Tema 1. Abstracciones y Especificaciones

1.1. Introducción

1.2. Especificaciones informales

- 1.2.1. Abstracciones funcionales
- 1.2.2. Abstracciones de datos
- 1.2.3. Abstracciones de iteradores

1.3. Especificaciones formales

- 1.3.1. Método axiomático (o algebraico)
- 1.3.2. Método constructivo (u operacional)

1.1. Introducción

Abstraer: eliminar lo irrelevante y quedarnos con lo realmente importante.

¿Qué es lo importante?

***Abstracción por especificación:** sólo necesitamos conocer qué va a hacer la operación y no cómo funciona. (Encapsulación)

***Abstracción por parametrización:** un algoritmo, un tipo, o una variable se definen a través de unos parámetros. (Genericidad)

1.1. Introducción

Tipos de abstracciones

- Abstracciones funcionales → Rutinas, procedimientos.
- Abstracciones de datos → Tipos Abstractos de Datos.
- Abstracciones de iteradores → Iteradores.

Especificaciones: Tipos de notaciones

- Notaciones informales.
- Notaciones formales.
  - Algebraicas.
  - Operacionales.

1.1. Introducción

Diseño mediante abstracciones

1. Identificación de los subproblemas.
2. Especificación abstracta de cada uno de ellos.
3. Implementación de cada abstracción.
4. Verificación del programa.

1.2. Especificaciones informales

1.2.1. Abstracciones funcionales

Notación

Operación <nombre> (*ent* ident: tipo..., *sal* ident: tipo ...)

Requiere: establecimiento de restricciones de uso.

Modifica: identificación de los datos de entrada que se modifican.

Calcula: descripción textual del comportamiento de la operación.

1.2.1. Abstracciones funcionales

Ejemplo 1: eliminar la repetición en los elementos de un array.

Operación QuitarDuplic (*ent a*: array [entero])
Modifica: *a*
Calcula: quita los elementos repetidos de *a*. El límite inferior del array no varía, pero sí lo puede hacer el superior.

Ejemplo 2: concatenar dos cadenas.

Operación Concat (*ent a,b*: cadena; *sal c*: cadena)
Calcula: la cadena de salida *c* es una nueva cadena que contiene los caracteres de *a* (en el mismo orden) seguidos de los caracteres de *b* (en el mismo orden).

1.2.1. Abstracciones funcionales

Ejemplo 3: buscar un elemento en un array de enteros.

Operación Buscar (*ent a*: array [entero]; *x*: entero; *sal i*: entero)

Requiere: *a* debe estar ordenado de forma ascendente.

Calcula: si *x* está en *a*, entonces *i* debe contener el valor del índice de *x* tal que *a[i]* = *x*. Si *x* no está en *a*, entonces *i* = *sup*+1, donde *sup* es el índice superior del array *a*.

1.2.1. Abstracciones funcionales

Generalización: una operación está definida independientemente de cuál sea el tipo de sus parámetros.

Ejemplo 4: eliminar la repetición en los elementos de un array.

Operación QuitarDuplic [T: tipo](*ent a*: array [T])
Requiere: -T debe tener una operación de comparación
IgualQue(*ent* T, T; *sal* booleano).
Modifica: *a*
Calcula: quita los elementos repetidos de *a*. El límite inferior del array no varía, pero sí lo puede hacer el superior.

Ejemplo 5: buscar un elemento en un array de enteros.

Operación Buscar [T: tipo](*ent a*: array [T]; *x*: T; *sal i*: entero)
Requiere: -T debe tener dos operación de comparación
MenorQue(*ent* T, T; *sal* bool), Igual(*ent* T, T; *sal* bool).
-*a* debe estar ordenado de forma ascendente.
-T debe estar totalmente ordenado
Calcula: si *x* está en *a*, entonces *i* debe contener...

1.2. Especificaciones informales

1.2.2. Abstracciones de datos

Notación

TAD <nombre_tipo> es <lista_operaciones>

Descripción

Descripción textual del tipo

Operaciones

Especificación informal de las operaciones de la lista anterior

Fin <nombre_tipo>.

1.2.2. Abstracciones de datos

Ejemplo 6: TAD Conjunto de enteros

TAD CjtoEnteros *es* Vacío, Insertar, Suprimir, Miembro,
EsVacío, Unión, Intersección, Cardinalidad

Descripción

Los CjtoEnteros son conjuntos matemáticos modificables, que almacenan valores enteros.

Operaciones

Operación Vacío (*sal* CjtoEnteros)

Calcula: devuelve un conjunto de enteros vacío.

Operación Insertar (*ent c*: CjtoEnteros; *x*: entero)

Modifica: *c*.

Calcula: añade *x* a los elementos de *c*. Después de la inserción, el nuevo conjunto es *c* U {*x*}.

…

Fin CjtoEnteros.

1.2.2. Abstracciones de datos

TAD ListaEnteros *es* Crear, Insertar, Primero, Ultimo,
Cabeza,Cola, EsVacío, Igual

Descripción

Las ListaEnteros son listas de enteros modificables. Las listas se crean con las operaciones Crear e Insertar…

Operaciones

Operación Crear (*sal* ListaEnteros)

Calcula: Devuelve una lista de enteros vacía.

Operación Insertar (*ent l*: ListaEnteros; *x*: entero)

Modifica: *l*.

Calcula: Añade *x* a la lista *l* en la primera posición.

…

Fin ListaEnteros.

1.2.2. Abstracciones de datos

- **Generalización (parametrización de tipo):** el tipo se define en función de otro tipo pasado como parámetro.
- Útil para definir tipos **contenedores** o **colecciones**. Por ej. Listas, pilas, colas, arrays, conjuntos, etc.
- En lugar de:

  ListaEnteros

  ListaCadenas

  ListaReales

  ….
- Tenemos:

  Lista[T]
- Instanciación: Lista[entero],Lista[cadena],…

1.2.2. Abstracciones de datos

TAD Conjunto[T: tipo] *es* Vacío, Insertar, Suprimir, Miembro,
EsVacío, Unión, Intersección, Cardinalidad

Descripción

Los Conjunto[T] son conjuntos matemáticos modificables, que almacenan valores de tipo T.

Operaciones

Operación Vacío (*sal* Conjunto[T])

...

Operación Insertar (*ent c*: Conjunto[T]; *x*: T)

…

Operación Suprimir (*ent c*: Conjunto[T]; *x*: T)

…

Operación Miembro (*ent c*: Conjunto[T]; *x*: T; *sal* booleano)

…

Fin Conjunto.

1.2.2. Abstracciones de datos

TAD Lista[T] *es* Crear, Insertar, Primero, Ultimo, Cabeza,
Cola, EsVacío, Igual

Descripción

Las Lista[T] son listas modificables de valores de tipo T.
Las listas se crean con las operaciones Crear e Insertar...

Operaciones

Operación Crear (*sal* Lista[T])

…

Operación Insertar (*ent l*: Lista[T]; *x*: entero)

…

Operación Primero (*ent l*: Lista[T]; *sal* Lista[T])

…

Fin Lista.

1.2. Especificaciones informales

1.2.3. Abstracciones de iteradores

- **Ejemplo:** Sobre el TAD CjtoEnteros queremos añadir operaciones para calcular la suma, el producto, ...

Operación suma_conj (*ent* c: ConjEnteros; *sal* entero)
    Calcula: devuelve la suma de los elementos de *c*.
    ....
Operación producto_conj (*ent* c: ConjEnteros; *sal* entero)
    ....
Operación varianza_conj (*ent* c: ConjEnteros; *sal* real)
    ....

1.2.3. Abstracciones de iteradores

- Necesitamos abstracciones de la forma:

  para cada elemento *i* del conjunto *A*
  hacer acción sobre *i*

  para cada elemento *i* de la lista *L*
  hacer acción sobre *i*

  para cada *i* de la cola *C* **tal que** *P(i)*
  hacer acción sobre *i*

  D:= **Seleccionar** todos los *i* de *C* **tal que** *P(i)*

- Abstracción de **iteradores**: permiten definir un recorrido abstracto sobre los elementos de una colección.

1.2.3. Abstracciones de iteradores

- La abstracción de iteradores no es soportada por la mayoría de los lenguajes de programación.
- **Posibles definiciones**:
  - Como una abstracción funcional:

Iterador ParaTodoHacer [T: tipo] (*ent C*: Conjunto[T]; *accion*: Operacion)

Requiere: *accion* debe ser una operación que recibe un parámetro de tipo T y no devuelve nada, *accion(ent T)*.

Calcula: Recorre todos los elementos *c* del conjunto *C*, aplicando sobre ellos la operación *accion(c)*.

1.2.3. Abstracciones de iteradores

- Como una abstracción de datos:

TipoIterador IteradorPreorden [T: tipo] **es** Iniciar, Actual, Avanzar, EsUltimo

Descripción

Los valores de tipo IteradorPreorden[T] son iteradores definidos sobre árboles binarios de cualquier tipo T. Los elementos del árbol son devueltos en preorden. El iterador se debe inicializar con Iniciar.

Operaciones

Operación Iniciar (*ent A*: ArbolBinario[T]; *sal* IteradorPreorden)

Calcula: Devuelve un iterador nuevo, colocado sobre la raíz de *A*.

Operación Actual (*ent iter*: IteradorPreorden; *sal* T)

…

Fin IteradorPreorden.

1.2.3. Abstracciones de iteradores

```
var
   A: ArbolBinario[T];
   i: IteradorPreorden[T];
begin
   ...
   i:= Iniciar(A);
   while Not EsUltimo(i) do begin
        Acción sobre Actual(i);
        i:= Avanzar(i);
   end;
   …
```

1.3. Especificaciones formales

- Las especificaciones en lenguaje natural son ambiguas e imprecisas.
- **Especificaciones formales:** definen un TAD o una operación de manera precisa, utilizando un lenguaje matemático.
- Ventajas de una especificación formal:
  - **Prototipado**. Las especificaciones formales pueden llegar a ser ejecutables.
  - **Corrección del programa**. Verificación automática y formal del funcionamiento correcto del programa.
  - **Reusabilidad**. Posibilidad de usar la especificación formal en distintos ámbitos.

1.3. Especificaciones formales

Notación

La descripción formal constará de cuatro partes:

- ***NOMBRE***. Nombre genérico del TAD.
- ***CONJUNTOS***. Conjuntos de datos que intervienen en la definición.
- ***SINTAXIS***. Signatura de las operaciones definidas.
- ***SEMÁNTICA***. Indica el significado de las operaciones, cuál es su resultado.

1.3. Especificaciones formales

- Sintaxis:

<nombre_operación> : <conj_dominio> → <conj_resultado>

- Los distintas notaciones formales difieren en la forma de definir la semántica:
  - **Método axiomático o algebraico.** Se establece el significado de las operaciones a través de relaciones entre operaciones (*axiomas*). Significado implícito de las operaciones.
  - **Método constructivo u operacional.** Se define cada operación por sí misma, independientemente de las otras, basándose en un modelo subyacente. Significado explícito de las operaciones.

1.3.1. Método axiomático (o algebraico)

- La semántica de las operaciones se define a través de un conjunto de **axiomas**.
- Un axioma es una regla de tipo algebraico de la forma:

<operación> (<valores particulares>) = <expresión del resultado>

- ¿Qué axiomas introducir en la semántica?
- Los axiomas deben ser los necesarios para satisfacer dos propiedades:
  - **Completitud**: los axiomas deben ser los suficientes para poder deducir el significado de cualquier expresión.
  - **Corrección**: a partir de una expresión sólo se puede obtener un resultado.

1.3.1. Método axiomático (o algebraico)

Ejemplo: TAD Natural de los números naturales.

NOMBRE

Natural

CONJUNTOS

N	Conjunto de naturales
Bool	Conjunto de booleanos {true, false}

SINTAXIS

cero:		$\rightarrow$	N
sucesor:	N	$\rightarrow$	N
suma:	N x N	$\rightarrow$	N
esCero:	N	$\rightarrow$	Bool
esIgual:	N x N	$\rightarrow$	Bool

1.3.1. Método axiomático (o algebraico)

SEMÁNTICA

$\forall$ m, n $\in$ N

1. suma (cero, n) = n
2. suma (sucesor (m), n) = sucesor (suma (m, n))
3. esCero (cero) = true
4. esCero (sucesor (n)) = false
5. esIgual (cero, n) = esCero (n)
6. esIgual (sucesor (n), cero) = false
7. esIgual(sucesor(n), sucesor(m)) = esIgual(n, m)

1.3.1. Método axiomático (o algebraico)

- Supongamos un TAD, **T**.
- **Tipos de operaciones**:
  - **Constructores.** Conjunto mínimo de operaciones del TAD, a partir del cual se puede obtener cualquier valor del tipo **T**.

$$\underline{c1}: \rightarrow T,\ \underline{c2}: V \rightarrow T,\ \underline{c3}: V_1 x ... x V_n \rightarrow T$$

  - **Modificación**. A partir de un valor del tipo obtienen otro valor del tipo **T**, y no son constructores.

$$\underline{m1}: T \rightarrow T,\ \underline{m2}: TxV \rightarrow T,\ \underline{m3}: V_1 x ... x V_n \rightarrow T$$

  - **Consulta.** Devuelven un valor que no es del tipo **T**.

$$\underline{o1}: T \rightarrow V,\ \underline{o2}: TxV \rightarrow V',\ o\underline{3}: V_1 x ... x V_n \rightarrow V_{n+1}$$

1.3.1. Método axiomático (o algebraico)

- La ejecución de una expresión acaba al expresarla en función de los constructores.
- ¿Cómo garantizar que una especificación es completa y correcta?
- Definir los axiomas suficientes para relacionar las operaciones de modificación y consulta con los constructores.
- No incluir axiomas que se puedan deducir de otros existentes.

1.3.1. Método axiomático (o algebraico)

- Ejemplo: Especificación del TAD genérico pila.

NOMBRE

Pila [T]

CONJUNTOS

P	Conjunto de pilas
T	Conjunto de elementos que pueden ser almacenados
Bool	Conjunto de booleanos {true, false}

SINTAXIS

pilaVacía:		$\rightarrow$	P
esVacía:	P	$\rightarrow$	Bool
pop:	P	$\rightarrow$	P
tope:	P	$\rightarrow$	T
push:	T x P	$\rightarrow$	P

1.3.1. Método axiomático (o algebraico)

- En el caso de **tope: P → T**, ¿qué pasa si la pila está vacía?
- Se puede añadir un conjunto de mensajes en **CONJUNTOS**, de la forma:

  M Conjunto de mensajes {Error.Pilavacía}

- Y cambiar en la parte de **SINTAXIS** la operación **tope**:

  tope: P → T U M

1.3.1. Método axiomático (o algebraico)

	pilaVacía	push (t, p)
esVacía ( )	esVacía(pilaVacía) =	esVacía(push(t, p)) =
pop ( )	pop(pilaVacía) =	pop(push(t, p)) =
tope ( )	tope(pilaVacía) =	tope(push(t, p)) =

1.3.1. Método axiomático (o algebraico)

SEMÁNTICA

$\forall$ t $\in$ T; $\forall$ p $\in$ P

1. esVacía (pilaVacía) = true
2. esVacía (push (t, p)) = false
3. pop (pilaVacía) = pilaVacía
4. pop (push (t, p)) = p
5. tope (pilaVacía) = Error.Pilavacía
6. tope (push (t, p)) = t

1.3.1. Método axiomático (o algebraico)

•Para facilitar la escritura de la expresión del resultado en la semántica, se pueden emplear expresiones condicionales de la forma:

SI <condición> $\Rightarrow$ <valor si cierto> **|** <valor si falso>

•**Ejemplo: Especificación algebraica del TAD bolsa.**

NOMBRE

Bolsa[I]

CONJUNTOS

B	Conjunto de bolsas
I	Conjunto de elementos
Bool	Conjunto de booleanos {true, false}
N	Conjunto de naturales

SINTAXIS

bolsaVacía:		$\rightarrow$	B
poner:	I x B	$\rightarrow$	B
esVacía:	B	$\rightarrow$	Bool
cuántos:	I x B	$\rightarrow$	N

1.3.1. Método axiomático (o algebraico)

- Incluir una operación quitar, que saque un elemento dado de la bolsa.
- ¿Y si queremos que los saque todos?
- Incluir una operación esIgual, de comparación de bolsas.

1.3.1. Método axiomático (o algebraico)

Conclusiones:

- Las operaciones no se describen de manera explícita, sino **implícitamente** relacionando el resultado de unas con otras.
- La construcción de los axiomas se basa en un **razonamiento inductivo**.
- ¿Cómo se podría especificar, por ejemplo, un procedimiento de ordenación?

1.3.2. Método constructivo (operacional)

- Para cada operación, se establecen las precondiciones y las postcondiciones.
- **Precondición**: Relación que se debe cumplir con los datos de entrada para que la operación se
pueda aplicar.
- **Postcondición**: Relaciones que se cumplen después de ejecutar la operación.

1.3.2. Método constructivo (operacional)

Notación

En la semántica, para cada operación <nombre>:

pre-<nombre> (<param_entrada>)::= <condición_lógica>

post-<nombre> (<param_entrada>;<param_salida>)::= <condición_lógica>

- **Ejemplo: operación máximo, que tiene como entrada dos números reales y da como salida el mayor de los dos.**

máximo: $R \times R \rightarrow R$ *(SINTAXIS)*

pre-máximo(x, y) ::= true *(SEMÁNTICA)*

post-máximo(x, y; r) ::= $(r \geq x) \wedge (r \geq y) \wedge (r=x \vee r=y)$

1.3.2. Método constructivo (operacional)

- **Ejemplo: operación máximo sobre números reales, pero restringida a números positivos.**

  máximop: $R \times R \rightarrow R$

  pre-máximop$(x, y) ::= (x \geq 0) \wedge (y \geq 0)$

  post-máximop$(x, y; r) ::= (r \geq x) \wedge (r \geq y) \wedge (r=x \vee r=y)$

- ¿Qué sucedería si *x* o *y* no son mayores que 0?
- No se cumple la precondición $\rightarrow$ no podemos asegurar que se cumpla la postcondición.

1.3.2. Método constructivo (operacional)

- **Otra posibilidad:** definir un conjunto **M** (de mensajes de error) y cambiar la imagen. Modificar la sintaxis y la semántica:

  máximop2: $R \times R \rightarrow R \cup M$

  pre-maximo2(x, y) ::= true

  pos-máximop2(x, y; r) ::= SI $(x \geq 0) \wedge (y \geq 0)$

  $\Rightarrow (r \geq x) \wedge (r \geq y) \wedge (r=x \vee r=y)$

  | r = “Fuera de rango”

- ¿Cuál es la mejor opción?

1.3.2. Método constructivo (operacional)

- ¿Cómo se pueden definir las pre- y post-condiciones cuando el TAD es más complejo? Por ejemplo, para TAD colecciones.
- Necesitamos un **modelo subyacente**, en el cual se base la definición del TAD.
- No siempre se encuentra uno adecuado...

- **Ejemplo:** Para definir el TAD **Pila[I]**, definiremos el TAD **Lista[I]** por el método axiomático, y luego lo usaremos para definir el TAD pila con el método constructivo.

1.3.2. Método constructivo (operacional)

NOMBRE

Lista[I]

CONJUNTOS

L	Conjunto de listas
I	Conjunto de elementos
B	Conjunto de booleanos {true, false}
N	Conjunto de naturales
M	Conjunto de mensajes {“La lista está vacía”}

SINTAXIS

crearLista:		→	L
formarLista:	I	→	L
concatenar:	L x L	→	L
último:	L	→	I U M
cabecera:	L	→	L
primero:	L	→	I U M
cola:	L	→	L
longitud:	L	→	N
esListaVacía:	L	→	B

1.3.2. Método constructivo (operacional)

SEMÁNTICA

$\forall i \in I; \forall a,b \in L$

1. último (crearLista) = “La lista está vacía”
2. último (formarLista (i)) = i
3. último (concatenar (a, b)) = SI esListaVacía (b) $\Rightarrow$
   último (a) | último (b)
4. cabecera (crearLista) = crearLista
5. cabecera (formarLista (i)) = crearLista
6. cabecera (concatenar (a, b)) = SI esListaVacía (b) $\Rightarrow$
   cabecera (a) | concatenar (a, cabecera (b))
7. primero (crearLista) = “La lista está vacía”
8. primero (formarLista (i)) = i
9. primero (concatenar (a, b)) = SI esListaVacía (a) $\Rightarrow$
   primero (b) | primero (a)

1.3.2. Método constructivo (operacional)

10. cola (crearLista) = crearLista
11. cola (formarLista (i)) = crearLista
12. cola (concatenar (a, b)) = SI esListaVacía (a) $\Rightarrow$
    cola (b) | concatenar (cola (a), b)

13. longitud (crearLista) = cero
14. longitud (formarLista (i)) = sucesor (cero)
15. longitud (concatenar (a, b)) = suma (longitud (a), longitud (b))

16. esListaVacía (crearLista) = true
17. esListavacía (formarLista (i)) = false
18. esListaVacía (concatenar (a, b)) = esListaVacía (a)
    AND esListaVacía(b)

Aserto invariante: siempre que aparezca un mensaje a la entrada de una operación, la salida será el mismo mensaje.

1.3.2. Método constructivo (operacional)

- Seguimos el ejemplo y aplicamos el método constructivo a la definición de **Pila[I]**, teniendo como modelo subyacente el tipo **Lista[I]**.

NOMBRE

Pila[I]

CONJUNTOS

S Conjunto de pilas
I Conjunto de elementos
B Conjunto de valores booleanos {true, false}
M Conjunto de mensajes {“La pila está vacía”}

SINTAXIS

crearPila: → S
tope: S → I U M
pop: S → S U M
push: I x S → S
esVacíaPila: S → B

1.3.2. Método constructivo (operacional)

SEMÁNTICA

$\forall i \in I; \forall s \in S; b \in B; r \in S; t \in I \cup M; p \in S \cup M$

1. pre-crearPila () ::= true
2. post-crearPila (s) ::= s = crearLista
3. pre-tope (s) ::= true
4. post-tope (s; t) ::= SI esListaVacía (s)
   $\Rightarrow$ t = "La pila está vacía"
   | t = primero (s)
5. pre-pop (s) ::= true
6. post-pop (s; p) ::= SI esListaVacía (s)
   $\Rightarrow$ p = "La pila está vacía"
   | p = cola (s)
7. pre-push (i, s) ::= true
8. post-push (i, s; r) ::= r = concatenar (formarLista (i), s)
9. pre-esVacíaPila (s) ::= true
10. post-esVacíaPila (s; b) ::= b = esListaVacía (s)

1.3.2. Método constructivo (operacional)

NOMBRE

Pila[I]

CONJUNTOS

S Conjunto de pilas

I Conjunto de elementos

B Conjunto de valores booleanos {true, false}

SINTAXIS

crearPila:		$\rightarrow$	S
tope:	S	$\rightarrow$	I
pop:	S	$\rightarrow$	S
push:	I x S	$\rightarrow$	S
esVacíaPila:	S	$\rightarrow$	B

1.3.2. Método constructivo (operacional)

SEMÁNTICA

$\forall$ i, t$\in$I; $\forall$ s, r, p$\in$S; $\forall$ b$\in$B

1. pre-crearPila () ::= true
2. post-crearPila (s) ::= s = crearLista
3. pre-tope (s) ::= NOT esListaVacía (s)
4. post-tope (s; t) ::= t = primero (s)
5. pre-pop (s) ::= NOT esListaVacía (s)
6. post-pop (s; p) ::= p = cola (s)
7. pre-push (i, s) ::= true
8. post-push (i, s; r) ::= r = concatenar (formarLista (i), s)
9. pre-esVacíaPila (s) ::= true
10. post-esVacíaPila (s; b) ::= b = esListaVacía (s)

1.3.2. Método constructivo (operacional)

- Seguimos el ejemplo y aplicamos el método constructivo a la definición de **Cola[I]**, teniendo como modelo subyacente el tipo **Lista[I]**.

NOMBRE

Cola[I]

CONJUNTOS

C Conjunto de colas
I Conjunto de elementos
B Conjunto de valores booleanos {true, false}
M Conjunto de mensajes {“La cola está vacía”}

SINTAXIS

crearCola:		$\rightarrow$	C
frente:	C	$\rightarrow$	I U M
inserta:	C	$\rightarrow$	C U M
resto:	I x C	$\rightarrow$	C
esVacíaCola:	C	$\rightarrow$	B

1.3.2. Método constructivo (operacional)

- **Ejecución de la especificación:** comprobar precondiciones y postcondiciones de todas las operaciones de la expresión.

- **Ejemplos**: Pila[Natural]

  a) tope (push (4, pop (push (2, crearPila))))

  b) esVacíaPila (push (2, pop (crearPila)))

1.3.2. Método constructivo (operacional)

- ¿Cuál es la mejor solución?
- **Programación por contrato** (tomado de OO).
- **Contrato de una operación:** si se cumplen unas condiciones en los parámetros de entrada, entonces garantiza una obtención correcta del resultado.
- **Idea**:
  - La operación no trata todos los casos de error, sino que hace uso de las precondiciones.
  - La responsabilidad de comprobar la condición de error es del que usa la operación.

1.3.2. Método constructivo (operacional)

Conclusiones:

- La especificación constructiva está limitada por la necesidad de **modelos subyacentes**.
- **No confundir** especificación con implementación.
- Es más fácil incluir especificaciones constructivas en los programas (p.ej., mediante asertos).