

Para evaluar las competencias de un lenguaje de programación a ser utilizado para resolver cierto problema, es necesario contar, entre otras cosas, con ciertos criterios a utilizar en el momento del análisis del lenguaje. Sin embargo, esos criterios a utilizar muchas veces difieren entre distintos autores y también entre usuarios del lenguaje. Porque a veces no es posible conseguir que 2 programadores estén de acuerdo en el valor de alguna característica de un lenguaje dado en relación con otro. A pesar de estas diferencias, la mayoría estaría de acuerdo que los criterios de elegibilidad, facilidad de escritura, confiabilidad y costo son los más relevantes. Como les decía, los lenguajes poseen ciertas características, algunas de las cuales influyen en estos criterios mencionados. Nosotros vamos a revisar estos criterios que hemos enunciado, las características más importantes que nos afectan y luego los resumiremos en una tabla.

### Criterios de Evaluación

Uno de los criterios más importantes a revisar en un lenguaje de programación es la elegibilidad. Es decir, que los programas escritos en un determinado lenguaje de programación puedan leerse y entenderse con facilidad. Antes de los años 70, el desarrollo del software fue pensado en términos de escribir código eficiente, más cercano a la máquina. En los años 70, como vimos en la unidad 1, se desarrolló el concepto de ciclo de vida del software y la codificación, podríamos decir, quedó relegada a un papel mucho más pequeño, cobrando mayor importancia la etapa del mantenimiento, particularmente en términos de costo. Dado que la facilidad de mantenimiento está principalmente determinada por la elegibilidad de los programas, es que la elegibilidad se volvió una medida importante para evaluar la calidad tanto del software como de los lenguajes de programación. La elegibilidad debe ser considerada en el contexto del dominio del problema. Como ya hemos dicho, si un programa que resuelve un determinado problema se escribiera en un lenguaje no diseñado para ese uso, el programa o la solución pueden resultar antinaturales y difíciles de leer, por lo tanto, difíciles de mantener. Las características que influyen en la elegibilidad son la simplicidad, la ortogonalidad, los tipos de datos y el diseño de sintaxis del lenguaje de programación. La simplicidad de un lenguaje de programación afecta fuertemente su legibilidad.

En primer lugar, un lenguaje que tiene muchos componentes básicos es más difícil de aprender que uno con pocos componentes básicos. Los programadores que trabajan con lenguajes de programación extensos, en general, tienden a aprender solo un subconjunto del lenguaje y pasan por alto otras características. En este caso, los problemas de elegibilidad ocurren cuando el autor del programa conoce un subconjunto de características del lenguaje de programación diferentes al que conoce quien se encarga del mantenimiento. Otra complicación de esta característica es la multiplicidad de expresiones.

**P.e. en Java:**

```
count = count + 1  
count += 1  
count++  
++ count
```

Es decir, el lenguaje tiene más de una manera de lograr una operación particular. Por ejemplo, si existen lenguajes que para incrementar una variable de tipo entero, proporcionan diferentes maneras de realizarlo, como ustedes ven ahí en el ejemplo de la diapositiva.

Y un tercer problema sería la sobrecarga de los operadores. Que, como ustedes ya saben, se produce cuando un símbolo, un solo símbolo de operador, tiene más de un significado. Si el lenguaje de programación, a su vez, permite a los usuarios crear su propia sobrecarga y los programadores no lo hacen apropiadamente, esto podría conducir a reducir la legibilidad del código. Como sabemos, el operador más está sobrecargado, pues se puede usar tanto para sumar enteros como para sumar reales. Esta sobrecarga simplifica un lenguaje de programación dado que reduce el número de operadores. Pero si, por ejemplo, el operador más está redefinido por el programador y es utilizado, por ejemplo, para sumar todos los elementos de los arreglos, si se utiliza en medio de operando de tipo arreglos multidimensionales, por ejemplo, esta situación haría el programa más confuso para su mantenimiento dado que, en general, el significado de la suma de matrices es bastante diferente matemáticamente hablando.

Ahora, por otro lado, el exceso de simplicidad disminuye la legibilidad.

Por ejemplo, la forma y significado de la mayoría de las sentencias de Assembler son modelos de simplicidad. Y como sus sentencias son simples, se requiere utilizar mucha más cantidad de estas sentencias que en los programas equivalentes en un lenguaje de alto nivel.

Otro concepto es el de la ortogonalidad. Este concepto de ortogonalidad en un lenguaje de programación significa que un conjunto relativamente pequeño de estructuras o de constructores primitivos pueden combinarse en un número relativamente pequeño de formas de construir estructuras de control y de datos del lenguaje. ¿Y dónde? Cada posible combinación de primitivas es legal y significativa.

Por ejemplo, supongamos un ejemplo, perdón, supongamos un lenguaje que tiene 3 tipos de datos primitivos, entero, punto flotante y char, y 2 operadores de tipo, array y puntero. Si los 2 operadores pueden aplicarse a ellos y a los 3 tipos de datos primitivos, se pueden definir un gran número de estructuras de datos. Pero si no se permitieran punteros para apuntar a los arreglos, muchas de esas

posibilidades se eliminarían. Si el lenguaje de programación permite todas las combinaciones posibles entre sus primitivas, hablamos de un lenguaje que cumple con esta característica.

La falta de ortogonalidad conduce a las excepciones, a las reglas del lenguaje. La ortogonalidad está estrechamente relacionada con la simplicidad. Cuanto más ortogonal es el diseño de un lenguaje de programación, se requieren menos excepciones a las reglas del lenguaje. Menos excepciones significan un mayor grado de regularidad en el diseño del lenguaje de programación y provoca que el lenguaje sea más fácil de aprender, leer y comprender. Demasiada ortogonalidad también puede causar problemas. Tal vez el lenguaje de programación más ortogonal sería el Gol 68, donde cada estructura del lenguaje tiene un tipo.


No hay restricciones en esos tipos y la mayoría de esas estructuras produce valores. Esta libertad para realizar combinaciones permite construir estructuras complejas. Esta forma extrema de ortogonalidad que puede resultar en un exceso de combinaciones conduce a una complejidad innecesaria.

Por otro lado, la falta de ortogonalidad se manifiesta con muchas excepciones a las reglas del lenguaje de programación. Podemos mencionar algunos ejemplos de falta de ortogonalidad en C, por ejemplo. Los registros pueden ser devueltos desde una función, pero los arreglos no. Un miembro de una estructura puede ser cualquier tipo de datos, excepto void o una estructura del mismo tipo. Un elemento del arreglo puede ser cualquier tipo de datos, excepto void o una función. Y los parámetros son pasados por copia de valor, a menos que ellos sean arreglos que se pasan por referencias, entre otros comportamientos del lenguaje.

Muchos consideran que los lenguajes funcionales brindan una buena combinación de simplicidad y ortogonalidad. Como ya hemos mencionado, en un lenguaje funcional como list, por ejemplo, los programas se construyen principalmente aplicando funciones a los parámetros dados. Mientras que en lenguajes imperativos como C, Pascal, y Java, por ejemplo, los programas se construyen básicamente con variables y sentencias de asignación. Por este motivo, los lenguajes funcionales ofrecen una mayor simplicidad porque pueden lograr todo con una sola estructura, la llamada a una función. Que, a su vez, también puede combinarse con otras llamadas de manera simple. Por lo tanto, podríamos decir que la simplicidad de un lenguaje de programación es, en parte, el resultado de una combinación de un número relativamente pequeño de estructuras primitivas y el uso limitado del concepto de ortogonalidad.


Pasamos ahora a ver otra característica del lenguaje que influye en la legibilidad y que son los tipos de datos. La presencia de estas características adecuadas para definir tipos y estructuras de datos en un lenguaje de programación son de gran ayuda para la legibilidad.

```

P.e. en C: 
main() {
    int z = 1;
    while (1) {
        printf("Z vale %d", z);
        z=z+1;
    }
}

```

```

P.e. en ADA: 
z : integer := 1;
begin
    while(true)
        loop
            put("Z vale ");
            put(z);
            z := z + 1;
        end loop;
end

```

Como ven en el ejemplo, es mucho más legible y tiene un significado absolutamente más claro utilizar como bandera a un tipo de datos booleano cuando el lenguaje de programación lo provee, antes que utilizar un tipo numérico porque no existe el tipo booleano en el lenguaje.

Otro ejemplo en cuanto a cómo los tipos de datos influyen en la legibilidad son los tipos de datos registro. Por ejemplo, los extracts de C que proveen una forma para representar datos más legibles que usar una serie de arreglos similares, uno para cada ítem de datos de un registro de alumnos, por ejemplo. Que sería la manera de resolver una estructura de este tipo en un lenguaje que no tiene manera o no provee el tipo de datos registro.

La otra característica que dijimos es el diseño de sintaxis. La sintaxis o forma, recuerdan, de los elementos de un lenguaje tiene un efecto significativo en la legibilidad de los programas. Los siguientes son dos ejemplos de lecciones de diseño de sintaxis que afectan la legibilidad.

La apariencia del programa y, por lo tanto, la legibilidad del mismo está fuertemente influenciada por las formas de las palabras especiales de un lenguaje. Por ejemplo, el begin y el end y el form. Como también por la formación de las sentencias compuestas o grupos de sentencias, principalmente en las estructuras de control. Ya hemos mencionado que varios lenguajes usan pares de palabras especiales o símbolos para formar los grupos. Pascal requiere, por ejemplo, el par begin-end para formar los grupos para todas las estructuras de control, excepto para la sentencia repeat, donde dicho par puede omitirse. Como ven en este ejemplo de Pascal, también lo podemos relacionar con una falta de ortogonalidad, como hablamos hace un momento. C usa llaves para encerrar sentencias compuestas de cualquier estructura. En ambos lenguajes se ve afectada la legibilidad, porque los grupos de sentencias siempre se terminan de la misma manera, lo que hace difícil determinar a qué grupo corresponde la finalización cuando un end o una llave aparece.

Ada resuelve este tema de una manera más clara, porque usa una sintaxis de cierre distinta para cada tipo de grupo de sentencia. Por ejemplo, usa endif para terminar una estructura de selección y end loop para terminar una estructura iterativa. Este es un ejemplo del conflicto entre simplicidad como resultado de una menor cantidad de palabras reservadas, como en Pascal o Java, y una legibilidad mayor que puede ser el resultado de usar más palabras reservadas, como lo hace Ada.

Otro problema es si las palabras especiales pueden usarse como nombres de variables o identificadores. Es decir, si el lenguaje de programación permite el uso de palabras claves. En ese caso, los programas

resultantes pueden ser muy confusos, dado que la aparición de una palabra reservada del lenguaje puede implicar una estructura sintáctica específica o no.

En cuanto a la forma y el significado, las sentencias en los lenguajes están diseñadas para que su apariencia indique al menos parcialmente cuál es su propósito, con lo cual favorece la legibilidad. En algunos casos, este principio se viola cuando dos estructuras del lenguaje que son idénticas o similares en apariencia, tienen significados diferentes dependiendo quizás del contexto. En C, por ejemplo, el significado de la palabra reservada `static` depende del contexto en que esta palabra aparece. Si se utiliza en la definición de una variable dentro de una función, significa que la variable se crea en tiempo de compilación. Mientras que si se usa en la definición de una variable que está fuera de todas las funciones, significa que la variable solo es visible en el archivo en que su definición aparece, es decir, no se exporta de ese archivo.

Otra cuestión que también afecta a la legibilidad es la longitud de los identificadores. Identificadores cuyos nombres están limitados a longitudes muy cortas disminuyen la legibilidad. Pero los lenguajes de programación actuales ya no presentan este problema que afecta fundamentalmente al mantenimiento del código.

El otro criterio de evaluación que anunciamos que se puede utilizar en el análisis de los lenguajes es la facilidad de escritura.

Este criterio indica la facilidad con que un lenguaje puede usarse para crear programas para un dominio de problema determinado. La mayoría de las características del lenguaje de programación que afectan la legibilidad también afectan la facilidad de escritura. En general, la afectan de forma inversa. Todos aquellos que afectan favorablemente a la legibilidad en general van en detrimento de la facilidad de escritura. Esto surge porque el proceso de escribir un programa frecuentemente le exige al programador que relea parte del programa que ya escribió.

La facilidad de escritura debe ser considerada en el contexto del objeto del dominio del problema de un lenguaje de programación. Es decir, que no es razonable comparar la facilidad de escritura de dos lenguajes de programación que pertenecen a los dominios de aplicaciones diferentes.

Las características de un lenguaje que afectan la facilidad de escritura, además de las que afectan la legibilidad, son el soporte para la abstracción y la expresividad. Las subdivisiones que vamos a ver a continuación hablarán de los factores más importantes que influyen en la facilidad de escritura.

En cuanto a la simplicidad y ortogonalidad, sí, la simplicidad y la ortogonalidad son las que más afectan a la facilidad de escritura. Si un lenguaje tiene un gran número de componentes básicos y demasiada ortogonalidad, esto disminuye la facilidad de escritura. Por lo tanto, un número más pequeño de estructuras primitivas y un conjunto consistente de reglas para combinarlas es mucho mejor que tener un gran número de primitivas. Con esto, un programador puede diseñar una solución a un problema complejo después de aprender solo un conjunto de estructuras primitivas. Por otro lado, demasiada ortogonalidad puede ir en detrimento de la facilidad de escritura, dado que errores en programas que se producen pueden no detectarse cuando cualquier combinación de primitivas es legal. Esto puede llevar a incoherencias en el código que no pueden ser descubiertas por el compilador.

En cuanto al soporte para la abstracción, como hemos mencionado a lo largo del desarrollo de la asignatura, sabemos que la abstracción es la habilidad para definir y usar estructuras u operaciones de manera que permitan que muchos de los detalles sean ignorados. El grado de abstracción permitido por un lenguaje de programación y la naturalidad de su expresión hacen al lenguaje de programación mucho más fácil de escribir, por lo que resulta también un concepto importante en el diseño de los lenguajes modernos.

Como ya hemos dicho, en la unidad 1, en la unidad 4 y en la unidad 6, los lenguajes de programación pueden dar soporte a 2 categorías de abstracción, la abstracción de procesos y la abstracción de datos. Un ejemplo de abstracción de procesos es el uso de un subprograma, por ejemplo, para implementar un algoritmo de ordenamiento que requiere ser utilizado varias veces en un programa. Sin el subprograma, el código del sort tendría que ser copiado en todos los lugares donde sea necesario utilizarlo, lo que haría el programa más extenso y más tedioso de escribir. Como ejemplo de una abstracción de datos, se podría considerar un árbol binario que guarda datos en sus nodos. En Fortran podríamos dar el ejemplo de que se implementaría usando 3 arreglos enteros paralelos, donde se usan 2 de los enteros como subíndice para especificar los nodos hijos. Mientras que en C++ o Java, un árbol binario puede implementarse usando una abstracción de un nodo del árbol en la forma de una clase con 2 punteros y un entero. La naturalidad de esta última representación usada por Java o C++ hace esto mucho más fácil de escribir que utilizando arreglos paralelos.

El soporte para la abstracción, como ven, es un factor importante que favorece la facilidad de escritura de un lenguaje de programación. La otra característica que afecta la facilidad de escritura es la expresividad. Que se refiere en un lenguaje de programación, la expresividad puede referirse a varias características diferentes. En un lenguaje como APL, por ejemplo, significa que hay operadores muy poderosos que permiten lograr mucho cálculo con un programa muy pequeño. Utilizando el operador reduce, por ejemplo, que se denota con una barra diagonal, su comportamiento reduce un arreglo a lo largo de un eje interponiendo la función de su operando. En otros lenguajes, la expresividad hace relativamente conveniente, ¿sí? En lugar de compleja, la forma de especificar los cálculos. Por ejemplo, en C, la anotación `count++` es más conveniente y más corta que si escribimos `count igual count más 1`. En ADA, el uso del operador booleano `and` es una manera conveniente de especificar la evaluación en cortocircuito de una expresión booleana. Mientras que en Java, la inclusión de la sentencia `for` hace la escritura de loops más fácil que con el uso del `while`. Estos ejemplos demuestran cómo la expresividad de un lenguaje de programación aumenta la facilidad de escritura del mismo.

El tercer criterio de evaluación que mencionamos es la confiabilidad. Tanto la elegibilidad como la facilidad de escritura influyen en la confiabilidad. Un programa escrito en un lenguaje que no admite formas naturales de expresar los algoritmos requeridos necesariamente utilizará enfoques no naturales para resolver un problema. Por lo tanto, es menos probable que esos enfoques no naturales sean correctos para todas las situaciones posibles. Cuanto más fácil es escribir un programa, más probabilidades hay de que ese código sea correcto.

La elegibilidad afecta la confiabilidad en las fases de escritura y mantenimiento del ciclo de vida del software. Los programas que son difíciles de leer son difíciles de escribir y modificar. Por lo tanto, las características que afectan la elegibilidad y la facilidad de escritura también afectan a la confiabilidad junto con el chequeo de tipos, el manejo de excepciones y el uso de alias.

Como vimos en la unidad 4, el chequeo de tipos es la búsqueda de errores de tipos en un programa dado, ya sea durante la compilación, lo que resulta más deseable porque los errores se descubren más tempranamente y facilitan las modificaciones requeridas, o durante la ejecución del programa que resulta caro en recursos de tiempo. El chequeo de tipos es un factor importante en la confiabilidad del lenguaje. En lenguajes como Java, los chequeos de tipos de todas las variables y expresiones se llevan a cabo en tiempo de compilación. Eso ya lo hemos mencionado. Ada tiene el mismo comportamiento, excepto cuando el usuario manifiesta explícitamente que los chequeos de tipos serán suspendidos. Mientras que la falta de chequeo de tipos en tiempo de compilación o en tiempo de ejecución ha llevado a muchos errores de programas en el uso de parámetros en sus programas, por ejemplo, en el lenguaje C original. En este lenguaje, el tipo de un parámetro actual en una llamada función, por ejemplo, no se chequea para determinar si este tipo coincide con el parámetro formal correspondiente a la función.

Una variable de tipo int o de tipo entero puede usarse como un parámetro actual en una llamada a una función que espera un tipo float como parámetro formal. Ni el compilador ni el runtime descubrirán la inconsistencia. En Pascal, por ejemplo, el rango del subíndice de una variable de un arreglo es parte del tipo de la variable. Por consiguiente, el chequeo del rango del subíndice es parte del tipo chequeado, aunque debe hacerse en tiempo de ejecución. Este tipo de chequeos es importante para la confiabilidad del programa porque los subíndices fuera de rango generalmente provocan errores que no son detectados. En el caso de Ada y Java, también exigen que todos los rangos de los subíndices sean chequeados.

En cuanto al manejo de excepciones, como vimos en la unidad 6, el manejo de excepciones es la habilidad de un programa de interceptar errores de ejecución, tomar las medidas correctivas y continuar. Esta característica es una gran ayuda a la confiabilidad. Ada, C++ y Sharp y Java incluyen construcciones para realizar manejo de excepciones producidas en ejecución. Pero esta característica es prácticamente inexistente en muchos lenguajes de programación ampliamente utilizados como, por ejemplo, C.

Y el último, la última característica que influye en la confiabilidad es el alias, que como vimos en la unidad 3, esto significa tener 2 o más métodos de referenciamiento o nombres para la misma celda de memoria, convirtiéndose en una característica peligrosa en un lenguaje de programación. Muchos lenguajes permiten algún tipo de alias, como por ejemplo C, con el uso del tipo de datos unión o bien con el uso de punteros que apuntan a la misma variable. En ambos casos, 2 variables diferentes de un programa pueden referirse a la misma celda de memoria. El diseño de ciertos lenguajes de programación prohíbe algunos tipos de alias. En otros lenguajes se utiliza para superar las deficiencias en las facilidades de abstracción de datos provista por el lenguaje y otros lenguajes restringen el alias para aumentar su confiabilidad.

		CRITERIOS		
CARACTERÍSTICAS		LEGIBILIDAD	FACILIDAD DE ESCRITURA	CONFIABILIDAD
	SIMPLICIDAD			
	ORTOGONALIDAD			
	TIPOS DE DATOS			
	DISEÑO DE SINTAXIS			
	SOPORTE PARA LA ABSTRACCIÓN			
	EXPRESIVIDAD			
	CHEQUEO DE TIPOS			
	MANEJO DE EXCEPCIONES			
	ALIAS			

Bueno, este cuadro resume las características de los lenguajes que inciden en los criterios más importantes considerados para la evaluación de un lenguaje de programación. Como dijimos, son la legibilidad, la facilidad de escritura y la confiabilidad.

El último criterio para considerar al evaluar un lenguaje de programación es el costo total del lenguaje que resulta ser una función de muchas de sus características. En primer término, está el costo de entrenar programadores para usar el lenguaje. Esta es una función de la simplicidad y ortogonalidad del lenguaje y la experiencia de los programadores.

En segundo lugar, el costo de escribir los programas en el lenguaje. Esta es una función de la facilidad de escritura del lenguaje que depende del grado de vinculación con el propósito de la aplicación en particular que se está desarrollando. Los primeros esfuerzos por diseñar e implementar lenguajes de alto nivel estuvieron motivados por el deseo de bajar los costos de desarrollo de software.

Tanto el costo de entrenar programadores como el costo de escribir programas en un lenguaje puede reducirse significativamente en un buen ambiente de programación, como lo vimos en la unidad 1.

También está el costo de compilación de los programas en el lenguaje. Como vimos también en la unidad 1, ADA tuvo una estandarización temprana, pero el impedimento mayor para su uso fue el alto costo de ejecutar la primera generación de compiladores de ADA. Este problema disminuyó por la aparición de buenos compiladores para el lenguaje.

En cuarto lugar, tenemos el costo de ejecución de programas escritos en un determinado lenguaje de programación, el que está fuertemente influenciado por el diseño de ese lenguaje. Un lenguaje que requiere muchos chequeos de tipos en tiempo de ejecución impedirá una ejecución rápida del código sin tener en cuenta la calidad del compilador.



Aunque la eficiencia de la ejecución fue la principal preocupación en el diseño de los primeros lenguajes, en la actualidad se considera que es menos importante. Ahora, existe una relación entre el costo de compilación y la velocidad de ejecución del código compilado. Optimización, como ustedes ya saben que lo vimos en la unidad 1, es el nombre que se le da al conjunto de métodos que los compiladores utilizan para disminuir el tamaño y aumentar la velocidad de la ejecución del código que se produce. Si se hace una pequeña o ninguna optimización, la compilación puede hacerse mucho más rápido, pero un esfuerzo extra de compilación produce una ejecución más rápida del código. La opción entre estas 2 alternativas está determinada por el entorno en el que se usará el compilador. Por ejemplo, si se utilizará en un ambiente de producción donde los programas completos se ejecutan muchas veces, es mucho más ventajoso pagar el costo extra de optimizar el código.

El quinto factor en el costo de un lenguaje es el costo del sistema de implementación de este lenguaje. Uno de los factores que explican la rápida aceptación de Java es que el sistema compilador intérprete estuvo disponible poco después que su diseño fue lanzado. Un lenguaje cuyo sistema de implementación es caro o solo corre en hardware que es caro, tendrá menos oportunidad de ser ampliamente utilizado.

En sexto lugar está el costo relacionado con la confiabilidad. Si un sistema de software falla en un sistema crítico, como puede ser un reactor nuclear, el costo podría ser muy alto. Los fracasos de sistemas no críticos también pueden ser muy caros en lo que se refiere a negocios perdidos o juicios sobre software que resultó defectuoso.

Y, por último, tenemos el costo de mantenimiento de programas que incluyen correcciones y modificaciones para agregar nuevas capacidades. El costo de mantenimiento del software depende de varias características del lenguaje, pero principalmente, como ya hemos dicho, de la legibilidad.

Porque, como ya dijimos, el mantenimiento generalmente es realizado por individuos que no son quienes escribieron el código originalmente y una pobre legibilidad puede hacer esta tarea sumamente engorrosa. La importancia del mantenimiento del software no puede tomarse, digamos, a la ligera. Se ha estimado que para los grandes sistemas de software con un tiempo de vida relativamente largo, los costos de mantenimiento pueden ser tan altos como 2 a 4 veces los costos de desarrollo.

De todos estos factores que contribuyen o que influyen en el costo del lenguaje, los más importantes son el desarrollo de programas, el mantenimiento y la confiabilidad. Y como estas son funciones tanto de la facilidad de escritura como de la legibilidad, estos 2 criterios de evaluación, a su vez, son los más importantes.

Obvio que existen otros criterios para evaluar los lenguajes de programación. Por ejemplo, uno es la portabilidad o la facilidad con que pueden trasladarse los programas entre distintos ambientes, tanto a plataformas de hardware como de un sistema operativo a otro. La portabilidad está fuertemente influenciada por el grado de estandarización del lenguaje.

Otros criterios también pueden ser la generalidad del lenguaje, es decir, la aplicabilidad del lenguaje a una amplia gama de aplicaciones, y la buena definición del lenguaje de programación, es decir, la integridad y precisión del documento que define oficialmente al lenguaje, entre otros criterios.

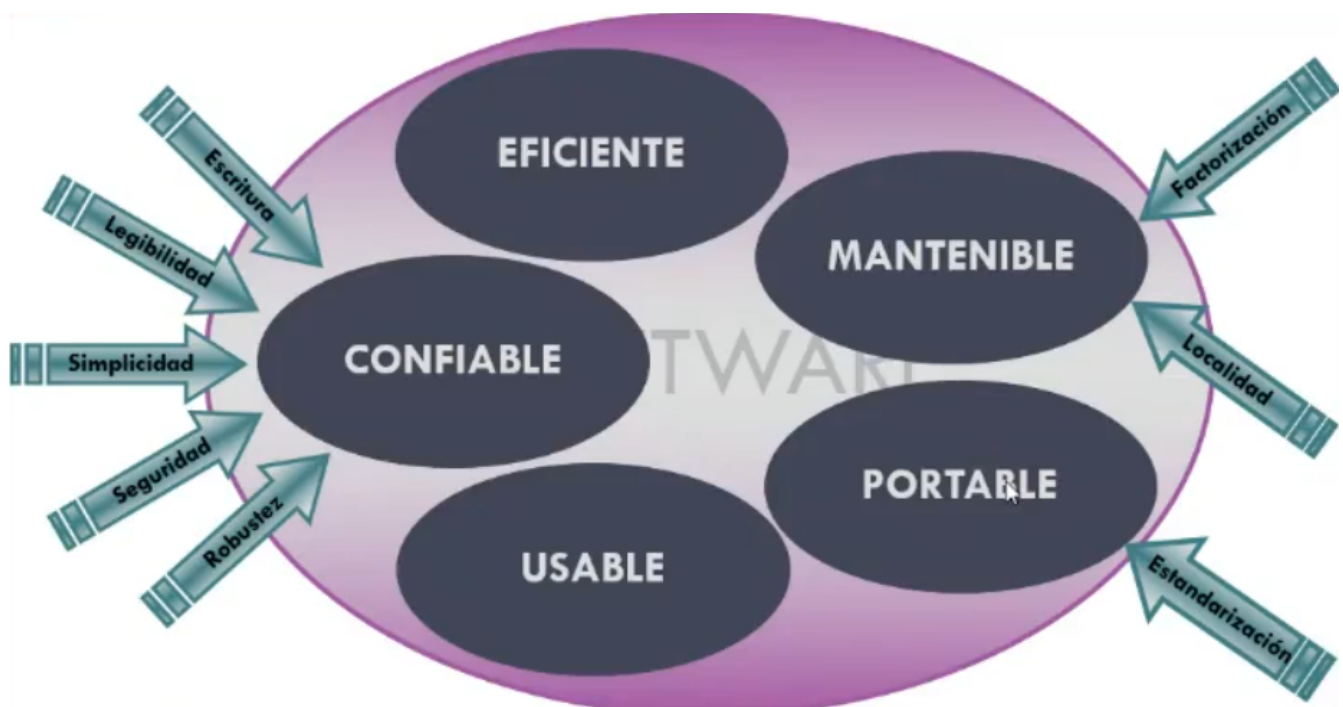
Muchos criterios, particularmente legibilidad, facilidad de escritura y confiabilidad, ni se definen de manera precisa ni son exactamente medibles. Sin embargo, ellos son conceptos útiles y proporcionan una visión en el diseño y evaluación de los lenguajes de programación.

Existen distintas perspectivas, digamos, que tienen que ver con los criterios de evaluación del lenguaje o cuando se va a realizar la evaluación de un lenguaje de programación. Los implementadores del lenguaje se preocupan principalmente por la dificultad de la implementación, tanto de las estructuras y características del lenguaje.

Mientras que el usuario del lenguaje, es decir, el programador, está preocupado primero por la facilidad de escritura y después por la legibilidad. Y es muy probable que los diseñadores del lenguaje pongan énfasis en la elegancia y en la habilidad de atraer su uso muy difundido.

En esta unidad nosotros hacemos fuerza o hacemos hincapié en la mirada del programador o de aquel profesional que decide el uso de un lenguaje u otro para resolver un determinado problema para un determinado dominio de aplicación.

Por último, vamos a hablar de los atributos de un lenguaje de programación en relación al software que produce. ¿Cómo se podrían definir las cualidades que debe presentar un lenguaje de programación?



Para responder a esta pregunta, debemos recordar que un lenguaje de programación es una herramienta que se utiliza para desarrollar software. Por lo tanto, la calidad del lenguaje debe estar relacionada con la calidad del software. Entonces, podríamos decir que el software debe ser confiable. Los usuarios deben poder confiar en el sistema. Es decir, que la probabilidad de fracaso debido a fallos y o fallas en el programa debe ser baja. En la medida de lo posible, el sistema debe ser tolerante a fallos. Es decir, debe continuar brindando soporte al usuario incluso en presencia de eventos poco frecuentes o no deseados, como fallas de hardware o de software. Este requisito de confiabilidad ha cobrado,

obviamente, mayor importancia a medida que se utiliza el software para realizar tareas críticas. Otro atributo es que el software debe ser mantenido. A medida que los costos del software han aumentado y se han desarrollado sistemas cada vez más complejos, no es económicamente factible descartar el software existente y desarrollar aplicaciones parecidas desde cero. Cuanto más tiempo se dedique a las etapas de análisis y diseño, menor será el costo de mantenimiento. Por lo tanto, los lenguajes de programación deben permitir que los programas sean fácilmente modificables. Otro atributo es que el software debe ejecutarse de manera eficiente. La eficiencia siempre ha sido un objetivo de cualquier sistema de software.

Este objetivo afecta tanto al lenguaje de programación, ¿sí? Con características que se pueden implementar de manera eficiente en las arquitecturas actuales, tanto como a la elección de los algoritmos que se utilizarán para resolver el problema. Aunque el costo del hardware continúa cayendo a medida que su rendimiento continúa aumentando en términos de velocidad y espacio, la necesidad de una ejecución eficiente permanece porque las computadoras se están utilizando en aplicaciones cada vez más exigentes. Otro atributo es que el software debe ser portable. Es decir, el software debe poder ejecutarse en distintos ambientes en cuanto a plataformas de hardware, como ambientes de software, como pueden ser un determinado sistema operativo. Debido a la proliferación de procesadores y sistemas operativos distintos, este atributo se ha transformado en un tema relevante dentro de las facilidades brindadas por los lenguajes de programación. Y, por último, el software debe ser usable. La usabilidad es muy importante en la calidad general del software que se produce porque hace que el sistema sea fácil de aprender y fácil de operar. Los 3 primeros requisitos, confiabilidad, capacidad de mantenimiento y eficiencia, se pueden lograr mediante la adopción de métodos adecuados de desarrollo de software. El uso de herramientas apropiadas y ciertas características del lenguaje de programación. Ahora vamos a ver si las características de los lenguajes que influyen directamente en estos atributos. El objetivo de confiabilidad está determinado por estas cualidades del lenguaje de programación. Facilidad de escritura, legibilidad, simplicidad, seguridad y robustez. Como hemos visto, la facilidad de escritura se refiere a la posibilidad de expresar un programa en una forma que sea natural para el problema. El programador no debe perder su atención en los detalles del lenguaje porque la actividad más importante es precisamente el problema a resolver.

Y aunque la facilidad de escritura es un criterio subjetivo, podríamos decir que los lenguajes de alto nivel son más fáciles de escribir que los lenguajes de bajo nivel. En cuanto a la legibilidad, debe ser posible seguir la lógica del programa y descubrir la presencia de errores revisando el código. La provisión de estructuras específicas para definir nuevas operaciones y nuevos tipos de datos que guardan la definición de dichos conceptos separados del resto del programa y que se pueden usar, mejora la legibilidad. En cuanto a la simplicidad, un lenguaje simple es fácil de dominar y permite expresar los algoritmos fácilmente y de alguna manera eso brinda seguridad al programador en la corrección del algoritmo. La simplicidad puede ser lograda minimizando las características de un lenguaje, sin embargo, esto puede llegar a reducir el poderío de ese lenguaje o el poder expresivo de ese lenguaje. Por ejemplo, Pascal es un lenguaje sencillo, pero es menos poderoso o con menos poder expresivo que C++. En cuanto a la seguridad, el lenguaje no debe proporcionar características que lo hagan posible describir programas defectuosos o peligrosos. Por ejemplo, un lenguaje que no proporciona la sentencia `goto` o variables de tipo puntero, eliminan 2 características muy conocidas de peligrosidad en un programa. Estas características pueden causar errores que son difíciles de rastrear

durante el desarrollo del programa y pueden manifestarse inesperadamente en el sistema de software entregado. En cuanto a la robustez, el lenguaje apoya esta característica siempre que proporcione la habilidad de tratar con eventos no deseados. Recuerden los desbordamientos de pila que ya hemos mencionado, las entradas no válidas, etcétera. Es decir, estos eventos que pueden atraparse y programarse una respuesta conveniente cuando una ocurrencia sucede. De esta manera, la conducta del sistema se vuelve predecible incluso en situaciones anómalas. En cuanto al mantenimiento del software, sí, si bien la legibilidad y la simplicidad son importantes en el mantenimiento, las 2 características principales que los lenguajes pueden proporcionar para apoyar la modificabilidad de un código son la factorización y la localidad.

La factorización ya la hemos mencionado en la unidad 6, ¿sí? El lenguaje debe permitir a los programadores factorizar las características relacionadas en una sola unidad. Si una operación se repite en varios puntos del programa, debe ser posible factorizarla en una rutina y reemplazar su uso por una llamada a dicha rutina. Haciendo esto, el programa se vuelve más legible, sobre todo si se dan nombres significativos a las rutinas, y más fácil de realizar es el mantenimiento, ¿sí? ¿Por qué? Porque un cambio en una porción de código se localiza en el cuerpo de esa rutina o de esa unidad.

En cuanto a la localidad, el efecto de una característica del lenguaje se restringe a una porción pequeña, local de todo un programa. Por otra parte, si se extiende a la mayor parte del programa, la tarea de hacer un cambio puede ser sumamente compleja. Por ejemplo, en programación, en cuanto a los datos abstractos, el cambio de una estructura de datos definida dentro de una clase debe garantizar que no afectará el resto del programa con tal que se invoquen las operaciones que manipulan la estructura de los datos de la misma manera. La factorización y la localidad son conceptos que están fuertemente relacionados. Porque, en realidad, factorizar origina localidad, donde los cambios solo pueden aplicarse a la porción de código factorizado.

En cuanto a la portabilidad, como ya hemos dicho, se refiere a la habilidad de un sistema de ser ejecutado en diferentes plataformas sin necesidad de realizar cambios importantes en los programas. Esta característica está fuertemente influenciada por el grado de estandarización del lenguaje de programación. Tal como vimos en la unidad 1, muchos lenguajes que nunca se estandarizaron hicieron que programas escritos en esos lenguajes sean muy difíciles de trasladar de una plataforma a otra.

En cuanto a la eficiencia, la necesidad de eficiencia ha regido el diseño de lenguajes desde los comienzos de la computación. Muchos lenguajes han tenido a la eficiencia como un objetivo principal de diseño, implícita o explícitamente. Pero el problema de la eficiencia ha cambiado considerablemente desde los primeros lenguajes de programación a la actualidad. Porque ya no solo es medida por la velocidad de ejecución y espacio de almacenamiento.

El esfuerzo requerido para producir un programa o iniciar un sistema y el esfuerzo requerido para el mantenimiento de dicho sistema, también pueden verse como componentes que permiten medir la eficiencia. En algunos casos, se prefiere la productividad del proceso de desarrollo de software que la performance o eficiencia del software resultante. Esto es, se podría estar interesado en componentes de software en procesos de desarrollo que son reutilizables en otras aplicaciones similares. O se podría estar interesado en desarrollos de software portables para hacerlo rápidamente disponible a distintos usuarios. La eficiencia es generalmente una cualidad combinada del lenguaje y su implementación. El

lenguaje afecta la eficiencia adversamente si deshabilita ciertas optimizaciones que pueden ser aplicadas por el compilador, como vimos cuando hablamos de costo. Mientras que la implementación afecta la eficiencia adversamente si no tiene en cuenta todas las oportunidades relacionadas con guardar espacio y mejorar la velocidad. En cuanto al lenguaje y usabilidad, la usabilidad es un atributo que no depende solamente del software producido, sino también de quién usa ese software. Es también un concepto clave de lo que se denomina Interacción Hombre Computadora o HCI por sus siglas en inglés. Donde, además de hacer que los sistemas sean fáciles de aprender y de usar, también se relaciona con el apoyo brindado a los usuarios durante sus interacciones con dichos sistemas. Al ser un atributo deseable para un sistema de software, tanto los diseñadores de sistemas como los programadores o desarrolladores, deben considerar aquellas características del lenguaje de programación que facilitan el desarrollo de un producto de software utilizable. Fundamentalmente, las relacionadas con facilidades para diseñar interfaces de usuario y las consanguíneas al manejo de los errores cometidos por el usuario mientras está interactuando con el sistema. Con esto terminamos lo que es la unidad 8 del programa de la materia. Y podríamos decir que en algún momento de cada proyecto de desarrollo de software, la selección de un lenguaje se convierte en una decisión sumamente importante. En general, los lenguajes muchas veces se seleccionan por moda, miedo al cambio y presiones comerciales, entre otras.

Los profesionales deberíamos basar estas decisiones en criterios técnicos y económicos relevantes que respondan a ciertos interrogantes como:

- ¿El lenguaje ayuda o dificulta las buenas prácticas de programación?
- ¿Hay un entorno de desarrollo integrado y, por ejemplo, de calidad disponible para el lenguaje?
- ¿El lenguaje proporciona tipos y operaciones asociadas que son adecuados para representar entidades en el área de aplicación relevante?
- ¿El lenguaje proporciona estructuras de control que son adecuadas para modelar el comportamiento de entidades en el área de aplicación relevante?
- ¿El lenguaje admite la descomposición de programas en unidades de programa?
- ¿El lenguaje facilita la reutilización de las unidades de programa?
- ¿El lenguaje está diseñado de tal manera que los errores de programación se pueden detectar y eliminar lo más rápido posible?
- ¿Se puede implementar el lenguaje de manera eficiente?
- ¿El lenguaje ayuda o dificulta la escritura del código?

Con las respuestas a estos interrogantes, y de acuerdo al problema y al dominio de aplicación en el que se va a utilizar ese lenguaje de programación, son las cuestiones que tenemos que tener en cuenta en el análisis y en la elección de un lenguaje de programación para un determinado proyecto.