2. Descripción del discurso: Diseño por contrato

Como bien hemos dicho anteriormente, **el uso del lenguaje natural tiende a la ambigüedad**, pues muchas veces se basa en el contexto. Como ejemplo habíamos descrito anteriormente la acción de "ver a una persona con unos prismáticos" ¿La persona que los portaba era el observador o el observado?

A la hora de describir el programa que se debe realizar, **ocurrirá lo mismo si nuestra descripción no es la adecuada**, si nuestra descripción es ambigua, si **existe más de una interpretación posible** a un mismo enunciado.

Por ejemplo, si nos solicitaran un programa que calculara el seno de x al cuadrado, existen dos posibles interpretaciones: $sin(x^2)$ o $sin^2 x$. **Necesitamos concretar nuestro lenguaje natural, describiendo con más precisión el procedimiento que debemos realizar**. Respectivamente, podrían describirse dichas opciones como "el seno de la función x al cuadrado" y "el cuadrado de la función seno de x"

Esta **descripción precisa del comportamiento esperado de un programa** se denomina **especificación**, y establece una importante **relación entre los datos de entrada permitidos y los datos de salida esperados**.

En este caso, se asume que x es un número real, pero ¿la medida del ángulo viene dada en grados o en radianes? No se ha descrito el programa adecuadamente, pues el enunciado sigue siendo ambiguo. Para evitar la ambigüedad, una especificación ideal debería contener una **descripción adecuada tanto de los datos de entrada**, p.ej. "Un número real x que expresa la medida de un ángulo en radianes" **como de los de salida**, p.ej "El cuadrado de la función seno de x" Siguiendo este ejemplo, la especificación de nuestro programa podría perfectamente ser: "El programa calcula el cuadrado de la función seno de x, siendo x un real que expresa la medida de un ángulo en radianes"

Cuando se nos pida diseñar un programa, en algunos casos se nos planteará un **enunciado** el cual es **ambiguo**, como en el caso anterior. Una adecuada especificación deberá servir para eliminar cualquier ambigüedad que pueda desprenderse del enunciado. **Si no se nos da la especificación o no es adecuada**, entonces se corre el riesgo de que el problema que se nos pida no sea el adecuado, por lo que entonces **podríamos recurrir a cualquiera de las posibles interpretaciones**, especificando cuál de ella se ha tomado.

Ahora bien, **si el contexto del enunciado es suficiente** como para inferir cuál es la interpretación más adecuada, entonces **nos basaremos en el contexto para diseñar nuestro programa**. En cuanto vayamos avanzando en este libro, nos encontraremos enunciados mucho más extensos, del tamaño de una página como mínimo, por lo que en este tipo de problemas radicará el contexto.

La especificación describe, en resumen, **qué es lo que hace el programa**, y no cómo. Para una misma especificación, existen muchos caminos diferentes para llegar a la misma conclusión, es decir, **cualquier algoritmo que se ajuste a las limitaciones de los**

datos de entrada permitidos y los datos de salida esperados, puede implementarse correctamente en un programa con dicha especificación.

Esto que parece una tontería, ya que el propio código del programa describe qué hace el programa y cómo lo hace, tiene gran importancia. **Una buena especificación hace comprensible el objetivo del programa** y por lo tanto, **sirve como documentación** de este. Esta también **ahorra costes de mantenimiento** al **facilitar la reutilización** del algoritmo.

Además, **facilita la verificación del programa** al tener instrucciones claras de qué hace el programa, qué valores recibe y qué valores emite, como si fuera una **función matemática** donde entran cosas y salen cosas, volviendo al símil de la **caja negra**. Si dado un **certificado o caso de prueba**, al introducir el valor de entrada en un programa, obtenemos el de salida esperado, **el programa es parcialmente correcto**. Si se certifican **todos los casos de prueba**, el programa es **totalmente correcto**.

Se podría decir que una **buena especificación** es aquella **descripción clara** (fácil de entender), **breve** (sin redundancias) **y precisa** (sin ambigüedad) **del quehacer de un programa**, además de tener un notable **equilibrio entre la concisión y la generalidad** (sin detalles superfluos pero sin omisión de detalles cruciales).

La especificación ha de ser entendida como un contrato. Tomemos como ejemplo cualquier electrodoméstico. Cuando compramos un producto siempre viene acompañado de un manual de instrucciones. En este manual se nos detalla qué se considera un uso correcto del aparato y las cláusulas de garantía.

En algunos casos, la especificación nos servirá para **restringir los casos posibles**. Esto puede suceder cuando, si no existiese ninguna condición para el dato de entrada, el **problema** que se nos plantea sería **muy costoso**. En otros casos ocurre que al establecer restricciones se **facilita el algoritmo** y al mismo no afectaría al uso normal del programa porque **se sabe que el usuario nunca va a realizar tal acción**. Por ejemplo, introducir una edad negativa.

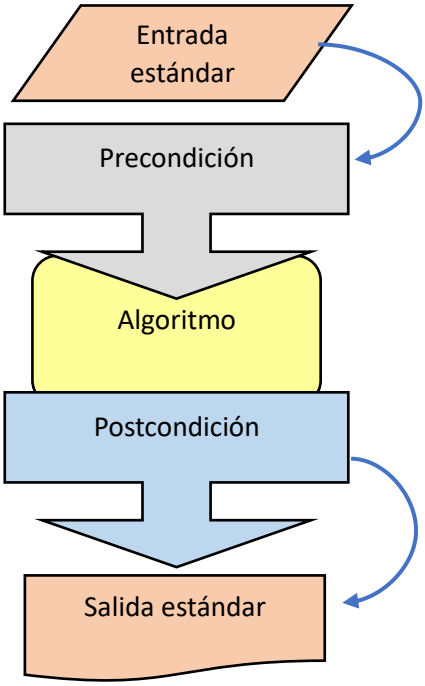
Cuando se diseña un producto **hay que tener en cuenta todos los casos de uso que se pueden presentar**; por ejemplo, alguien podría decidir conectar el aparato a una red de 300V cuando se indica explícitamente en el manual que no se puede conectar a corrientes superiores a 250V. **Las especificaciones** (en este caso, los modos de uso) **deben restringir los usos no previstos o estudiados** del aparato, deben indicar bajo qué condiciones se espera que sea utilizado. Así, si estas condiciones quedan claramente especificadas y es el usuario el que no las sigue, entonces la responsabilidad es del usuario, quedando el fabricante eximido de toda responsabilidad si el producto no funciona.

¿Cuál es el objetivo de la especificación en programación? Aplicar la misma idea, diseñar por contrato un programa.

El **diseño por contrato** fue diseñado por Bertrand Meyer, y lo aplicó a su propio lenguaje de programación, Eiffel. Este diseño abarca también ciertas metodologías y formalismos en la programación los cuales omitiremos, al ser este un curso de Bases de la Programación, resumiendo este diseño a lo más importante. A este diseño simplificado lo llamaremos **especificación pre-post**.

2.1 Especificación pre-post

Se denomina así porque esta especificación **sólo da la descripción de los datos de entrada permitidos y la de los de salida esperados**, aunque suficiente para diseñar un programa. La especificación se compone principalmente de seis partes:



- **Datos de entrada**: tipos de dato de los inputs y sus nombres
- **Entrada estándar**: nombre de los datos que se reciben por el teclado
- **Precondición**: condición de los datos de entrada que debe satisfacerse antes de la ejecución del algoritmo
- **Datos de salida**: tipos de dato de los outputs y sus nombres
- **Salida estándar**: nombre de los datos que se imprimen en pantalla
- **Postcondición**: condición de los datos de salida que debe satisfacerse justamente tras la ejecución del algoritmo

Por ejemplo, la especificación pre-post del ejemplo del capítulo anterior, del algoritmo que calculaba la media de tres trimestres podría ser:

- **Datos de entrada**: una secuencia de reales “calif”
- **Entrada estándar**: sus reales “calif”
- **Precondición**: 0.0 <= calif <= 10.0 y secuencia de longitud 3
- **Datos de salida**: un real “prom”
- **Salida estándar**: el real “prom”
- **Postcondición**: prom = la media de las tres calificaciones, 0.0 <= prom <= 10.0

Otra especificación en lenguaje natural no ambigua de este programa podría ser:

“Este programa lee por el teclado una secuencia de tres números reales comprendidos entre 0 y 10, y corresponden a la nota de cada trimestre que un alumno ha obtenido en la asignatura, e imprime en pantalla un número real que indica la media aritmética de las tres notas”

Por otra parte, en enunciados ambiguos como en el que se nos solicitaba calcular el seno de x al cuadrado, cualquiera de estas interpretaciones podrían ser especificaciones válidas:

a) Interpretación 1: el seno de la función x al cuadrado, x se mide en grados
   - **Datos de entrada**: un real “x”
   - **Entrada estándar**: no solicitada
   - **Precondición**: x está medido en grados, 0 <= x < 360
   - **Datos de salida**: un real “result”
   - **Salida estándar**: no solicitada
   - **Postcondición**: result = seno(x^2), -1 <= result <= 1

b) Interpretación 2: el seno de la función x al cuadrado, x se mide en radianes
   - **Datos de entrada**: un real “x”
   - **Entrada estándar**: no solicitada
   - **Precondición**: x está medido en radianes, 0 <= x < 2π
   - **Datos de salida**: un real “result”
   - **Salida estándar**: no solicitada
   - **Postcondición**: result = seno(x^2), -1 <= result <= 1

c) Interpretación 3: el cuadrado de la función seno de x, x se mide en grados
   - **Datos de entrada**: un real “x”
   - **Entrada estándar**: no solicitada
   - **Precondición**: x está medido en grados, 0 <= x < 360
   - **Datos de salida**: un real “result”
   - **Salida estándar**: no solicitada
   - **Postcondición**: result = (seno x)^2, result <= 1

d) Interpretación 4: el cuadrado de la función seno de x, x se mide en radianes
   - **Datos de entrada**: un real “x”
   - **Entrada estándar**: no solicitada
   - **Precondición**: x está medido en radianes, 0 <= x < 2π
   - **Datos de salida**: un real “result”
   - **Salida estándar**: no solicitada
   - **Postcondición**: result = (seno x)^2, result <= 1

Por lo tanto, si en este enunciado ambiguo no se da la especificación, a la hora de diseñar el programa, somos libres de elegir cualquiera de estas cuatro opciones siempre y cuando el contexto no sea suficiente como para determinar cuál es la interpretación más adecuada, indicaremos cuál es la descripción o especificación que se ha utilizado.

Uno de los tres conceptos que debemos tener en cuenta a la hora de especificar pre-post un programa es, la diferenciación clara entre tipo de dato y condición del dato.

Es muy importante saber diferenciar entre el tipo de dato y la condición que debe cumplir el dato. Un **tipo de dato** puede ser por ejemplo un booleano, un entero, un String, un vector de nº enteros, etc. algo que caracterice a los datos que se vayan a usar en el algoritmo y que formen parte de un **conjunto ya definido**, se detallará bastante más su definición en el siguiente tema; mientras que la **condición del dato** es una **característica muy específica** del dato **en un programa determinado** y que no se puede exactamente clasificar en un conjunto definido.

Por ejemplo, para un algoritmo que divide dos números X, Y cualquiera, se debería expresar en la precondición del programa {y /= 0}, como una característica específica de Y en la división de dos números sean naturales, enteros, reales, complejos, etc., siendo esto último dicho el tipo de dato que puede ser X, Y.

Si no fuera necesaria una precondición para el dato de entrada y se restringe solamente al tipo de dato que es este, se podría omitir la pre o decir más formalmente que la pre es {true}

También se debe tener en cuenta que **el tipo de dato que se utilice podría variar la descripción del algoritmo original el cual implementa otro tipo determinado**. Por ejemplo, la especificación del algoritmo del producto de dos números X e Y utilizando su definición matemática = > $x * y = x + x + x \dots ,\ y\ veces,\ x, y \in \mathbb{N}$ cuando estos son naturales en

contraposición con cuando estos son enteros, o bien manteniendo el algoritmo anterior o bien modificándolo para abarcar todos los posibles casos que dicho tipo de dato nos pueda ofrecer.

	Algoritmo original / Tipo de dato A	**Algoritmo original / Tipo de dato B**	**Algoritmo adaptado / Tipo de dato B**
Input	Dos naturales, X e Y	Dos enteros, X e Y	Dos enteros, X e Y
Output	Un natural Z	Un entero Z	Un entero Z
Pre	true	x,y >= 0	true
Post	Si x or y = 0, z=0 Si x and y /= 0, z = x + x + x… y veces	Si x or y = 0, z=0 Si x and y /= 0, z = x + x + x… y veces	Si x or y = 0, z=0 Si (x>0 and y>0) or (x<0 and y<0) , z = |x|+|x|+|x|… |y| veces Si (x>0 and y<0) or (x<0 and y>0) , z = -|x|-|x|-|x|… |y| veces
Result	Espec. original	Alteración de la pre	Alteración de la post

Observamos la especificación del programa con el tipo de dato A y la comparamos con la especificación del nuevo programa que usa el algoritmo original pero cambia el tipo de dato que se utiliza a B. Ahora X, Y son enteros, pero con la especificación en tipo A, el tipo de dato utilizado B es incompatible con la afirmación de la postcondición de A, puesto no existe el sumatorio cuando su nº de términos (y) a calcular de la sucesión (x) es negativo, es decir, no puedes sumar un número cualquiera -3 veces, carece de sentido. Por ello, para mantener el algoritmo original, debemos indicar qué tipos de enteros queremos, restringirlos a una característica específica de ellos, y por ello, tanto x como y deben ser mayores o iguales que 0, aunque eso es sinónimo de ser naturales, limitando los enteros a un conjunto ya definido y por lo tanto, se podría considerar este como tipo de dato. Luego, ¿es correcto precondicionar los enteros X, Y?¿Es correcto definir los naturales como un tipo de dato?

No entraremos en debate ahora en lo que puede o no definirse como tipo de dato, pero sí podemos afirmar que cualquier natural es un entero, pero no todos los enteros, como son los negativos, son naturales. Al considerar el tipo de dato utilizado como naturales, no existe ninguna característica específica que limite cuáles naturales hay que utilizar y cuáles no, pues todos se pueden utilizar y el algoritmo sigue siendo correcto. Esto no sucede en el caso de que se consideren enteros, pues existe una característica específica que limita cuáles enteros hay que utilizar y cuáles no, y es que como hemos citado antes, no podemos usar negativos, por lo que X, Y deben ser mayores o iguales que cero, dando una precondición de lo que debe cumplir el entero para que el algoritmo sea correcto.

Para que este algoritmo de producto de enteros sea más eficiente, debemos usar todos los números enteros existentes, añadiendo por supuestísimo los negativos y llegar al objetivo de poder multiplicar cualquier par de nº enteros sin ninguna precondición. Pero para ello, hemos tenido que modificar la postcondición de la especificación, y por tanto, adaptado el algoritmo a todos los posibles casos con el nuevo tipo de dato.

Una vez aclarada esta común confusión entre tipo de dato y condición del dato, pasemos a los otros dos puntos importantes a la hora de especificar pre-post.

El siguiente concepto se refiere al **nivel de restricción de las especificaciones**. Si las afirmaciones correspondientes a la pre o a la post son **más restrictivas**, se denominan **condiciones fuertes**. Sin embargo, cuando son más permisivas, se denomina **condiciones débiles**. P.ej. ser un número primo es una condición más fuerte que ser un número par.

Una de las principales tareas del programador a la hora de especificar un programa es convertir conjuntos de condiciones débiles en fuertes. **Cuanto más fuerte sea una condición, más seguro será el contrato**. Por ejemplo, estas conjunciones de condiciones débiles implican estas condiciones fuertes:

- x>=0 and x/=0 => x>0
- (x or y) and not(x and y) => x xor y
- N divisible entre 1 y N, no lo es para todos los nº entre 2 y N-1 => N es un nº primo

Como casos extremos, la afirmación {True} es la condición más débil posible mientras que la afirmación {False} es la condición más fuerte posible.

Otra característica más sobre las restricciones de condiciones a destacar es que **las restricciones de las variables son propagables**, de tal forma que si un término está acotado entre dos valores, cualquier variable compuesta por una combinación lineal de variables acotadas también lo será.

Por ejemplo, volviendo al problema de la media, si 0 <= calif <= 10, entonces la suma sum = calif1 + calif2 + calif3 se acota en 0+0+0 <= sum <= 10+10+10, 0 <= sum <= 30, y el promedio prom = sum/3, se acota en 0/3 <= prom <= 30/3, 0 <= prom <= 10, tal y como se indica en la especificación.

Finalmente, con respecto al **uso de las herramientas de E/S, se debe indicar su uso explícitamente** en la especificación del algoritmo. Si no se indica, se asume que los datos de entrada y de salida se leen y escriben en memoria. Se debe indicar correspondientemente que se recibe por **entrada estándar** o se **lee por el teclado** y que se emite por **salida estándar** o se **imprime en la pantalla**, respectivamente.

2.2 Casos de prueba

Para finalizar este bloque, en donde hemos sido capaces de diseñar algoritmos muy sencillos y describirlos, debemos ser capaces de argumentar por qué nuestro algoritmo es válido, por qué es correcto. La **corrección** es una propiedad intrínseca del algoritmo, y se puede definir como la **coincidencia entre el comportamiento real del programa y el del programa pretendido** o bien como la **coincidencia entre lo que se obtiene de valor salida al ejecutar el programa y lo que se debería obtener según la especificación** del programa.

Existen varios métodos para comprobar la corrección de un programa, entre los que destacan por ser muy utilizados métodos formales lógicos-matemáticos. Estos métodos requieren un conocimiento previo de sistemas formales de lógica, por lo que no los utilizaremos.

Uno de los métodos más comunes y básicos consiste en **dar distintos y variados valores a las variables de entrada y comprobar que se obtiene un valor de salida esperado de acuerdo con la especificación** o descripción del funcionamiento del programa. Cabe añadir que **los datos de entrada que introduzcamos al programa deben cumplir con su precondición y el resultado esperado, con su postcondición**.

Estos valores que nosotros le daremos a nuestro programa para comprobar el correcto funcionamiento del programa se denominan **certificados o casos de prueba**.

A la hora de crear casos de prueba **debemos procurar que los valores de los datos de entrada nos permitan cubrir todas las posibilidades que los valores de los datos de salida puedan abarcar**. Ciertos valores de entrada se podrían agrupar en distintos grupos en los cuales ese conjunto de datos en específico posee una serie de características comunes. Por ejemplo, para un algoritmo que multiplicara dos números enteros X e Y, tres grandes grupos podrían ser: pares con igual signo, pares con distinto signo y pares con al menos un cero. Estos grupos

también caracterizan a los datos de salida esperados. Así respectivamente, se esperarían para cada caso de prueba valores positivos, negativos y ceros.

Por ejemplo, si nuestro programa calcula el producto de dos enteros, podríamos definir la función característica de la siguiente manera:

$$x * y \begin{cases} 0 \; si \; x = 0 \vee y = 0 \\ \sum_{|y|} |x| \;\; si \; (x > 0 \wedge y > 0) \vee (x < 0 \wedge y < 0) \\ \sum_{|y|} -|x| \;\; si \; (x > 0 \wedge y < 0) \vee (x < 0 \wedge y > 0) \end{cases}$$

De esta forma, podemos construir una tabla de **casos de prueba significativos**. Un caso significativo es aquel que **representa a una agrupación en concreto de datos**. Por ejemplo, el par (6,4) es un caso significativo para el grupo "pares de igual signo, ambos factores positivos", cuyo resultado esperado es un número positivo. Si en una tabla de casos significativos encontramos por ejemplo (3,4) y (1,5), pares del mismo signo, ambos positivos; entonces dos casos de prueba estarán representando al mismo grupo anteriormente citado, lo que significa que uno de los casos de prueba no es significativo.

Por otra parte, los pares (3,4) y (-6, -2) simbolizan dos casos significativos distintos, puesto las características que caracterizan a estos pares son ligeramente distintas, pues aunque ambos pares son de igual signo, en uno ambos son positivos y en otro ambos son negativos.

Si construyéramos una **tabla de casos de prueba significativos**, resultaría de esta forma:

Entrada	Descripción del caso significativo	Salida
(5, 0)	Segundo factor nulo	0
(0, 3)	Primer factor nulo	0
(0, 0)	Ambos factores nulos	0
(1, 7)	Pares de igual signo, ambos factores positivos	7
(-2, -4)	Pares de igual signo, ambos factores negativos	8
(2, -3)	Pares de distinto signo, primer factor positivo, segundo factor negativo	-6
(-4, 5)	Pares de distinto signo, primer factor negativo, segundo factor positivo	-20

Para construir una tabla de casos significativos debemos por cada uno de nuestros casos **indicar los valores de entrada que se usan en nuestro ejemplo** de caso significativo, **hacer una breve descripción argumentando por qué este caso es significativo** y distinto a las demás agrupaciones posibles de datos, y **definir el valor de salida esperado tras la ejecución del programa**.

Hay que tener también en cuenta que **dependiendo de la fórmula empleada, existe un rango de valores que pueden llevarnos a errores de cálculo** puesto no podemos obtener un número de, por ejemplo, una división entre 0, **o errores relacionados con la asignación u obtención de un valor fuera de contexto**, como por ejemplo, intentar dar como dato de medida una longitud negativa.

Estos supuestos casos que se denominan **excepciones** no los trataremos en este curso, y asumiremos que el quien usa nuestro programa es consciente de que realizar este tipo de operaciones dará un error y corromperá el programa. De todas formas, **la precondición ya nos limita el rango de valores de entrada que podemos utilizar**, y **utilizar un valor fuera del rango de la pre exime de responsabilidad al programador sobre lo que se pudiera obtener de salida** y por tanto, no aporta importancia a la hora de verificarlo. Luego, es obligatorio que los valores de nuestros casos de prueba se encuentren en el rango de la pre.

Por ejemplo, en un programa que calcula dado un valor x la imagen de la función radical $f(x) = \sqrt{1-x}$ podemos intuir que esta función de variable real tiene un dominio determinado de valores, pues no puede tomar valores que hagan que el radicando sea un número negativo, ya que no existe la raíz cuadrada real de un número negativo.

Luego, la precondición del programa debe ser {1-x >= 0} o más bien {x <= 1}

Por lo tanto, a la hora de definir los grupos, destacaremos los valores pertenecientes al rango de la pre y los que no, pero nuestro único caso significativo será el valor de entrada elegido que cumple con su precondición. El resto de casos que no cumplen la condición de entrada, no son válidos. Dentro de los que sí la cumplen, podríamos clasificar los datos de entrada entre los negativos, el cero y los positivos.

Entrada	Descripción del caso significativo	Salida
-3	Números negativos	2
0	El cero	1
1	Números positivos hasta el 1	0
~~2~~	~~Resto de números positivos~~	~~i~~

Una vez definidos lo que son los casos de prueba y cómo se establecen, nos debemos cuestionar la siguiente pregunta. ¿Cómo puedo asegurarme con un alto nivel de certeza que mi programa es correcto? **Consideraremos que un programa es correcto si a la hora de ejecutarlo dando como parámetros de entrada los valores de entrada que le hemos dado a los casos de prueba significativos obtenemos como resultado la salida esperada según la tabla de casos**, y se repite este paso **para todos los casos significativos de la tabla**, asegurándose de que en todos se obtenga la salida esperada. Si en sólo uno de los casos no se obtiene el valor de salida esperado, el programa es incorrecto.