la distribución de responsabilidades entre las diferentes unidades de Software, la delegación consiste en la invocación que desde un módulo se efectúa a otro módulo, de manera que el que invoca indica de forma explicita qué es lo que pretende y el que es invocado se ocupa de todo lo

necesario para realizarlo. Puede realizarse de numerosas maneras, variando el criterio de distribución de responsabilidades, el modo de evaluación, la forma de paso de parámetros, los tipos de datos que utiliza, de acuerdo a las posibilidades y restricciones de cada lenguaje y paradigma.

**Declaratividad** La declaratividad, en términos generales, se basa en la separación del conocimiento sobre la definición del problema con la forma de buscar su solución, una separación entre la lógica y el control.

En un programa declarativo se especifican un conjunto de declaraciones, que pueden ser proposiciones, condiciones, restricciones, afirmaciones, o ecuaciones, que caracterizan al problema y describen su solución. A partir de esta información el sistema utiliza mecanismos internos de control, comúnmente llamado "motores", que evalúan y relacionan adecuadamente dichas especificaciones, la manera de obtener la solución. De esta forma, en vez de ser una secuencia de órdenes, un programa es un conjunto de definiciones sobre el dominio del problema.

Basándose en la noción de delegación, la declaratividad plantea como criterio para distribuir las responsabilidades, separar las relacionadas con modelar o definir el conocimiento del problema de aquellas de manipular ese conocimiento para alcanzar un objetivo concreto. En otras palabras, distinguir el "qué" del "cómo".

La declaratividad brinda la posibilidad de usar una misma descripción en múltiples contextos en forma independiente de los motores que se utilicen.

Permite focalizar por un lado en las cuestiones algorítmicas del motor, por ejemplo para trabajar en forma unificada sobre eficiencia, y por otro en la definición del dominio del problema y la funcionalidad de la aplicación en sí.

La noción opuesta, aunque en cierta medida complementaria, de la declaratividad, puede denominarse "proceduralidad". Los programas procedurales se construyen indicando explícitamente la secuencia de ejecución en la que se procesan los datos y obtienen los resultados. Para ello se detalla un conjunto de sentencias, ordenadas mediante estructuras de control como decisiones, iteraciones y secuencias, que conforman "algoritmos".

En un sistema complejo, no se puede hablar de declaratividad o proceduralidad como conceptos excluyentes o totalizantes, sino que coexisten y se relacionan en una permanente tensión. Dependiendo de los lenguajes y de las herramientas que se utilicen, y en particular del diseño del sistema, habrá

partes del sistema que por su sentido o ubicación dentro del sistema global serán más declarativas o procedurales que otras, logrando con ello aprovechar las ventajas respectivas.

**Tipos de Datos** Un tipo de dato, o como también es llamado, un tipo abstracto de dato, es un conjunto de valores y de operaciones asociadas a ellos. La utilización de diversos tipos de datos permite la agrupación o clasificación del gran volumen y variedad de valores que es necesario representar y operar en un sistema, según sus semejanzas y diferencias.

Tomando como criterio las similitudes en cuanto al contenido de lo que representan, una primera condición es la existencia de un conjunto de valores o entidades homogéneos en su representación. Otra condición es que los elementos del mencionado conjunto se comporten en forma uniforme respecto a una serie de operaciones.

Cada paradigma de programación, y en particular cada lenguaje, tiene su forma de determinar tanto la conformación de cada tipo de dato, con sus valores y operaciones, como la forma en que se relacionan entre sí conformando un sistema de tipo de datos.

En un lenguaje fuertemente tipado, toda variable y parámetro deben ser definidos de un tipo de dato en particular que se mantiene sin cambios durante la ejecución del programa, mientras que en uno débilmente tipado no, sino que pueden asumir valores y tipos de datos diferentes durante la ejecución del programa.

El tipo de dato al que pertenece una entidad determina la operatoria que se puede realizar con el. Una tarea que realizan muchos de los lenguajes de programación como forma de su mecanismo interno es el chequeo del tipo de dato de las entidades del programa. Esta acción se realiza de diferentes maneras, con mayor o menor flexibilidad, y en diferentes momentos, como la compilación o la ejecución del programa, y en otros no se realiza.

De todas maneras, el tipo de datos permite entender qué entidades tienen sentido en un contexto, independientemente de la forma de chequeo o si el tipado es débil o fuerte. Por ejemplo, a un bloque de Software que recibe como argumento una variable, conocer de qué tipo de dato es, le permite saber qué puede hacer con ella.

**Estructuras de Datos** Los valores atómicos, es decir, aquellos que no pueden ser descompuestos en otros valores, se representan mediante tipos de

datos simples.

En contrapartida, en un tipo de dato compuesto los valores están compuestos a su vez por otros valores, de manera que conforma una estructura de datos. Cada uno de los valores que forman la estructura de datos corresponde a algún tipo de dato que puede ser tanto simple como compuesto.

Su utilidad consiste en que se pueden procesar en su conjunto como una unidad o se pueden descomponer en sus partes y tratarlas en forma independiente. En otras palabras, las estructuras de datos son conjuntos de valores.

**Polimorfismo y Software Genérico** El polimorfismo es un concepto para el que se pueden encontrar numerosas y diferentes definiciones. En un sentido amplio, más allá de las especificidades de cada paradigma y del alcance que se le dé a la definición del concepto desde la perspectiva teórica desde la que se le aborde, el objetivo general del polimorfismo es construir piezas de Software genéricas que trabajen indistintamente con diferentes tipos de entidades, para otra entidad que requiere interactuar con ellas; en otras palabras, que dichas entidades puedan ser intercambiables. Mirando con mayor detenimiento los mecanismos que se activan y sus consecuencias para el desarrollo de sistemas, se pueden distinguir dos grandes situaciones.

Hay polimorfismo cuando, ante la existencia de dos o más bloques de Software con una misma interfaz, otro bloque de Software cualquiera puede trabajar indistintamente con ellos. Esta noción rescata la existencia de tantas implementaciones como diferentes tipos de entidades con que se interactúe.

Una entidad emisora puede interactuar con cualquiera de las otras entidades de acuerdo a las características de la interfaz común, y la entidad receptora realizará la tarea solicitada de acuerdo a la propia implementación que tenga definida, independientemente de las otras implementaciones que tengan las otras entidades. Consistentemente con la noción de delegación, la entidad emisora se desentiende de la forma en que las otras entidades implementaron sus respuestas, ya sea igual, parecida o totalmente diferente.

Analizando el concepto desde el punto de vista de las entidades que responden a la invocación, el proceso de desarrollo debe contemplar la variedad y especificidad de cada una para responder adecuadamente a lo que se les solicita. Desde el punto de vista de la entidad que invoca, el proceso es transparente, y es en definitiva esta, la que se ve beneficiada por el uso del concepto.

Otra forma de obtener un bloque de Software genérico es ante el caso en

que las entidades con las que el bloque quiere comunicarse, en vez de tener diferentes implementaciones, aún siendo de diferente tipo, sean lo suficientemente similares para compartir una misma y única implementación.

Desde el punto de vista de la entidad que invoca, el proceso continúa siendo transparente y sigue aprovechando los beneficios de haber delegado la tarea y haberse despreocupado de qué tipo de entidad es la receptora. En este caso, el concepto de polimorfismo se relaciona o se basa en la coexistencia de herencia, de variables de tipo de dato o en las formas débiles de declaración y chequeo de tipos de datos, dependiendo de las diferentes herramientas de cada paradigma.

**Asignación y Unificación** La asignación destructiva es una operación que consiste en cambiar la información representada por una variable, de forma tal que si se consulta su valor antes y después de dicha operación, se obtiene un resultado distinto. Estas asignaciones se realizan repetitivamente sobre la misma celda de memoria, remplazando los valores anteriores.

La asignación determina el estado de una variable, que consiste en el valor que contiene en un momento en particular. La asignación destructiva es la forma más usual de provocar efecto de lado, pero no la única, ya que, dependiendo de los lenguajes, hay otro tipo de instrucciones que también permiten modificar el estado de información de un sistema.

La unificación es un mecanismo por el cual una variable que no tiene valor, asume un valor. Una vez unificada, o "ligada", como también se le dice, una variable no cambia su valor, por lo que no existe la noción de estado. La duración de la unificación está condicionado por el alcance que tienen las variables en cada lenguaje en particular, pudiendo una variable unificarse con varios valores alternativos en diferentes momentos de la ejecución del bloque de Software en el que se encuentran. Se suele denominar como "indeterminada" a una variable sin ligar o no unificada.

**Modo de Evaluación** Los parámetros que se utilizan en la invocación de un bloque de Software cualquiera pueden ser evaluados en diferentes momentos de acuerdo al modo de evaluación que utilice el lenguaje de programación.

La evaluación ansiosa consiste en que los argumentos son evaluados antes de invocar al bloque de Software y es responsabilidad de la entidad que los invoca.

La evaluación diferida plantea que la evaluación de los argumentos es res-

ponsabilidad del bloque de Software invocado, quien decide el momento en que lo hará. Provoca que se difiera la evaluación de una expresión, permitiendo que en algunos casos, de acuerdo a cómo sea la implementación, no sea necesario evaluarla nunca, con el consiguiente beneficio en términos de eficiencia.

La evaluación diferida es utilizada en cierto tipo de situaciones que serían consideradas erróneas o imposibles de resolver con la evaluación ansiosa, como por ejemplo los bucles o las listas infinitas.

**Orden Superior** Asumiendo un esquema de un único orden, en un programa existen por un lado datos y por otro los procedimientos -y todo bloque de Software- que trabajan con ellos, de manera tal que los proce-dimientos reciben datos como parámetros y devuelven datos como resultados. La noción de orden superior plantea que los procedimientos son tratados como datos y en consecuencia pueden ser utilizados como parámetros, representados en variables, devueltas como resultados u operados en cálculos más complejos con otros datos. Se denomina de orden superior a los procedimientos que reciben como argumentos a otros procedimientos.

**Recursividad** La recursividad, entendida como iteración con asignación no destructiva, está relacionada con el principio de inducción. En general, un bloque de Software recursivo se define con al menos un término recursivo, en el que se vuelve a invocar el bloque que se está definiendo, y algún término no recursivo como caso base para detener la recursividad.

## 2.4 Algo de Programación

**Tipos de datos** Dependiendo del lenguaje (véase [12], [13], [14] y [15]), la implementación del mismo y la arquitectura en la que se compile/ejecute el programa, se tienen distintos tipos de datos, pero los básicos son[12]:

- char, 8 bits, rango de -128 a 127
- unsigned char, 8 bits, rago de 0 a 255
- int, 16 bits, rango de -32,768 a 32,767

[12]La precedencia de los operadores básicos son: $=$ el de mayor procedencia, siguen $-$ y $+$ unario, luego $*, /, \%$ y finalmente los operadores $+$ y $-$.

- unsigned int, 16 bits, rango de 0 a 65,535
- long int, 32 bits, rango de -2,147,483,648 a 2,147,483,647
- long, 64 bits, rango de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
- float, 32 bits, rango de -3.4x$10^{38}$ a 3.4x$10^{38}$, con 14 dígitos de precisión
- double, 64 bits, rango de -1.7x$10^{308}$ a 1.7x$10^{308}$, con 16 dígitos de precisión
- long double, 80 bits, 3.4x$10^{-4932}$ a 3.4x$10^{4932}$, con 18 dígitos de precisión
- boolean, 1 bit, rango de True o False

La conversión entre los diferentes tipos de datos esta en función del lenguaje y existe una perdida de datos si se hace de un tipo de dato que no "quepa" en el rango del otro; estos y otros errores de programación han generado graves errores en el sistema automatizado de control que han desembocado por ejemplo en, misiones fallidas de cohetes espaciales y su carga en numerosas ocasiones[13], la quiebra de la empresa Knight Capital 2012, sobredosis de radioterapia en 1985 a 1987, la desactivación de servidores de Amazon en el 2012, el apagón en el noreste de EE.UU. en el 2003, fallo en el sistema de reserva de vuelos de American Airlines en el 2013, fallo en el sistema de misiles estadounidense Patriot en Dhahran en el 1991, etc.

**Aritmética de Punto Flotante** La aritmética de punto flotante es considerada un tema esotérico para muchas personas. Esto es sorprendente porque el punto flotante es omnipresente en los sistemas informáticos. Casi todos los lenguajes de programación tienen un tipo de datos de punto flotante, i.e. los números no enteros como 1.2 ó $1e+45$.

Los problemas de precisión del punto flotante son una preocupación crítica en la informática y la computación numérica, donde la representación y manipulación de números reales puede conducir a resultados inesperados y erróneos. Estos obstáculos surgen debido alas limitaciones inherentes de la aritmética de punto flotante, que aproxima números reales con una precisión finita.

Esto puede causar problemas importantes en diversas aplicaciones, desde simulaciones científicas hasta cálculos financieros, donde incluso las imprecisiones menores pueden propagarse y amplificarse, dando lugar a errores

[13] Una de las pérdidas, fue el satélite mexicano Centenario en mayo del 2015.

sustanciales. Comprender estos obstáculos es esencial para que los desarrolladores, ingenieros y científicos implementen algoritmos numéricos sólidos y confiables, garantizando que las matemáticas realizadas por las computadoras sigan siendo confiables y precisas.

**Comprensión de los Errores de Redondeo en Aritmética de Punto Flotante** El meollo de la cuestión es la representación de números de punto flotante. Las computadoras utilizan una cantidad finita de bits para almacenar estos números, y generalmente cumplen con el estándar IEEE 754[14]. Este estándar define cómo se almacenan los números en formato binario, con un número fijo de bits asignados para el signo, el exponente y la mantisa. Si bien esto permite representar una amplia gama de valores, también impone limitaciones a la precisión. No todos los números decimales se pueden representar exactamente en forma binaria, lo que genera pequeñas discrepancias conocidas como errores de redondeo.

Una de las primeras cosas que uno encuentra con sorpresa cuando hace cálculos en una computadora es que por ejemplo, si usamos números muy grandes y seguimos incrementando su valor eventualmente el resultado será negativo ... ¿qué pasó? Esto se llama desbordamiento aritmético al intentar crear un valor numérico que está fuera del rango que puede representarse con un número dado de dígitos, ya sea mayor que el máximo o menor que el mínimo valor representable. Algo similar pasa al restar al menor número representable en la máquina, el resultado será positivo y se denomina subdesbordamiento.

Por otro lado, tenemos errores de redondeo, estos ocurren porque el sistema binario no puede representar con precisión ciertas fracciones. Por ejemplo, el número decimal 0.1 no se puede representar exactamente en binario, lo que da como resultado una aproximación, por ejemplo al sumar 0.1 + 0.2 en una computadora usando por ejemplo el lenguaje Python obtenemos:

```
print(0.1+0.2)
0.30000000000000004
```

que no es exactamente lo que esperábamos[15]. Cuando se utilizan tales aproximaciones en los cálculos, los errores pueden acumularse. Este fenó-

[14] El estándar IEEE 754-2008 define varios tamaños de números de punto flotante: media precisión (binary16), precisión simple (binary32), precisión doble (binary64), precisión cuádruple (binary128), etc., cada uno con su propia especificación.

[15] Lo mismo obtenemos si usamos la multiplicación, por ejemplo:

meno es particularmente problemático en procesos iterativos, donde el mismo cálculo se realiza repetidamente y los errores se acumulan con el tiempo.

Estos errores de precisión se vuelven particularmente problemáticos cuando se realizan controles de igualdad. En un mundo ideal, comparar la igualdad de dos números de punto flotante sería sencillo. Sin embargo, debido a los pequeños errores introducidos durante las operaciones aritméticas, dos números que deberían ser iguales pueden no ser exactamente iguales en su representación binaria.

Por ejemplo, el resultado de sumar 0.1 y 0.2 puede no ser exactamente igual a 0.3 debido a la forma en que estos números se representan en binario, por ejemplo:

```
if 0.1 + 0.2 == 0.3:
    print("si")
else:
    print("no")
```

la respuesta será "no". Esta discrepancia puede provocar que fallen las comprobaciones de igualdad, lo que provocará un comportamiento inesperado en los programas.

Para mitigar estos problemas, los desarrolladores suelen utilizar una técnica conocida como "comparación épsilon". En lugar de verificar la igualdad exacta, verifican si la diferencia absoluta entre dos números de punto flotante es menor que un valor pequeño predefinido, conocido como épsilon. Este enfoque reconoce la imprecisión inherente de la aritmética de punto flotante y proporciona una forma más sólida de comparar números. Sin embargo, elegir un valor épsilon apropiado puede resultar complicado, ya que depende del contexto específico y del rango de valores involucrados.

El problema se ve exacerbado por la precisión finita de los números de punto flotante. Al realizar operaciones aritméticas, el resultado a menudo debe redondearse para que se ajuste a los bits disponibles. Este redondeo puede introducir más imprecisiones. Por ejemplo, sumar dos números de punto flotante de magnitudes muy diferentes puede provocar una pérdida de precisión, ya que el número más pequeño puede ignorarse de hecho. Esto

```
print(0.1 * 3)
0.30000000000000004
```

se conoce como cancelación catastrófica y puede afectar gravemente a la precisión de los cálculos.

Por ejemplo, ¿qué podría tener de interesante la humilde fórmula cuadrática?. Después de todo, es una fórmula. Simplemente le pones números. Bueno, hay un detalle interesante. Cuando el coeficiente lineal $b$ es grande en relación con los otros coeficientes, la fórmula cuadrática puede dar resultados incorrectos cuando se implementa en aritmética de punto flotante. Eso es cierto, pero veamos qué sucede cuando tenemos $a = c = 1$ y $b = 10e8$ en

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ y } x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{2.1}$$

regresa

$$x_1 = -7.450580596923828e - 09 \text{ y } x_2 = -100000000.0$$

La primera raíz está equivocada en aproximadamente un 25%, aunque la segunda es correcta.

¿Qué pasó? La ecuación cuadrática violó la regla cardinal del análisis numérico: evitar restar números casi iguales. Cuanto más similares sean dos números, más precisión puedes perder al restarlos. En este caso $(b^2 - 4ac)$ es casi igual a $b$. Si evaluamos, obtenemos $1.49e - 8$ cuando sería la respuesta correcta $2.0e - 8$.

Si usamos la otra formula[16]

$$x_1 = \frac{-2c}{b + \sqrt{b^2 - 4ac}} \text{ y } x_2 = \frac{-2c}{b - \sqrt{b^2 - 4ac}} \tag{2.2}$$

obtenemos

$$x_1 = -1e - 08 \text{ y } x_2 = -134217728.0$$

Entonces, ¿cuál formula cuadrática es mejor? Da la respuesta correcta para la primera raíz, exacta dentro de la precisión de la máquina. Pero ahora la segunda raíz está equivocada en un 34%. ¿Por qué la segunda raíz es incorrecta? La misma razón que antes: ¡restamos dos números casi iguales!

La versión familiar de la fórmula cuadrática calcula correctamente la raíz más grande y la otra versión calcula correctamente la raíz más pequeña. Ninguna versión es mejor en general. No estaríamos ni mejor ni peor usando

[16] Para obtener la segunda fórmula, multiplique el numerador y denominador de $x_1$ por $-b - \sqrt{b^2 - 4ac}$ (y de manera similar para $x_2$).

siempre la nueva fórmula cuadrática que la anterior. Cada uno es mejor cuando evita restar números casi iguales. La solución es utilizar ambas fórmulas cuadráticas, utilizando la apropiada para la raíz que estás intentando calcular.

Otro error común es la suposición de que la aritmética de punto flotante es asociativa y distributiva, como lo es en matemáticas puras. En realidad, el orden de las operaciones puede afectar significativamente el resultado debido a errores de redondeo. Por ejemplo, la expresión (a+ b) + c puede producir un resultado diferente que a + (b + c) cuando se trata de números de punto flotante. Esta no asociatividad puede provocar errores sutiles, especialmente en algoritmos complejos que se basan en cálculos precisos.

Además, la aritmética de punto flotante puede introducir problemas en los algoritmos que requieren comparaciones exactas, como los que se utilizan en la clasificación o la búsqueda. Cuando los números de punto flotante se utilizan como claves en estructuras de datos como tablas hash o árboles de búsqueda binarios, los errores de precisión pueden provocar un comportamiento incorrecto, como no encontrar un elemento existente o insertar incorrectamente un duplicado. Para solucionar este problema, es posible que los desarrolladores necesiten implementar funciones de comparación personalizadas que tengan en cuenta la imprecisión del punto flotante.

Pasando a implicaciones prácticas, estos errores de redondeo pueden tener consecuencias de gran alcance. En las simulaciones científicas, pequeñas imprecisiones pueden dar lugar a predicciones incorrectas, lo que podría socavar la validez de la investigación. En aplicaciones financieras, los errores de redondeo pueden dar lugar a importantes discrepancias monetarias, afectando todo, desde el análisis del mercado de valores hasta las transacciones bancarias. Incluso en aplicaciones cotidianas como la representación de gráficos, los errores de redondeo pueden causar artefactos visuales que restan valor a la experiencia del usuario.

Además, comprender las limitaciones de la aritmética de punto flotante puede ayudar a diseñar sistemas más robustos. Al anticipar dónde es probable que se produzcan errores de redondeo, los desarrolladores pueden implementar controles y equilibrios para detectar y corregir imprecisiones. Por ejemplo, la aritmética de intervalos puede proporcionar límites a los posibles valores de un cálculo, ofreciendo una manera de cuantificar la incertidumbre introducida por los errores de redondeo.

**Estrategias para Mitigar los Problemas de Precisión del Punto Flotante en el Desarrollo de Software** Los problemas de precisión del punto flotante son un desafío común en el desarrollo de Software y a menudo conducen a resultados inesperados y errores sutiles. Estos problemas surgen debido a las limitaciones inherentes a la representación de números reales en formato binario, lo que puede provocar errores de redondeo y pérdida de precisión. Para mitigar estos obstáculos, los desarrolladores deben emplear una variedad de estrategias que garanticen la precisión numérica y la confiabilidad en sus aplicaciones.

Una estrategia eficaz es utilizar tipos de datos de mayor precisión cuando sea necesario. Si bien los tipos de punto flotante estándar como "flotante" y "doble" son suficientes para muchas aplicaciones, es posible que no proporcionen la precisión requerida para cálculos más sensibles. En tales casos, el uso de tipos de precisión extendidos, como "long double" en C++ o bibliotecas de precisión arbitraria como GMP (Biblioteca aritmética de precisión múltiple GNU), puede reducir significativamente el riesgo de pérdida de precisión.

Sin embargo, es importante equilibrar la necesidad de precisión con consideraciones de rendimiento, ya que los tipos de mayor precisión pueden resultar costosos desde el punto de vista computacional. Otro enfoque consiste en implementar algoritmos numéricos que sean inherentemente más estables. Algunos algoritmos son más susceptibles a errores de redondeo que otros y elegir el algoritmo correcto puede marcar una diferencia significativa.

Por ejemplo, al resolver sistemas de ecuaciones lineales, utilizar métodos como la descomposición LU o la descomposición QR puede ser más estable que la eliminación gaussiana. Además, se pueden emplear técnicas de refinamiento iterativo para mejorar la precisión de la solución corrigiendo iterativamente los errores introducidos por la aritmética de punto flotante.

Los desarrolladores deben tener en cuenta el orden de las operaciones en sus cálculos. Las propiedades asociativas y distributivas de la aritmética no siempre se cumplen en la aritmética de punto flotante debido a errores de redondeo. Por lo tanto, reorganizar el orden de las operaciones a veces puede conducir a resultados más precisos. Por ejemplo, al sumar una gran cantidad de números de punto flotante, sumarlos en orden de magnitud ascendente puede minimizar la acumulación de errores de redondeo. Esta técnica, conocida como suma de Kahan, ayuda a preservar la precisión al compensar los pequeños errores que ocurren durante el proceso de suma.

Además de estas estrategias, es fundamental realizar pruebas y validaciones exhaustivas del Software numérico. Las pruebas unitarias deben di-

señarse para cubrir una amplia gama de valores de entrada, incluidos casos extremos que probablemente expongan problemas de precisión. Comparar los resultados de los cálculos de punto flotante con soluciones analíticas conocidas o utilizar aritmética de mayor precisión como referencia puede ayudar a identificar discrepancias.

Además, se puede realizar un análisis de sensibilidad para evaluar cómo pequeños cambios en los valores de entrada afectan la salida, proporcionando información sobre la estabilidad y confiabilidad de los algoritmos numéricos.

Por último, la documentación y la comunicación desempeñan un papel fundamental a la hora de mitigar los problemas de precisión del punto flotante. Los desarrolladores deben documentar las limitaciones y suposiciones de sus algoritmos numéricos, así como cualquier fuente potencial de error. Esta información es invaluable para otros desarrolladores que necesiten mantener o ampliar el Software. Además, una comunicación clara con las partes interesadas sobre la precisión esperada del Software puede ayudar a gestionar las expectativas y evitar mal entendidos.

**Nombres de Variables** Los nombres de las variables deben:

- Empezar por una letra y solo pueden contener letras, números y '_'.
- Las constantes se escriben en mayúsculas

Ejemplos para C, C++, Java

```
int numeroEntero = 5;
double numeroDecimal = 3.14;
boolean esEstudiante = true;
char inicial = 'J';
String nombre = "Juan"
```

Ejemplos para Python

```
Valid namescat_color = 'Brown'
number_of_threads = 8
phone_number = 78469334212
ISIN_CODE = 8479362
CONSTANT_SPEED = 9.8
```

**Condicionales** El flujo de programas a menudo tiene que dividirse. Es decir, si una condición se cumple, se hace algo, y si no, se hace otra cosa. Los enunciados condicionales permiten cambiar el comportamiento del programa de acuerdo a una condición dada, la estructura básica de un condicional es:

Pseudocódigo:

```
Si <condición> entonces
  <acción>
```

Ejemplo en C, C++ y Java[17]:

```
if (x > 0) x*= 10;
```

```
if (x > 0) {
  x *= 10;
}
```

Ejemplo en Python:

```
if x > 0:
  x = x * 10
```

La estructura completa del condicional es:
Pseudocódigo:

```
Si <condición> entonces
  <acción>
else
  <acción>
```

Ejemplo en C, C++ y Java:

```
if (x % 2 == 0) x = x * x;
else x = x * 2;
```

```
if (x % 2 == 0) {
  x = x * x;
} else {
  x = x * 2;
}
```

[17] En condicionales, tienen mayor precedencia $<, >, >=, <=$, luego están $==$ y $!=$, finalmente && (AND) y || (OR).

Ejemplo en Python:

```
if x % 2 == 0:
  x = x * x
else:
  x = x * 2
```

La estructura condicional encadenada es:
Pseudocódigo:

```
Si <condición> entonces
  <acción>
else if
  <acción>
else
  <acción>
```

Ejemplo en C, C++ y Java:

```
if (x < y) {
  z = x;
} else if (x > y) {
  z = y;
} else {
  z = 0;
}
```

Ejemplo en Python:

```
if x < y:
  z = x
elif[18] x > y:
  z = y
else:
  z = 0
```

[18] elif es una contracción de "else if", sirve para enlazar varios "else if", sin tener que aumentar las tabulaciones en cada nueva comparación.

Condicionales con operadores lógicos, tenemos dos operadores lógicos el AND (&&) y el OR (||), ejemplos:

Pseudocódigo:

```
Si <condición operador condición> entonces
  <acción>
else
  <acción>
```

Ejemplo en C, C++ y Java:

```
if (x % 2 == 0 || x > 0) x = x * x;
else x = x * 2;
```

```
if (x % 2 == 0 && x > 0) {
  x = x * x;
} else {
  x = x * 2;
}
```

Ejemplo en Python:

```
if x % 2 == 0 and x > 0:
  x = x * x
else:
  x = x * 2
```

**Bucle for** La implementación de bucles mediante el comando *for* permite ejecutar el código que se encuentre entre su cuerpo mientras una variable se encuentre entre 2 determinados parámetros, la estructura básica es:

Pseudocódigo:

```
for <inicio; mientras; incremento/decremento> entonces
  <acción>
```

Ejemplo en C, C++ y Java:

```
for (x = 0; x < 21; x++) {
  z = z + x;
}
```

Ejemplo en Python:

```
for x in range(0, 21):
  z = z + x
for i in range(1,3):
  if n == 3:
    break
else[19]
  print("no se encontro el numero 3")
```

**Bucle for para navegar en por ejemplo Listas** Si se tene una lista declarada, es posible hacer un recorrido en sus elementos usando *for*, por ejemplo:

Ejemplo en Java:

```
List<String> lista = new ArrayList<String>();
...
for (String elem : lista) {
     System.out.print(elem);
}
```

Ejemplo en Python:

```
lista = ["uno","dos","tres","cuatro"]
for elem in lista:
     print elem
```

**Bucle while** La implementación de bucles mediante el comando *while* permite ejecutar el código que se encuentre entre su cuerpo mientras una condición se cumple, la estructura básica es:

Pseudocódigo:

```
while <condición> entonces
  <acción>
```

Ejemplo en C, C++ y Java:

[19] La instrucción *else* sólo se ejecutará si concluye normalmente el ciclo *for*, es decir, sin la interrupción por *break*. En este caso nunca se ejecutara el *print*.

```
while (z < 20) {
  z = z + x;
}
```

Ejemplo en Python:

```
while z < 20:
  z = z + x
else[20]
  print("concluyo")
```

**Bucle do-while** La implementación de bucles mediante el comando *do-while* permite ejecutar el código que se encuentre entre su cuerpo y después revisa si se cumple la condición para continuar el ciclo, se puede usar *do-while* en C, C++ y Java, pero no esta presente en Python (pero siempre se puede usar un *while* para emularlo). La estructura básica es:

Pseudocódigo:

```
do
  <acción>
while <condición>
```

Ejemplo en C, C++ y Java:

```
do {
  z = z + x;
} while (z < 20);
```

**Condicional switch** Cuando se requiere hacer un condicional sobre una variable y que esta tenga varias alternativas, una opción para evitar una cadena de sentencias *if-else*, se puede usar *switch* en C, C++ y Java, pero no esta presente en Python (siempre se puede usar *if-elif* para emularlo). La estructura básica es:

Pseudocódigo:

[20] La instrucción *else* sólo se ejecutará si concluye normalmente el ciclo *while*, es decir, sin la interrupción por *break*. En este caso al terminar el *while* se ejecutara el *print*.

```
switch(expresion) {
  case constante:
    <acción>
    break;
  case constante:
    <acción>
    break;
  .
  .
  .
  default:
  <acción>
}
```

son opcionales los *break* y el *default*.

Ejemplo en C, C++ y Java:

```
switch(i) {
  case 1:
    x=23;
  case 2:
    x ++;
    break;
  default:
  x=0;
}
```

**Funciones** una función es un bloque de código que realiza alguna operación. Los nombres de las funciones deben empezar por una letra y solo pueden contener letras, números y '_'.

Una función tiene tres componentes importantes:

- Los parámetros, que son los valores que recibe la función como entrada;
- El código de la función, que son las operaciones que hace la función; y
- El resultado (o valor de retorno), que es el valor final que entrega la función.

La siguiente función acepta dos enteros de llamada y devuelve su suma; a y b son parámetros de tipo int.

En el caso de C y C++:

```
int suma(int a, int b)
{
return a + b;
}
```

Las variables que son creadas dentro de la función (incluyendo los parámetros y el resultado) se llaman variables locales, y sólo son visibles dentro de la función, no desde el resto del programa. Por otra parte, las variables creadas fuera de alguna función se llaman variables globales, y son visibles desde cualquier parte del programa.

No hay ningún límite práctico para la longitud de la función, pero el buen diseño tiene como objetivo las funciones que realizan una sola tarea bien definida. Los algoritmos complejos deben dividirse en funciones más sencillas y fáciles de comprender siempre que sea posible.

La función puede ser invocada, o llamada, desde cualquier lugar del programa. Los valores que se pasan a la función son los argumentos, cuyos tipos deben ser compatibles con los tipos de los parámetros en la definición de la función.

En el caso de C y C++:

```
int main()
{
int i = suma(10, 32);
int j = suma(i, 66);
printf("El valor de j es: ", j);
}
```

Para el caso de Python tenemos:

```
def sum (a, b):
    return a + b
i = sum(10, 32)
j = sum(i, 66)
print("El valor de j es: ", j)
```

**Primer ejemplo** Como ya se ha hecho costumbre, el primer ejemplo de un programa es: Hola Mundo, así que iniciemos con ello creando el archivo correspondiente *hola.ext*[21] en cualquier editor de texto o en un IDE, entonces:

Ejemplo en C:

```
#include <stdio.h>
int main(void) {
   printf("Hello World\n");
   return 0;
}
```

para compilarlo usamos *gcc* en línea de comandos mediante:

```
$ gcc hola.c -o hola
```

y lo ejecutamos con:

```
$ ./hola
```

Ejemplo en C++:

```
#include <iostream>
int main() {
   std::cout << "Hello World!\n";
}
```

para compilarlo usamos *g++* en línea de comandos mediante:

```
$ g++ hola.cpp -o hola
```

y lo ejecutamos con:

```
$ ./hola
```

[21] La extensión depende del lenguaje de programación, para el lenguaje C la extensión es .c, para el lenguaje C++ la extensión es .cpp, para el lenguaje Java la extensión es .java, para el lenguaje Python la extensión es .py.

Ejemplo en Java:

```
class hola {
   public static void main(String[] args) {
      System.out.println("Hello world!");
   }
}
```

para compilarlo usamos *javac* en línea de comandos mediante:

```
$ javac hola.java
```

y lo ejecutamos con:

```
$ java hola
```

Ejemplo en Python 2:

```
print "Hello World\n"
```

para compilarlo y ejecutarlo usamos *python2* en línea de comandos mediante:

```
$ python2 hola.py
```

Ejemplo en Python 3:

```
print ("Hello World\n")
```

para compilarlo y ejecutarlo usamos *python3* en línea de comandos mediante:

```
$ python3 hola.py
```

**Un Ejemplo de Cálculo de Primos** El algoritmo más simple, para determinar si un número es primo o compuesto, es hacer una serie de divisiones sucesivas del número, con todos los números primos menores que él, si alguna división da como residuo 0 o es divisible con el número entonces es compuesto en caso contrario es primo.

- Iniciamos con el número 4
- Verificamos si es divisible con los primos almacenados (iniciamos con 2 y 3)
- Si es divisible con algún número primo entonces es compuesto, en caso contrario es primo y este se guarda como un nuevo primo
- se incrementa uno al número y se regresa a verificar si es divisible entre los números primos almacenados, hasta encontrar por ejemplo los primeros mil primos

Ejemplo en C y C++:

```
#include <stdio.h>
// NPB Numero de primos a buscar
#define NPB 1000
int main ()
{
  int n, i, np;
  int p[NPB];
  // Guarda los primeros 2 primos
  p[0] = 2;
  p[1] = 3;
  np = 2;
  // Empieza la busqueda de primos a partir del numero 4
  n = 4;
  // Ciclo para buscar los primeros NPB primos
  while (np < NPB)
  {
    for (i = 0; i < np; i++)
    {
      if((n % p[i]) == 0) break;
```

```
        }
        if(i == np)
        {
          p[i] = n;
          np++;
        }
        n++;
      }
      // Visualiza los primos encontrados
      printf("\nVisualiza los primeros %d primos\n", NPB);
      for (i = 0; i < NPB; i++)
        printf("%d\n", p[i]);
      return 0;
    }
```

Ejemplo en Java:

```
    public class CalPrimos {
      public static void main(String[] args) {
        // NPB Numero de primos a buscar
        int NPB = 1000;
        int n, i, np;
        int p[] = new int[NPB];
        // Guarda los primeros 2 primos
        p[0] = 2;
        p[1] = 3;
        np = 2;
        // Empieza la busqueda de primos a partir del numero 4
        n = 4;
        // Ciclo para buscar los primeros NPB primos
        while (np < NPB) {
          for (i = 0; i < np; i++) {
            if((n % p[i]) == 0) break;
          }
          if (i == np) {
            p[i] = n;
            np++;
          }
```

```
            n++;
          }
          // Visualiza los primos encontrados
          System.out.println("Visualiza los primeros " + NPB + "
primos");
          for (i = 0; i < NPB; i++) System.out.println(p[i]);
        }
      }
```

Nótese que el algoritmo para buscar primos de los lenguajes C, C++ y el de Java son muy similares salvo la declaración de la función inicial y el arreglo que contendrá a los primos. Esto deja patente la cercanía entre dichos lenguajes y por que en este trabajo los presentamos en forma conjunta.

Ejemplo en Python:

```
def criba_Eratostenes(N):
  p = [] # inicializa el arreglo de primos encontrados
  # Guarda los primeros 2 primos
  p.append(2)
  p.append(3)
  np = 2
  # Empieza la busqueda de primos a partir del numero 4
  n = 4
  #Ciclo para buscar los primeros N primos
  while np < N:
    xi = 0
    for i in p:[22]
      xi = xi + 1
      if (n % i) == 0:
        break
    if xi == np:
      p.append(n)
      np = np + 1
    n = n + 1
   # Visualiza los primos encontrados
```

[22] En este código, al usar el *for* sobre los elementos del arreglo de primos, es necesario usar un contador para saber si ya se recorrieron todos los primos existentes y así determinar si es un nuevo primo y agregarlo a la lista.

```
    print("Visualiza los primeros " + str(N) + " primos ")
    for i in range(np):
      print(p[i])
    return p
# Solicita el calculo de los primeros primos
P = criba_Eratostenes(1000)
print(P)
```

Otro ejemplo en Python:

```
def criba_Eratostenes(N):
    p = [] # inicializa el arreglo de primos encontrados
    # Guarda los primeros 2 primos
    p.append(2)
    p.append(3)
    np = 2
    # Empieza la busqueda de primos a partir del numero 4
    n = 4
    #Ciclo para buscar los primeros N primos
    while np < N:
      for i in range(np):[23]
        if (n % p[i]) == 0:
          break
      if i == np-1:
        p.append(n)
        np = np + 1
      n = n + 1
     # Visualiza los primos encontrados
    print("Visualiza los primeros " + str(N) + " primos ")
    for i in range(np):
      print(p[i])
    return p
# Solicita el calculo de los primeros primos
P = criba_Eratostenes(1000)
print(P)
```

[23] Aquí, se hace el recorrido sobre el arreglo de primos usando la indexación sobre sus elementos, usando el número de elementos que se tiene mediante el uso del *for* con un *range*.

## 2.5 Introducción a los Paradigmas de Programación

La construcción de programas de cómputo -Software- puede involucrar elementos de gran complejidad, que en muchos casos no son tan evidentes como los que se pueden ver en otras Ciencias e Ingenierías. Un avión, una mina, un edificio, una red de ferrocarriles son ejemplos de sistemas complejos de otras Ingenierías, pero el programador construye sistemas cuya complejidad puede parecer que permanece oculta. El usuario siempre supone que en informática todo es muy fácil -"apretar un botón y ya esta"-.

Cuando se inicia uno en la programación, es común el uso de pequeños ejemplos, generalmente se programa usando una estructura secuencial -una instrucción sigue a la otra- en programas cortos[24], cuando los ejemplos crecen se empieza a usar programación estructurada[25] -que usa funciones- para después proseguir con ejemplos más complejos haciendo uso de la programación orientada a objetos -uso de clases y objetos- o formulaciones híbridas de las anteriores[26].

A continuación delinearemos estos paradigmas de programación:

**Programación Secuencial** es un paradigma de programación en la que una instrucción del código sigue a otra en secuencia también conocido como código espagueti. Las tareas se suceden de tal modo que la salida de una es la entrada de la siguiente y así sucesivamente hasta el fin del programa.

[24] Los ejemplos no son complejos, suelen estar construidos y mantenidos por una o pocas personas, son códigos de cientos o miles de líneas y tienen un ciclo de vida corto. Además, se puede construir aplicaciones alternativas en un período razonable de tiempo y no se necesitan grandes esfuerzos en anális y diseño.

[25] Al crecer la complejidad del Software a desarrollar, es muy difícil o imposible que un desarrollador o un grupo pequeño de ellos pueda comprender todas las sutilidades de su diseño, para paliar los problemas que conlleva el desarrollo de grandes y complejos sistemas informáticos surge la programación orientada a objetos.

[26] El surgimiento de la programación orientada objetos trata de lidiar con una gran cantidad de requisitos que compiten entre sí, incluso contradiciéndose, tienen desacoplamientos de impedancias entre usuarios del sistema y desarrolladores y es común la modificación de los requisitos con el paso del tiempo pues los usuarios y desarrolladores comienzan a compenetrarse mejor. Así, la programación orientada a objetos permite dirigir un equipo grande de desarrolladores, manejar una gran cantidad de código, usar estándares de desarrollo —al igual que en otras ingenierías— y verificar la fiabilidad de los estándares existentes en el mercado.

**Programación Estructurada** también llamada Procedimental, es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora recurriendo únicamente a subrutinas y tres estructuras básicas: secuencia, selección (if y switch) e iteración (bucles for y while); así mismo, se considera innecesario y contraproducente el uso de la instrucción de transferencia incondicional, que podría conducir a código espagueti, mucho más difícil de seguir y de mantener, y fuente de numerosos errores de programación.

Entre las ventajas de la programación estructurada sobre el modelo anterior -hoy llamado despectivamente código espagueti-, cabe citar las siguientes:

- Los programas son más fáciles de entender, pueden ser leídos de forma secuencial y no hay necesidad de tener que rastrear saltos de líneas dentro de los bloques de código para intentar entender la lógica interna.
- La estructura de los programas es clara, puesto que las instrucciones están más ligadas o relacionadas entre sí.
- Se optimiza el esfuerzo en las fases de pruebas y depuración. El seguimiento de los fallos o errores del programa (debugging), y con él su detección y corrección se facilita enormemente.
- Se reducen los costos de mantenimiento. Análogamente a la depuración, durante la fase de mantenimiento, modificar o extender los programas resulta más fácil.
- Los programas son más sencillos y más rápidos de confeccionar.
- Se incrementa el rendimiento de los programadores.

**Programación Orientada a Objetos** (POO, u OOP según sus siglas en inglés) es un paradigma de programación que viene a innovar la forma de obtener resultados. El surgimiento de la programación orientada objetos trata de lidiar con una gran cantidad de requisitos que compiten entre sí, incluso contradiciéndose, tienen desacoplamientos de impedancias entre usuarios del sistema y desarrolladores y es común la modificación de los requisitos con el paso del tiempo pues los usuarios y desarrolladores comienzan a compenetrarse mejor. Así, la programación orientada a objetos permite

dirigir un equipo grande de desarrolladores, manejar una gran cantidad de código, usar estándares de desarrollo -al igual que en otras ingenierías- y verificar la fiabilidad de los estándares existentes en el mercado.

Los objetos manipulan los datos de entrada para la obtención de datos de salida específicos, donde cada objeto ofrece una funcionalidad especial. Los objetos son entidades que tienen un determinado «estado», «comportamiento (método)» e «identidad»:

- La identidad es una propiedad de un objeto que lo diferencia del resto; dicho con otras palabras, es su identificador -concepto análogo al de identificador de una variable o una constante-.

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

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

La programación orientada a objetos difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. En la programación estructurada solo se escriben funciones que procesan

datos. Los programadores que emplean POO, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos.

**Conceptos fundamentales** La programación orientada a objetos es una forma de programar que trata de encontrar solución a los problemas que genera el desarrollo de proyectos de tamaño mediano o grande y/o complejos. Introduce nuevos conceptos, que superan y amplían conceptos antiguos ya conocidos. Entre ellos destacan los siguientes:

- Clase: se puede definir de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ella.
- Herencia: Por ejemplo, la herencia de la clase C a la clase D, es la facilidad mediante la cual la clase D hereda en ella cada uno de los atributos y operaciones de C, como si esos atributos y operaciones hubiesen sido definidos por la misma D. Por lo tanto, puede usar los mismos métodos y variables registrados como «públicos (public)» en C. Los componentes registrados como «privados (private)» también se heredan pero se mantienen escondidos al programador y sólo pueden ser accedidos a través de otros métodos públicos. Para poder acceder a un atributo u operación de una clase en cualquiera de sus subclases pero mantenerla oculta para otras clases es necesario registrar los componentes como «protegidos (protected)», de esta manera serán visibles en C y en D pero no en otras clases.
- Objeto: La instancia de una clase. Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos), los mismos que consecuentemente reaccionan a eventos. Se corresponden con los objetos reales del mundo que nos rodea, o con objetos internos del sistema (del programa).
- Método: Es un algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un «mensaje». Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un «evento» con un nuevo mensaje para otro objeto del sistema.

- Evento: Es un suceso en el sistema -tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto-. El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento la reacción que puede desencadenar un objeto; es decir, la acción que genera.

- Atributos: Características que tiene la clase.

- Mensaje: Una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

- Propiedad o atributo: Contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera del objeto y esto se define como sus características predeterminadas, y cuyo valor puede ser alterado por la ejecución de algún método.

- Estado interno: Es una variable que se declara privada, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos). No es visible al programador que maneja una instancia de la clase.

- Componentes de un objeto: Atributos, identidad, relaciones y métodos.

- Identificación de un objeto: Un objeto se representa por medio de una tabla o entidad que esté compuesta por sus atributos y funciones correspondientes.

Por ejemplo, definamos a la clase Persona y un objeto p de esa clase en el lenguaje Java:

```
public class Persona {
  private String nombre;
  private String apellidos;
  private int edad;
  public Persona(String nombre, String apellidos, int edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
```

```
    }
    public String getNombre() {
      return nombre;
    }
    public String getApellidos() {
      return apellidos;
    }
    public int getEdad() {
      return edad;
    }
    public static void main(String [ ] Args) {
      Persona p = new Persona ("Antonio", "Carrillo", 50);
    }
}
```

Y hacemos lo mismo pero en el lenguaje Python:

```
class Persona:
  def __init__(self, nombre, apellidos, edad):
    self.Nombre = nombre
    self.Apellidos = apellidos
    self.Edad = edad
  def nombre(self):
    return self.Nombre
  def apellidos(self):
    return self.Apellidos
  def edad(self):
    return self.Edad
if __name__ == "__main__":
  p = Persona("Antonio", "Carrillo", 50)
```

Y ahora haremos uso de la herencia para definir la clase Profesor que es heredada de la clase persona, en Java:

```
public class Profesor extends Persona {
  private String idProfesor;
  public Profesor(String nombre, String apellidos, int edad,
String idProfesor) {
```

```
        super(nombre, apellidos, edad);
        this.idProfesor = idProfesor;
      }
      public void setIdProfesor(String idProfesor) {
        this.idProfesor = idProfesor;
      }
      public String getIdProfesor() {
        return idProfesor;
      }
      public static void main(String [ ] Args) {
        Profesor p = new Profesor("Antonio", "Carrillo", 50, "Prof
3289239823");
      }
    }
```

Y en Python:

```
class Profesor(Persona):
  def __init__(self, nombre, apellidos, edad, identificador):
    Persona.__init__(self, nombre, apellidos, edad)
    self.Identificador = identificador
  def identificador(self):
  return self.Identificador
p = Profesor("Antonio", "Carrillo", 50, "Prof 3289239823")
```

**Características de la POO** Existe un acuerdo acerca de qué características contempla la «orientación a objetos». Las características siguientes son las más importantes:

- Abstracción: Denota las características esenciales de un objeto, donde se capturan sus comportamientos. Cada objeto en el sistema sirve como modelo de un «agente» abstracto que puede realizar trabajo, informar y cambiar su estado, y «comunicarse» con otros objetos en el sistema sin revelar «cómo» se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos, y, cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción. El proceso de abstracción permite seleccionar las

características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevos tipos de entidades en el mundo real. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad o el problema que se quiere atacar.

- Encapsulamiento: Significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión (diseño estructurado) de los componentes del sistema.

- Polimorfismo: Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O, dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en «tiempo de ejecución» esta última característica se llama asignación tardía o asignación dinámica.

- Herencia: Las clases no se encuentran aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento, permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos en clases y estas en árboles o enrejados que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple; siendo de alta complejidad técnica por lo cual suele recurrirse a la herencia virtual para evitar la duplicación de datos.

- Modularidad: Se denomina «modularidad» a la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las partes restantes. Estos módulos se pueden

compilar por separado, pero tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan la modularidad de diversas formas.

- Principio de ocultación: Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una «interfaz» a otros objetos que específica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas; solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no puedan cambiar el estado interno de un objeto de manera inesperada, eliminando efectos secundarios e interacciones inesperadas.

- Recolección de basura: La recolección de basura (garbage collection) es la técnica por la cual el entorno de objetos se encarga de destruir automáticamente, y por tanto desvincular la memoria asociada, los objetos que hayan quedado sin ninguna referencia a ellos. Esto significa que el programador no debe preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un nuevo objeto y la liberará cuando nadie lo esté usando.

Muchos de los objetos prediseñados de los lenguajes de programación actuales permiten la agrupación en bibliotecas o librerías, sin embargo, muchos de estos lenguajes permiten al usuario la creación de sus propias bibliotecas. Está basada en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

**Recursión o Recursividad** Un concepto que siempre le cuesta bastante a los programadores que están empezando es el de recursión o recursividad (se puede decir de las dos maneras). La recursividad consiste en funciones que se llaman a sí mismas, evitando el uso de bucles y otros iteradores.

Uno ejemplo fácil de ver y que se usa a menudo es el cálculo del factorial de un número entero. El factorial de un número se define como ese número multiplicado por el anterior, éste por el anterior, y así sucesivamente hasta llegar a 1. Así, por ejemplo, el factorial del número 5 sería: 5x4x3x2x1 = 120.

Tomando el factorial como base para un ejemplo, ¿cómo podemos crear una función que calcule el factorial de un número? Bueno, existen multitud de formas. La más obvia quizá sería simplemente usar un bucle determinado para hacerlo, algo así en C:

```
long factorial(int n){
  long res = 1;
  for(int i=n; i>=1; i- -) res = res * i;
  return res;
}
```

Sin embargo hay otra forma de hacerlo sin necesidad de usar ninguna estructura de bucle que es mediante recursividad. Esta versión de la función hace exactamente lo mismo, pero es más corta, más simple y más elegante:

```
long factorial(int n)
{
   long fact;
   if (n <= 1) return 1;
   return n*factorial(n-1);
}
```

Aquí lo que se hace es que la función se llama a sí misma (eso es recursividad), y deja de llamarse cuando se cumple la condición de parada (en este caso que el argumento sea menor o igual que 1 que es lo que hay en el condicional).

**Ventajas e inconvenientes** ¿Ganamos algo al utilizar recursión en lugar de bucles/iteradores para casos como este?

En este caso concreto del cálculo factorial no, y de hecho es una forma más lenta de hacerlo y ocupa más memoria. Esto no es preocupante en la realidad, pero conviene saberlo. Lo del factorial es solo una forma de explicarlo con un ejemplo sencillo y que sea fácil de entender.

Pero entonces, si no ganamos nada en este caso ¿para qué sirve realmente la recursividad?

Pues para resolver ciertos problemas de manera elegante y eficiente. El ejemplo típico sería el recorrer un árbol de elementos para hacer algo con todos ellos. Imagínate un sistema de archivos con carpetas y subcarpetas

y archivos dentro de estas carpetas, o el árbol de elementos de una página Web donde unos elementos incluyen a su vez otros y no sabes cuántos hay en cada uno. En este tipo de situaciones la manera más eficiente de hacer una función que recorra todos los elementos es mediante recursión. Es decir, se crea una función que recorra todos los elementos hijo del nodo que se le pase y que se llame a sí misma para hacer lo mismo con los subnodos que haya. En el caso del sistema de archivos se le pasaría una carpeta y se llamaría a sí misma por cada subcarpeta que hubiese, y así sucesivamente con todas las demás. Con una sola llamada inicial recorrerá automáticamente toda la estructura del sistema de archivos.

Con eso, y sin necesidad de complicarse, de repente se tiene una función muy poderosa capaz de enumerar cualquier estructura arbitraria por compleja que sea. Ahí es donde se ve el verdadero poder de la recursividad, aunque hay aplicaciones más potentes y más complejas todavía.

**Detalles a Tener en Cuenta** Otra cosa importante a tener en cuenta es que, cada vez que se hace una llamada a una función desde otra función (aunque sea a sí misma), se crea una nueva entrada en la pila de llamadas del intérprete. Esta tiene un espacio limitado por lo que puede llegar un punto en el que si se hacen demasiadas se sature y se produzca un error. A este error se le denomina "Desbordamiento de pila" o "Stack Overflow". Ahora ya sabemos de donde viene el nombre del famoso sitio para dudas de programadores sin el que la programación moderna no sería posible.

Además, hay que tener mucho cuidado con la condición de parada. Esta se refiere a la condición que se debe comprobar para determinar que ya no se harán más llamadas a la función. Es en ese momento en el que empiezan a devolverse los valores hacia "arriba", retornando a la llamada original.

Si no tienes la condición de parada controlada pueden pasar varias cosas (todas malas), como por ejemplo:

- Que se sature la pila y se produzca un desbordamiento
- Que se ocupe cada vez más memoria
- Que se produzcan desbordamientos de variables al ir acumulando resultados.

## 2.6 Errores de Redondeo y de Aritmética

La aritmética que realiza una computadora es distinta de la aritmética de nuestros cursos de álgebra o cálculo. En nuestro mundo matemático tradicional consideramos la existencia de números con una cantidad infinita de cifras, en la computadora cada número representable tienen sólo un número finito, fijo de cifras (véase 2.4), los cuales en la mayoría de los casos es satisfactoria y se aprueba sin más, aunque a veces esta discrepancia puede generar problemas.

Un ejemplo de este hecho lo tenemos en el cálculo de raíces de:

$$ax^2 + bx + c = 0$$

cuando $a \neq 0$, donde las raíces se calculan comúnmente con el algoritmo:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ y } x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

o de forma alternativa con el algoritmo que se obtiene mediante la racionalización del numerador:

$$x_1 = \frac{-2c}{b + \sqrt{b^2 - 4ac}} \text{ y } x_2 = \frac{-2c}{b - \sqrt{b^2 - 4ac}}$$

Otro algoritmo que implementaremos es el método de Newton-Raphson[27] en su forma iterativa:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

en el cual se usa $x_0$ como una primera aproximación a la raíz buscada y $x_n$ es la aproximación a la raíz después de $n$ iteraciones (sí se converge a ella), donde $f(x) = ax^2 + bx + c$ y $f'(x_i) = 2ax + b$.

[27] También podemos usar otros métodos, como el de Newton Raphson Modificado para acelerar la convergencia

$$x_{i+1} = x_i - \frac{f(x_i)f'(x_i)}{[f'(x_i)]^2 - f(x_i)f''(x_i)}$$

que involucra la función $f(x)$, la primera derivada $f'(x)$ y a la segunda derivada $f''(x)$.

**Salida del Cálculo de Raíces** La implementación computacional que se puede consultar en la página del texto, se muestra en los programas desarrollados en distintos lenguajes (C, C++, Java y Python) usando diferentes paradigmas de programación (secuencial, procedimental y orientada a objetos), todos ellos generan la siguiente salida:

```
Polinomio (1.000000) X^2 + (4.000000 )X + (1.000000) = 0

Chicharronera 1
Raiz (-0.2679491924311228), evaluacion raiz: -4.4408920985006262e-16
Raiz (-3.7320508075688772), evaluacion raiz: 0.0000000000000000e+00

Chicharronera 2
Raiz (-0.2679491924311227), evaluacion raiz: 0.0000000000000000e+00
Raiz (-3.7320508075688759), evaluacion raiz: -5.3290705182007514e-15

Metodo Newton-Raphson
Valor inicial aproximado de X1 = -1.2679491924311228
Raiz (-0.2679491924311227), evaluacion raiz: 0.0000000000000000e+00
Valor inicial aproximado de X2 = -4.7320508075688759
Raiz (-3.7320508075688772), evaluacion raiz: 0.0000000000000000e+00
```

En esta salida se muestra la raíz calculada y su evaluación en la ecuación cuadrática, la cual debería de ser cero al ser una raíz, pero esto no ocurre en general por los errores de redondeo. Además, nótese el impacto de seleccionar el algoritmo numérico adecuado a los objetivos que persigamos en la solución del problema planteado.

En cuanto a la implementación computacional, el paradigma de programación seleccionado depende la complejidad del algoritmo a implementar y si necesitamos reusar el código generado o no.

Si lo que necesitamos implementar computacionalmente es una fórmula o conjunto de ellas que generen un código de decenas de líneas, la implementación secuencial es suficiente, si es menor a una centena de líneas puede ser mejor opción la implementación procedimental y si el proyecto es grande o complejo, seguramente se optará por la programación orientada a objetos o formulaciones híbridas de las anteriores.

En última instancia, lo que se persigue en la programación es generar un código: correcto, claro, eficiente, de fácil uso y mantenimiento, que sea flexible, reusable y en su caso portable.

**Ejemplo en C Usando Programación Estructurada:** Escribimos el código siguiente en cualquier editor y lo llamamos *cuadratica.c:*

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

// Funcion cuadratica
double f(double x, double a, double b, double c) {
  return (x * x * a + x * b + c);
}

// Derivada de la funcion cuadratica
double df(double x, double a, double b) {
  return (2.0 * x * a + b);
}

// Evalua el valor X en la función cuadratica
void evalua(double x, double a, double b, double c) {
  printf("\nRaiz (%1.16f), evaluacion raiz: %1.16e", x, (a * x * x
+ b * x + c));
}

// Metodo Newton-Raphson
// x = x - f(x)/f'(x)
double metodoNewtonRapson(double x, int ni, double a, double
b, double c) {
   int i;
  for (i = 0; i < ni; i++) {
    x = x - (f(x, a, b, c) / df(x, a, b));
  }
  return x;
}

// Funcion Principal ....
int main() {
   // Coeficientes del polinomio
   double A = 1.0, B = 4.0, C = 1.0;
   // Raices del polinomio
```

```
        double X1, X2, x;
        // Calculo del discriminante
        double d = B * B - 4.0 * A * C;
        // Raices reales
        if (d >= 0.0) {
          printf("\nPolinomio (%f) X^2 + (%f )X + (%f) = 0\n", A,
B, C);
          printf("\nChicharronera 1");
          X1 = (-B + sqrt(d)) / (2.0 * A);
          X2 = (-B - sqrt(d)) / (2.0 * A);
          evalua(X1, A, B, C);
          evalua(X2, A, B, C);
          printf("\n\nChicharronera 2");
          X1 = (-2.0 * C) / (B + sqrt(d));
          X2 = (-2.0 * C) / (B - sqrt(d));;
          evalua(X1, A, B, C);
          evalua(X2, A, B, C);
          // Metodo Newton-Raphson
          printf("\n\nMetodo Newton-Raphson");
          x = X1 - 1.0;
          printf("\nValor inicial aproximado de X1 = %f", x);
          x = metodoNewtonRapson(x, 6, A, B, C);
          evalua(x, A, B, C);
          x = X2 - 1.0;
          printf("\nValor inicial aproximado de X2 = %f", x);
          x = metodoNewtonRapson(x, 6, A, B, C);
          evalua(x, A, B, C);
          printf("\n");
        } else {
          // Raices complejas
         printf("Raices Complejas ...");
        }
        return 0;
    }
```

**Ejemplo en C++ Usando Programación Orientada a Objetos** Escribimos el código siguiente en cualquier editor y lo llamamos *cuadratica.cpp:*

```
#include <stdio.h>
#include <math.h>
#include<iostream>
using namespace std;
class cuadratica {
 private:
    // Coeficientes del polinomio
    double A, B, C;

  // Funcion cuadratica
  double f(double x, double a, double b, double c) {
    return (x * x * a + x * b + c);
  }

  // Derivada de la funcion cuadratica
  double df(double x, double a, double b) {
    return (2.0 * x * a + b);
  }

  // Evalua el valor X en la función cuadratica
  void evalua(double x, double a, double b, double c) {
    printf("\nRaiz (%1.16f), evaluacion raiz: %1.16e", x, (a * x
* x + b * x + c));
  }

  // Metodo Newton-Raphson
  // x = x - f(x)/f'(x)
  double metodoNewtonRapson(double x, int ni, double a, double
b, double c) {
    int i;
    for (i = 0; i < ni; i++) {
      x = x - (f(x, a, b, c) / df(x, a, b));
    }
    return x;
  }
```

```
    public:
      // Constructor de la clase
      cuadratica(double a, double b, double c) {
        A = a;
        B = b;
        C = c;
      }
      // calculo de raices
      void raices() {
        // Raices del polinomio
        double X1, X2, x;
        // Calculo del discriminante
        double d = B * B - 4.0 * A * C;

        // Raices reales
        if (d >= 0.0) {
          printf("\nPolinomio (%f) X^2 + (%f )X + (%f) = 0\n",
A, B, C);
          printf("\nChicharronera 1");
          X1 = (-B + sqrt(d)) / (2.0 * A);
          X2 = (-B - sqrt(d)) / (2.0 * A);
          evalua(X1, A, B, C);
          evalua(X2, A, B, C);

          printf("\n\nChicharronera 2");
          X1 = (-2.0 * C) / (B + sqrt(d));
          X2 = (-2.0 * C) / (B - sqrt(d));;
          evalua(X1, A, B, C);
          evalua(X2, A, B, C);
          // Metodo Newton-Raphson
          printf("\n\nMetodo Newton-Raphson");
          x = X1 - 1.0;
          printf("\nValor inicial aproximado de X1 = %f", x);
          x = metodoNewtonRapson(x, 6, A, B, C);
          evalua(x, A, B, C);
          x = X2 - 1.0;
          printf("\nValor inicial aproximado de X2 = %f", x);
```

```
        x = metodoNewtonRapson(x, 6, A, B, C);
        evalua(x, A, B, C);
        printf("\n");
      } else {
        // Raices complejas
        printf("Raices Complejas ...");
      }
    }
};

int main(void)
{
    cuadratica cu1 = cuadratica(1.0, 4.0, 1.0);
    cu1.raices();
    return 0;
}
```

**Ejemplo en Java Usando Programación Orientada a Objetos** Escribimos el código siguiente en cualquier editor y lo llamamos *cuadratica.java:*

```
import java.lang.Math;
public class cuadratica {
    // Coeficientes del polinomio
    double A, B, C, d;

    // Constructor de la clase
    public cuadratica(double a, double b, double c) {
        A = a;
        B = b;
        C = c;
    }

    // Funcion cuadratica
    public double f(double x) {
        return (x * x * A + x * B + C);
    }
```

```
        // Evalua el valor X en la función cuadratica
        public void evalua(double x) {
          System.out.println("Raiz (" + x + ") , evaluacion raiz:" +
    (A * x * x + B * x + C));
        }

        // Derivada de la funcion cuadratica
        public double df(double x) {
          return (2.0 * x * A + B);
        }

        // Metodo Newton-Raphson
        // x = x - f(x)/f'(x)
        public double metodoNewtonRapson(double x, int ni) {
          for (int i = 0; i < ni; i++) {
            x = x - (f(x) / df(x));
          }
          return x;
        }

        public void raices() {
          // Raices del polinomio
          double X1, X2, x;
          // Calculo del discriminante
          d = B * B - 4.0 * A * C;
          // Raices reales
          if (d >= 0.0) {
            System.out.println("");
            System.out.println("Raices Reales (" + A + ")X^2 + ("
    + B + ")X + (" + C + ") = 0");
            System.out.println("Chicharronera 1");
            X1 = (-B + Math.sqrt(d)) / (2.0 * A);
            X2 = (-B - Math.sqrt(d)) / (2.0 * A);
            evalua(X1);
            evalua(X2);
            System.out.println("");
            System.out.println("Chicharronera 2");
            X1 = (-2.0 * C) / (B + Math.sqrt(d));
```

```
                X2 = (-2.0 * C) / (B - Math.sqrt(d));;
                evalua(X1);
                evalua(X2);
                System.out.println("");
                  // Metodo Newton-Raphson
                System.out.println("Newton-Rapson");
                X1 = Math.round(X1);
                System.out.print("Valor inicial aproximado de X1 = " +
X1);
                x = metodoNewtonRapson(X1, 8);
                evalua(x);
                X2 = Math.round(X2);
                System.out.print("Valor inicial aproximado de X2 = " +
X2);
                x = metodoNewtonRapson(X2, 8);
                evalua(x);
              } else {
                // Raices complejas
                System.out.println("Raices Complejas ...");
              }
            }

            // Funcion Principal ....
              public static void main(String[] args) {
              cuadratica cu1 = new cuadratica(1.0, 4.0, 1.0);
              cu1.raices();
            }
        }
```

**Ejemplo en Python Usando Programación Orientada a Objetos**
Escribimos el código siguiente en cualquier editor y lo llamamos *cuadratica.py:*

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
import math
class Ejemplo:
  # Constuctor
  def __init__(self, a, b ,c):
    self.A = a;
    self.B = b;
    self.C = c;
  # Funcion cuadratica
  def f(self, x, a, b, c):
    return (x * x * a + x * b + c);
  # Derivada de la funcion cuadratica
  def df(self, x, a, b):
    return (2.0 * x * a + b);
  # Evalua el valor X en la funcion cuadratica
  def evalua(self, x, a, b, c):
    print('Raiz (%1.16f), evaluacion raiz: %1.16e' % (x, (a * x *
x + b * x + c)))
  # Metodo Newton-Raphson
  # x = x - f(x)/f'(x)
  def metodo_Newton_Rapson(self, x, ni, a, b, c):

    for i in range(ni):
      x = x - (self.f(x, a, b, c) / self.df(x, a, b))
    return x
  # Calculo de raices
  def raices(self):
    # Calculo del discriminante
    d = self.B * self.B - 4.0 * self.A * self.C

    # Raices reales
    if d >= 0.0:
```

```
            print('\nPolinomio (%f) X^2 + (%f )X + (%f) = 0\n' %
(self.A, self.B, self.C))
            print('\nChicharronera 1')
            X1 = (-self.B + math.sqrt(d)) / (2.0 * self.A)
            X2 = (-self.B - math.sqrt(d)) / (2.0 * self.A)
            self.evalua(X1, self.A, self.B, self.C)
            self.evalua(X2, self.A, self.B, self.C)
            print('\nChicharronera 2')
            X1 = (-2.0 * self.C) / (self.B + math.sqrt(d))
            X2 = (-2.0 * self.C) / (self.B - math.sqrt(d))
            self.evalua(X1, self.A, self.B, self.C)
            self.evalua(X2, self.A, self.B, self.C)
            # Metodo Newton-Raphson
            print("\n\nMetodo Newton-Raphson")
            x = X1 - 1.0;
            print("\nValor inicial aproximado de X1 = %1.16f" % x)
            x = self.metodo_Newton_Rapson(x, 6, self.A, self.B, self.C)
            self.evalua(x, self.A, self.B, self.C);
            x = X2 - 1.0;
            print("\nValor inicial aproximado de X2 = %1.16f" % x);
            x = self.metodo_Newton_Rapson(x, 6, self.A, self.B, self.C)
            self.evalua(x, self.A, self.B, self.C)
            print("\n")
        else:
            # Raices complejas
            print("Raices Complejas ...")

if __name__ == '__main__':
    ej = Ejemplo(1.0, 4.0, 1.0)
    ej.raices()
```

## 2.7 Documentación del Código Fuente

La documentación dentro del código fuente (véase 5.7.1) tiene como objetivo que los que lo lean, mantengan y reparen el código, lo entiendan. Así que la documentación debe tener los siguientes propósitos:

- Explicar el propósito de cada clase y comportamiento, aunque pueda parecer obvia para tí, puede no ser tan obvia para otras personas.
- Describir los parámetros que espera cada comportamiento, los valores de retorno, y las excepciones que puede lanzar.
- Si la clase o los métodos están fuertemente acoplados con otra clase o forma de llamado, es necesario mencionarlo.
- Los comentarios deben indicar: El **por qué** (razones) y **cómo funciona el código**.
- Las cadenas de documentación deben indicar: **Cómo usar** el código.

Tener cadenas de documentación o comentarios incorrectos es mucho peor que no tenerlos en absoluto. Por ello es nuestro menester mantenerlos actualizados cando se hagan cambios, asegurándonos de que los comentarios y las cadenas de documentación continúan siendo consistentes con el código, y no lo contradicen. Algunas reglas son:

1. Los comentarios no deben duplicar el código.
2. Los buenos comentarios no excusan el código poco claro.
3. Si no puede escribir un comentario claro, es posible que haya un problema con el código.
4. Los comentarios deben disipar la confusión, no causarla.
5. Explique el código unidiomático en los comentarios.
6. Proporcione enlaces a la fuente original del código copiado.
7. Incluya enlaces a referencias externas donde sean más útiles.
8. Agregue comentarios al corregir errores.

9. Utilice comentarios para marcar implementaciones incompletas.

La documentación básica en C, C++ y Java se realiza usando:

- // para comentar dentro de una línea de código:

```
for (int i = 0; i < 10; i ++) // Este ciclo se realiza 10 veces
```

- /* y */ para comentar una o más líneas:

```
/*
  Este ciclo se realiza 10 veces
*/
for (int i = 0; i < 10; i ++) xp += 10 + i;
```

La documentación básica en Python se realiza usando:

- # para comentar dentro de una línea de código:

```
for i in Array: # Este ciclo se realiza tantas veces como elementos en Array
```

- """ y """ para comentar una o más líneas:

```
"""
  Este ciclo se realiza tantas veces como elementos en Array
"""
for i in Array:
  xp = xp + i
```

Además, si se realiza la documentación con cierta estructura, esta se puede utilizar para generar un manual de referencia del código en formatos: *HTML, PDF, PS, o XML* a partir de los fuentes con unos cuantos comandos de texto en unos segundos, pues qué mejor.

Existen varias herramientas para ello, una de ellas es DOXYGEN para códigos de Java, Fortran, C y C++, en Python se puede usar *Docstring* o una cadena de documentación como se verá en la siguientes secciones.

**Instalación de Doxygen** Para instalar DOXYGEN usar:

```
# apt install doxygen graphviz
```

una vez instalada, hay que generar el archivo de configuración de DOXYGEN, para ello usar:

```
$ doxygen -g
```

de aquí podemos editar el archivo Doxyfile generado según las necesidades de la documentación, un ejemplo de dicha configuración para generar la salida en *HTML, LaTeX y XML* esta en:
Jerarquia de Clases DOXYGEN
Para generar la documentación de los fuentes en la carpeta donde este el archivo de configuración y los archivos fuentes, usar:

```
$ doxygen
```

La documentación generada con DOXYGEN se mostrará en carpetas separadas para cada una de las salidas seleccionadas por ejemplo: *HTML, LaTeX, XML*, etc.
Para ver la documentación generada, usar en la consola:

```
$ cd html
$ xpdf index.html
```

Para generar la documentación en formato *PDF* a partir de la salida de *LaTeX* usar:

```
$ cd latex
$ make pdf
$ xpdf refman.pdf
```

en este caso se supone que se tiene instalado *LaTeX* en la máquina, en caso contrario podemos instalar lo básico usando:

```
# apt install science-typesetting texlive-science
```

y adicionalmente, si se requieren otras opciones instalamos:

```
# apt install texmaker texmacs texmacs-extra-fonts texlive-
  latex-base texlive-latex-recommended myspell-en-us myspell-es
```

### 2.7.1 Documentar en C, C++ y Java

Hay varios estilos de documentación (véase 5.7.1), aquí ponemos una que es fácil de usar para códigos en C++, pero es lógicamente extensible a lenguajes como Java.

```
#ifndef __test__
#define __test__
/// Descripcion breve de la clase.
/**
 * Descripcion detallada de la clase ...
 *
 * @author Antonio Carrillo
 * @date Winter 2010
 * @version 0.0.1
 * @bug No errors detected
 * @warning No warnings detected
 * @todo Exception handling
 */
class test
{
 private:

 /// Descripcion breve.
 const char *nmClass;
 /**
 * Descripcion corta.
 *
 * Descripcion larga ...
 *
 * 0 = Dirichlet, 1 = Neumann (or Robin)
 */
 int bdType;



 public:
 /**
```

```
* Descipcion breve.
*
* Descripcion detallada ...
*
* Algo de LaTeX ...
*
* \f[
* |I_2|=\left| \int_{0}^T \psi(t)
* \left\{
* u(a,t)-
* \int_{\gamma(t)}^a
* \frac{d\theta}{k(\theta,t)}
* \int_{a}^\theta c(\xi)u_t(\xi,t)\,d\xi
* \right\} dt
* \right|
* \f]
*
*
* @param[out] clas Descripcion del parametro de salida
* @param[in] fun Descripcion del parametro de entrada
*/
test(const char *clas, const char *fun)
{
nameClassFunct(clas, fun);
}

/**
* Descripcion breve.
*
* Descripcion detallada
*
* @param nVert Descripcion del parametro
* @param[in] g Descripcion del parametro
* @param[in] me Descripcion del parametro
* @param[out] values Descripcion del parametro
* @param z Descripcion del parametro
* @return Descripcion de lo que regresa
*/
```

```
    int eval(int nVert, double **g, StdElem *me, double ***values,
double *z);
  };
  /**
   * Descripcion breve de la clase.
   *
   * Descripcion detallada de la clase
   *
   * Otro parrafo de la descripcion ...
   *
   * Algo de formulas con LaTeX
   *
   * \f{eqnarray*}{
   * g &=& \frac{Gm_2}{r^2} \\
   * &=& \frac{(6.673 \times 10^{-11}\,\mbox{m}^3\,\mbox{kg}^{-
1}\,
   * \mbox{s}^{-2})(5.9736 \times 10^{24}\,\mbox{kg})}{(6371.01\,\mbox{km})^2}
\\
   * &=& 9.82066032\,\mbox{m/s}^2
   * \f}
   *
   *
   * Documentacion sobre la cual se basa la clase o archivo(s) que
hagan una descripcion de la
   * misma: Archivo.doc
   *
   * Descripcion breve del ejemplo de uso de esta clase (este archivo
se supone que estara en
   * una carpeta de nombre ./Examples en la posicion actual del
código)
   *
   * Algo de LaTeX
   *
   * La distancia entre \f$(x_1,y_1)\f$ and \f$(x_2,y_2)\f$ is
\f$\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}\f$.
   *
   * @example ExampleText.cpp
   */
```

```
#endif
```

Adicionalmente es deseable que algunos comportamientos o clases tengan información adicional como son: propósito, entradas, salidas, estructuras de datos usadas en entradas y salidas, dependencia de datos o ejecución, restricciones, etc., usando una estructura como la siguiente:

```
/**
* Proposito y Metodo:
* Entradas:
* Salidas:
* Entradas TDS:
* Salidas TDS:
* Dependencias:
* Restricciones y advertencias"
*/
```

Para controlar las versiones se podría usar algo como lo siguiente:

```
/**
* @file release.notes
* @brief Package TkrRecon
* @verbatim
* Coordinator: Leon Rochester
*
* v4r4p8 09-Mar-2002 LSR Remove GFxxxxx and SiRecObjs,
no longer used
* v4r4p7 07-Mar-2002 TU Mainly, add a combo vertexing to
the TkrRecon sequence
* @endverbatim
*/
```

Un ejemplo completo puede ser el siguiente:

```
#ifndef __ErrorControl__
#define __ErrorControl__
#include <new>
using namespace std;
#include <stdlib.h>
```

```
#include "Printf.hpp"
#ifdef USE_HYPRE
#include <mpi.h>
#endif
/// Error Control, this class handles errors for the system RESSIM
/**
* @author Antonio Carrillo and Gerardo Cisneros
* @date Winter 2010
* @version 0.0.2
* @verbatim
Coordinator: Robert Yates
v0.0.1 Junuary 2011 Antonio Carrillo generates the first version of the class
v0.0.2 March 2011 Gerardo Cisneros add HYPRE errors control
Inputs: Name of class and function
Outputs: Exit of program
TDS Inputs: none
TDS Outputs: none
Dependencies: #ifdef USE_HYPRE, MPI package
Restrictions and Caveats: Non exception handling still
@endverbatim
* @bug No errors detected
* @warning No warnings detected
* @todo Exception handling
*/
class ErrorControl {
private:
/// Name of class
const char *nmClass;
/// Name of function generating the error
const char *nmFunction;
public:
/**
* Class Constructor
*/
ErrorControl(void) {
```

```
    nameClassFunct(" ", " ");
    }
    /**
    * Class Constructor
    * @param clas Class name
    */
    ErrorControl(const char *clas) {
    nameClassFunct(clas, " ");
    }
    /**
    * Class Constructor
    * @param clas Class name
    * @param fun Name of function generating the error
    */
    ErrorControl(const char *clas, const char *fun) {
    nameClassFunct(clas, fun);
    }
    /**
    * Name of class and function
    * @param clas Class name
    * @param func Name of function generating the error
    */
    void nameClassFunct(const char * clas, const char *func) {
    nameClass(clas);
    nameFunct(func);
    }
    /**
    * No memory for this request
    * @param var Var name
    */
    void memoryError(const char * var) {
    Afprintf(stderr, "\n\nNo memory for %s request in %s of class
%s\n\n", var, nmFunction, nmClass);
    fatalError(1);
    }
    /**
    * No memory for this request
    * @param var Var name
```

```
    * @param i Index number
    */
    void memoryError(const char * var, int i) {
    Afprintf(stderr, "\n\nNo memory for %s request %d in %s of
  class %s\n\n", var, i, nmFunction, nmClass);
    fatalError(1);
    }
    /**
    * No memory for this request
    * @param var Var name
    * @param func Name of function generating the error
    */
    void memoryError(const char * var, const char *func) {
    Afprintf(stderr, "\n\nNo memory for %s request in %s of class
  %s\n\n", var, func, nmClass);
    fatalError(1);
    }
    /**
    * Fatal error.
    * @param cod Error code
    */
    void fatalError(int cod) {
    Afprintf(stderr, "\nFatal Error\nEnd program\n");
    #ifdef USE_HYPRE
    MPI_Abort(MPI_COMM_WORLD, cod);
    #else
    exit(cod);
    #endif
    }
    /**
    * Fatal error.
    * @param cod Error code
    */
    void fatalError(int cod, const char *txt) {
    Afprintf(stderr, txt);
    Afprintf(stderr, "\nFatal Error\nEnd program\n");
    #ifdef USE_HYPRE
    MPI_Abort(MPI_COMM_WORLD, cod);
```

```
#else
exit(cod);
#endif
}
/**
* Set name of class
* @param clas Class name
*/
void nameClass(const char *clas) {
nmClass = clas;
}
/**
* Set name of function
* @param func Function name
*/
void nameFunct(const char *func) {
nmFunction = func;
}
};
/**
* Error Control, this class handles errors for the system RESSIM
*
* Use of the class ErrorControl for error handling within the
system RESSIM,
* for example in the error control of memory request
*
* @example ExampleErrorControl.cpp
*/
#endif
```

Más detalles sobre los parámetros en la documentación del código fuente para ser usada por DOXYGEN se pueden ver en:
http://www.stack.nl/~dimitri/doxygen/commands.html#cmdparam

### 2.7.2 Documentar en Python

En Python el uso de acentos y caracteres extendidos esta soportado por la codificación UTF-8 (véase 5.7.1), para ello en las primeras líneas de código es necesario usar:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

siempre y cuando se use un editor o IDE que soporte dicha codificación, en caso contrario los caracteres se perderán.

Para documentar en Python se usa un *Docstring* o cadena de documentación, esta es una cadena de caracteres que se coloca como primer enunciado de un módulo, clase, método o función, y cuyo propósito es explicar su intención. Un ejemplo sencillo (en Python 3), es:

```
def promedio(a, b):
  """Calcula el promedio de dos numeros."""
  return (a + b) / 2
```

Un ejemplo más completo:

```
def formula_cuadratica(a, b, c):
  """Resuelve una ecuación cuadratica.
  Devuelve en una tupla las dos raices que resuelven la
  ecuacion cuadratica:

  ax^2 + bx + c = 0.
  Utiliza la formula general (tambien conocida
  coloquialmente como el "chicharronero").
  Parametros:
  a – coeficiente cuadratico (debe ser distinto de 0)
  b – coeficiente lineal
  c – termino independiente
  Excepciones:
  ValueError – Si (a == 0)
  """
if a == 0:
  raise ValueError(
    'Coeficiente cuadratico no debe ser 0.')
from cmath import sqrt
discriminante = b ** 2 - 4 * a * c
x1 = (-b + sqrt(discriminante)) / (2 * a)
x2 = (-b - sqrt(discriminante)) / (2 * a)
return (x1, x2)
```

La cadena de documentación en el segundo ejemplo es una cadena multilíneas, la cual comienza y termina con triples comillas (""""). Aquí se puede observar el uso de las convenciones establecidas en el PEP 257 (Python Enhancement Proposals):

- La primera línea de la cadena de documentación debe ser una línea de resumen terminada con un punto. Debe ser una breve descripción de la función que indica los efectos de esta como comando. La línea de resumen puede ser utilizada por herramientas automáticas de indexación; es importante que quepa en una sola línea y que este separada del resto del *docstring* por una línea en blanco.

- El resto de la cadena de documentación debe describir el comportamiento de la función, los valores que devuelve, las excepciones que arroja y cualquier otro detalle que consideremos relevante.

- Se recomienda dejar una línea en blanco antes de las triples comillas que cierran la cadena de documentación.

Todos los objetos documentables (módulos, clases, métodos y funciones) cuentan con un atributo $\_\_doc\_\_$ el cual contiene su respectivo comentario de documentación. A partir de los ejemplos anteriores podemos inspeccionar la documentación de las funciones *promedio* y *formula_cuadratica* desde el *shell* de Python:

```
>>> promedio.__doc__
'Calcula el promedio de dos numeros.'
>>> formula_cuadratica.__doc__
'Resuelve una ecuación cuadratica.\n\n Devuelve en una
tupla las dos raices que resuelven la\n ecuacion
cuadratica:\n \n ax^2 + bx + c = 0.\n\n
Utiliza la formula general (tambien conocida\n
coloquialmente como el "chicharronero").\n\n
Parametros:\n a – coeficiente cuadratico (debe ser
distinto de 0)\n b – coeficiente lineal\n
c – termino independiente\n\n Excepciones:\n
ValueError – Si (a == 0)\n \n '
```

Sin embargo, si se esta usando el *shell* de Python es mejor usar la función *help()*, dado que la salida producida queda formateada de manera más clara y conveniente:

```
>>> help(promedio)
Help on function promedio in module __main__:
promedio(a, b)
Calcula el promedio de dos numeros.
>>> help(formula_cuadratica)
Help on function formula_cuadratica in module __main__:
formula_cuadratica(a, b, c)
Resuelve una ecuacion cuadratica.
Devuelve en una tupla las dos raices que resuelven la
ecuacion cuadratica:
ax^2 + bx + c = 0.
Utiliza la formula general (tambien conocida
coloquialmente como el "chicharronero").
Parametros:
a – coeficiente cuadratico (debe ser distinto de 0)
b – coeficiente lineal
c – termino independiente
Excepciones:
ValueError – Si (a == 0)
```

Ciertas herramientas, por ejemplo *shells* o editores de código, pueden ayudar a visualizar de manera automática la información contenida en los comentarios de documentación. De esta manera un usuario puede tener a su alcance de manera sencilla toda la información que necesita para poder usar nuestras funciones.

**Generando documentación en páginas de HTML** Los *Docstrings* se pueden usar también para producir documentación en páginas de HTML que pueden ser consultadas usando un navegador de Web. Para ello se usa el comando *pydoc* desde una terminal. Por ejemplo, si las dos funciones anteriores (*promedio* y *formula_ cuadratica*) se encuentran en un archivo fuente llamado *ejemplos.py*, podemos ejecutar el siguiente comando en una terminal dentro del mismo directorio donde esta el archivo fuente:

```
pydoc -w ejemplos
```

La salida queda en el archivo *ejemplos.html*, y así se visualiza desde un navegador. La documentación de *pydoc* explica otros artilugios que puede hacer esta utilería de Python.

**Docstrings vs. Comentarios** Un comentario en Python inicia con el símbolo de número (#) y se extiende hasta el final de la línea. En principio los *Docstrings* pudieran parecer similares a los comentarios, pero hay una diferencia pragmática importante: los comentarios son ignorados por el ambiente de ejecución de Python y por herramientas como *pydoc*; esto no es así en el caso de los *Docstrings*.

A un nivel más fundamental hay otra diferencia aún más grande entre los *Docstrings* y los comentarios, y esta tiene que ver con la intención:

- Los *Docstrings* son documentación, y sirven para entender qué hace el código.
- Los comentarios sirven para explicar cómo lo hace.

La documentación es para la gente que usa tu código. Los comentarios son para la gente que necesita entender cómo funciona tu código, posiblemente para extenderlo o darle mantenimiento.

El uso de *Docstrings* en Python facilita la escritura de la documentación técnica de un programa. Escribir una buena documentación requiere de disciplina y tiempo, pero sus beneficios se cosechan cuando alguien -quizás mi futuro yo dentro de seis meses- necesita entender qué hacen nuestros programas. Los *Docstrings* no sustituyen otras buenas prácticas de programación, como son el uso apropiado de comentarios o el empleo de nombres descriptivos para variables y funciones.

## 2.8 Los Diagramas UML para Documentar

Todo el conocimiento está en el código fuente pero este no tiene una forma de fácil y rápida comprensión a alto nivel, para comprender un sistema es necesario una forma de documentación que muestre los detalles importantes de los que se compone el sistema. El lenguaje unificado de desarrollo o UML permite describir un sistema utilizando diferentes diagramas específicos para

mostrar diferentes aspectos del sistema. Hay muchas aplicaciones de Software libre disponible para diferentes sistemas operativos que permiten crear y exportar a imágenes los diferentes diagramas.

El trabajo de desarrollo de Software en mayor medida un trabajo colaborativo entre un grupo de personas, por ello la comunicación y transmisión de información es fundamental. Por otro lado, los sistemas informáticos de cierto tamaño son complejos de los cuales ninguna persona tiene el conocimiento completo y detallado de todas las partes que lo forman. Esto hace que la documentación y la transferencia de conocimiento sea importante en un grado similar al desarrollo de nuevas funcionalidades.

El problema de la documentación es que se quede obsoleta ni refleje el estado actual del sistema por los continuos cambios realizados en el código, para evitarlo la documentación ha de ser a alto nivel que describan los conceptos fundamentales del sistema que no serán propensos a cambiar. Otra forma de evitar esta obsolescencia en algunos casos es generar la documentación a partir de la información embebida en el código fuente como la documentación Javadoc de las clases y API de Java o la documentación de una API REST con Swagger. Sin embargo, no toda la documentación que aporta valor es posible generarla de forma automatizada.

El código fuente de una aplicación contiene el conocimiento más detallado y fiel de una aplicación , sin embargo, tratar de comprender un sistema complejo compuesto por unas cuantas decenas de miles de líneas de código aún usando un lenguaje de alto nivel ya sea Java , Python o C# es una tarea complicada o al menos que requiere mucho tiempo de investigación y esfuerzo de comprensión. Una breve descripción en prosa y un esquema del sistema permite obtener la misma información que analizando el código de forma mucho más rápida.

Si en una organización cuando una persona la abandona deja un vacío importante de conocimiento en las reglas de negocio implementadas o en la incorporación de una nueva persona a un equipo toma demasiado tiempo y dedicación adquirir el mismo conocimiento que el resto de miembros del equipo un problema de la organización quizá sea la falta de documentación. Con el paso del tiempo a medida que una aplicación es más compleja y dado que las personas en una organización se van con el conocimiento que posean y otras vienen pero necesitan adquirir conocimiento, la documentación es una forma de transferencia de conocimiento que está disponible de forma permanente y en forma de autoservicio.

Una buena documentación ha de ser concisa y transmitir su información

de forma rápida, una forma es utilizar diagramas que esquematizan el sujeto que se trata de describir complementados descripciones en texto. El lenguaje unificado de modelado o UML describe una notación estándar para los diagramas y varios tipos de diagramas.

**El lenguaje unificado de modelado o UML** estandariza las convenciones y elementos utilizados en los diagramas además de identificar y describir varios tipos de diagramas para describir un sistema desde varios puntos de vista. Cada uno de estos diagramas describen un sistema a alto nivel de forma esquematizada los conceptos fundamentales.

Los diagramas UML son de dos tipos estructurales que describen la parte estática del sistema y de comportamiento que describen la parte dinámica.

Diagramas UML estructurales:

- Diagrama de clases: en un lenguaje de programación orientado a objetos muestra las clases e interfaces por las que está compuesto el sistema y sus relaciones de herencia, composición e implementación con diferentes tipos de flechas. También muestra los atributos y métodos que poseen las clases.
- Diagrama de paquetes: los paquetes son agrupaciones lógicas de varias clases, en este diagrama se muestran los paquetes fundamentales asi como su relación y dependencia con otros.
- Diagrama de objetos: utilizan la misma notación que los diagramas de clases, se diferencian de los diagramas de clases en que muestran las instancias en un momento determinado del sistema.
- Diagrama de componentes: muestran los componentes fundamentales en los que se divide un sistema complejo y sus relaciones.
- Diagrama de estructura compuesta: muestra la estructura interna de una clase.
- Diagrama de despliegue: muestra en que elementos de computación se despliega el Software asi como sus relaciones. Son especialmente útiles en los sistemas distribuidos o basados en microservicios.

Diagramas UML de comportamiento:

- Diagrama de flujo o actividad: muestra la secuencia de acciones y decisiones implementadas en un algoritmo o proceso.
- Diagrama de secuencia: muestran las interacciones de los objetos entre si y el orden en el que se producen esas interacciones. El orden de las acciones se muestran de forma vertical y las interacciones como flechas.
- Diagrama de caso de uso: proporciona una visión general de los actores involucrados en un sistema, las diferentes funciones que usa esos actores y cómo interactúan estas diferentes funciones. Proporciona una descripción global del sistema inicial.
- Diagrama de estado: algunos objetos o el sistema se comportan de forma diferente en función del estado, estos diagramas muestran los diferentes estados de los que se compone y cual es su comportamiento en función del estado y condiciones.
- Diagrama de comunicación
- Diagrama de interacción: muestra un proceso al igual que un diagrama de actividad pero empleando una colección de diagramas de secuencia con su misma notación.
- Diagrama de tiempos

**Aplicación para crear diagramas de UML** para crear los diagramas de UML es necesario utilizar una aplicación, una orientada y específica para crear diagramas UML. El Software libre dispone de muchas, algunas en la red y otras están disponibles como aplicación Flatpak como forma de empaquetar una aplicación para cualquier distribución GNU/Linux, para Windows y macOS.

Hay muchas herramientas de este tipo de modelado que son multiplataforma, pero no tantas si queremos que sean libres, sencillamente gratuitas y que se puedan usar en Linux. Puede que para muchos no sean las mejores, pero yo empezaría con dos que crean los desarrolladores de dos de los escritorios Linux más usados, o gente relacionada a ellos:

- Umbrello
- Gaphor

- PlantUML
- diagrams.net (https://www.diagrams.net)