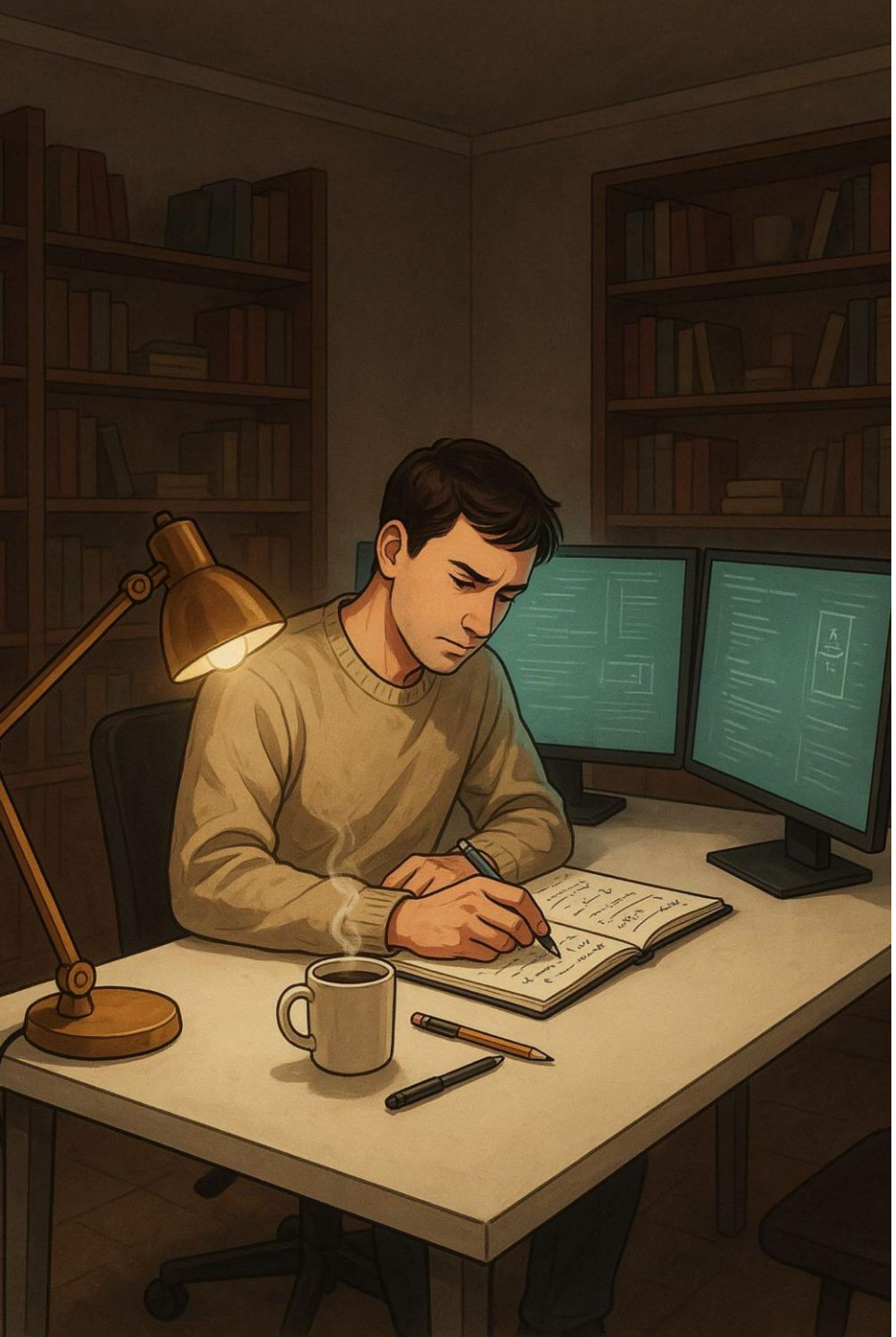


Correctitud y Cierre de la Verificación Antes de las Pruebas

DESARROLLO FORMAL DE SOFTWARE

INGENIERÍA DE SISTEMAS · UNIVERSIDAD DEL PACÍFICO

by ykcaicedo



PROPÓSITO

Correctitud parcial vs. total

Consolidar la diferencia y saber cuándo aplica cada concepto.

Contratos, aserciones e invariantes

Aplicarlos a una funcionalidad real del proyecto integrado.

Qué verificar antes de las pruebas

Identificar las condiciones que deben justificarse previamente.

Producto de clase

Ficha breve de verificación de una funcionalidad del proyecto.

by ykcaicedo

CONCEPTO FUNDAMENTAL

¿Qué significa que un programa sea correcto?

Satisface la especificación

El comportamiento del programa debe corresponder exactamente a las condiciones definidas.

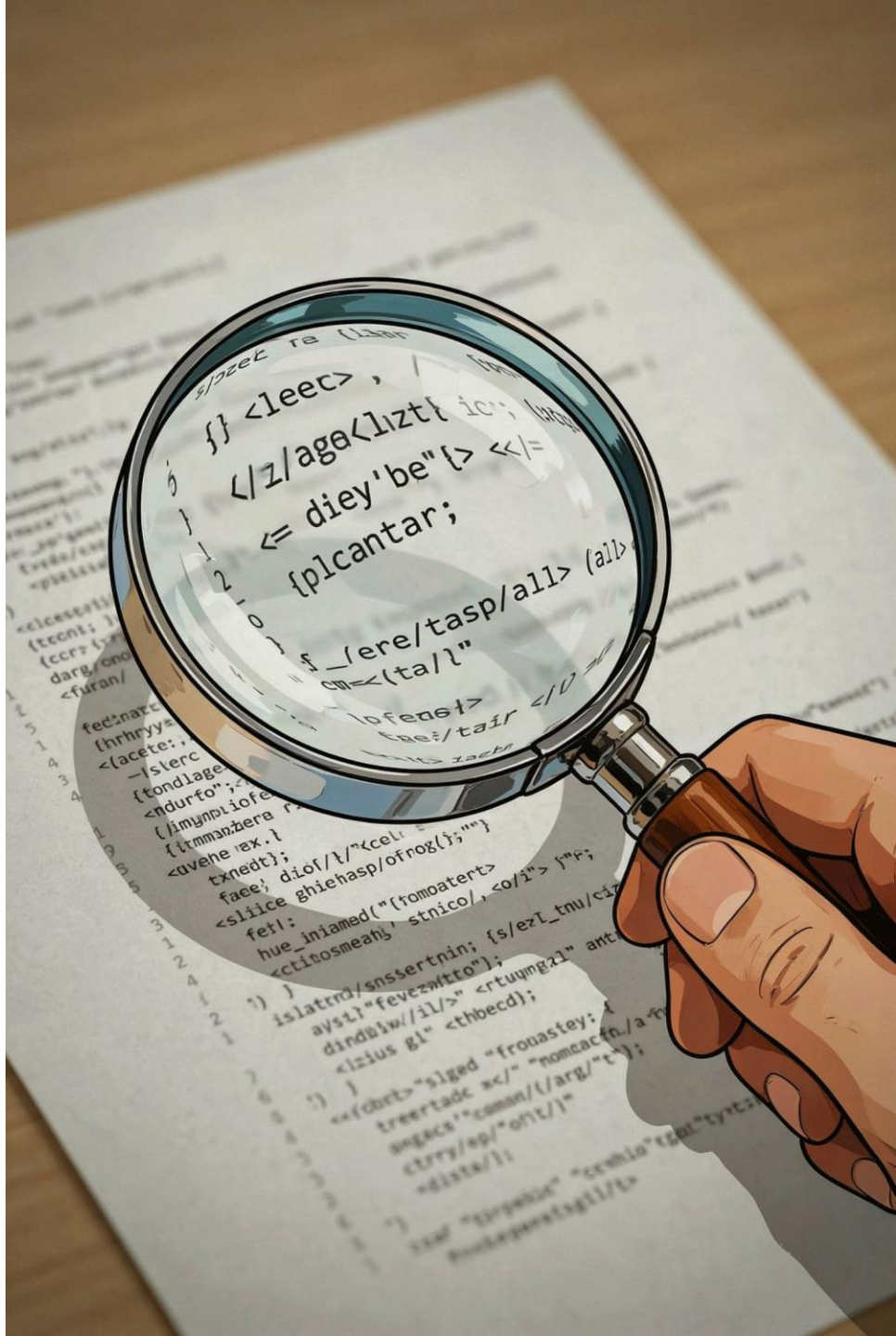
No basta con compilar

Producir una salida en un único ejemplo no garantiza correctitud ante todas las entradas permitidas.

La verificación aporta argumentos

Se razona sobre entradas, instrucciones y resultados para justificar la correspondencia.

Pregunta inicial: ¿un programa que compila necesariamente es correcto?



CONTRATOS

Recuperemos el contrato: PRE y POST

El contrato delimita **qué se espera del método** y **bajo qué condiciones puede utilizarse**. Una implementación debe respetar la relación entre PRE, ejecución y POST.

Ejemplo: `retirar(valor)`

```
PRE:  valor > 0  ∧  valor <= saldo

Operación: retirar(valor)

POST: saldoFinal = saldoInicial - valor
```

PRE

Condición que debe cumplirse *antes* de ejecutar la operación.

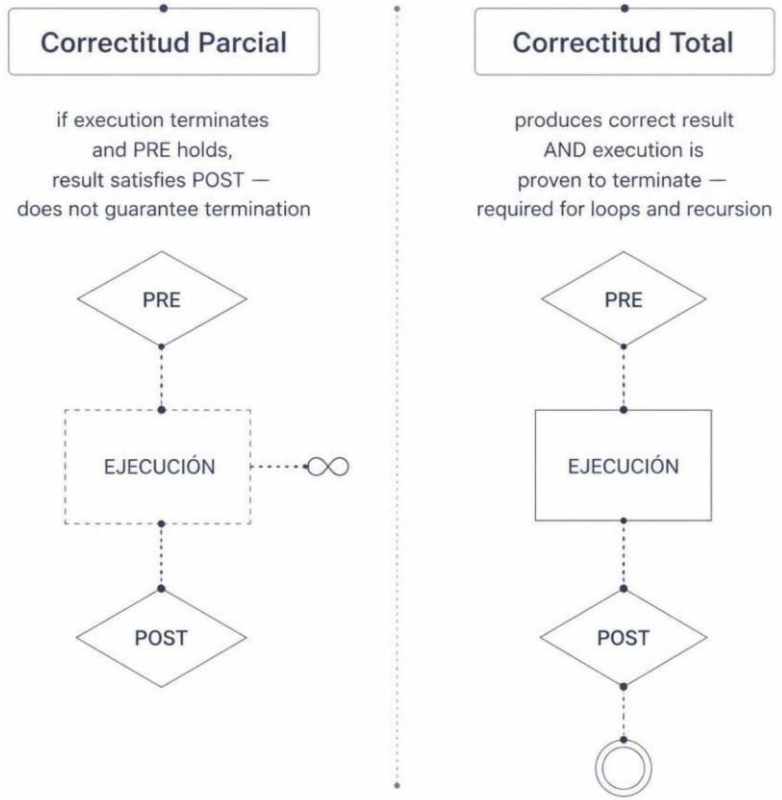
POST

Condición que debe cumplirse al *finalizar* correctamente.

by ykcaicedo

CONCEPTOS CLAVE

Correctitud parcial y correctitud total



La **correctitud parcial** no garantiza por sí sola que el programa finalice. Para ciclos o recursividad se necesita un argumento específico de **terminación**: demostrar que la ejecución concluye en tiempo finito bajo cualquier entrada válida.

EJEMPLO APLICADO

Retiro de una cuenta bancaria

Escenario válido

```
saldo = 100
valor = 30

PRE:  valor > 0  ∧  valor <= saldo
      ✓ 30 > 0  ∧  30 <= 100

POST: saldoFinal = 100 - 30 = 70
```

Entrada no permitida

```
valor = 130
PRE: 130 <= 100  → FALSA
→ La operación debe rechazar
  la solicitud.
```

Interpretación

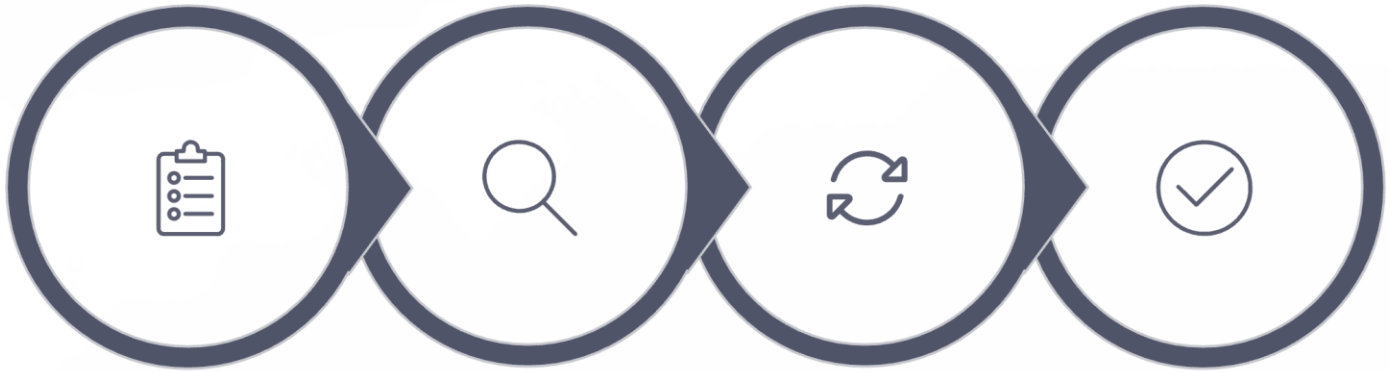
Si la PRE no se cumple, el contrato **no obliga** a la operación a producir ningún resultado concreto. Rechazar la solicitud es el comportamiento correcto.

⚠ No confundir una entrada no permitida con un incumplimiento de la POST.

by ykcaicedo

ESTRUCTURAS SECUENCIALES

Verificación de una estructura secuencial



Estado inicial

Analizar instrucción

Actualizar variables

Comparar estado final

Aplicación concreta

```
saldo = 100
valor = 30
saldo = saldo - valor
// Resultado: saldo = 70
```

Se parte del estado inicial, se sigue la instrucción y se contrasta el estado final con la POST. El resultado `70` satisface `saldoFinal = saldoInicial – valor`.

ESTRUCTURAS CONDICIONALES

Verificación de una estructura condicional

Fragmento analizado

```
if (valor > 0 && valor <= saldo) {
    saldo = saldo - valor;
    resultado = true;
} else {
    resultado = false;
}
```

Claves del análisis

- Identificar la condición que decide qué rama se ejecuta.
- Verificar que **cada rama** conduce a un estado compatible con la POST.
- Analizar también el caso en que la condición es falsa.
- Una rama no analizada puede ocultar un comportamiento incorrecto.

by ykcaicedo

CICLOS·INVARIANTE

Verificación de ciclos: el invariante

Ejemplo

```
int i = 0;
int suma = 0;
while (i < n) {
    suma += datos[i];
    i++;
}
// Invariante: suma contiene la
// suma de los elementos
// ya procesados (índices 0..i-1)
```

01

Inicialización

El invariante debe ser verdadero antes de iniciar el ciclo.

02

Preservación

Cada iteración debe mantener verdadero el invariante.

03

Uso al salir

El invariante + la condición de salida permiten razonar sobre la POST.

by ykcaicedo

TERMINACIÓN

¿Cómo se justifica la terminación?

Ejemplo de ciclo

```
while (n > 0) {
    n--;
}
// Medida: n, entero no negativo.
// En cada iteración disminuye 1.
// Límite inferior: 0.
```

Recursividad

Verificar que cada llamada recursiva se acerca a un **caso base** definido.

1

Identificar el avance

¿Qué hace que el ciclo o la recursión progrese hacia su fin?

2

Definir una medida

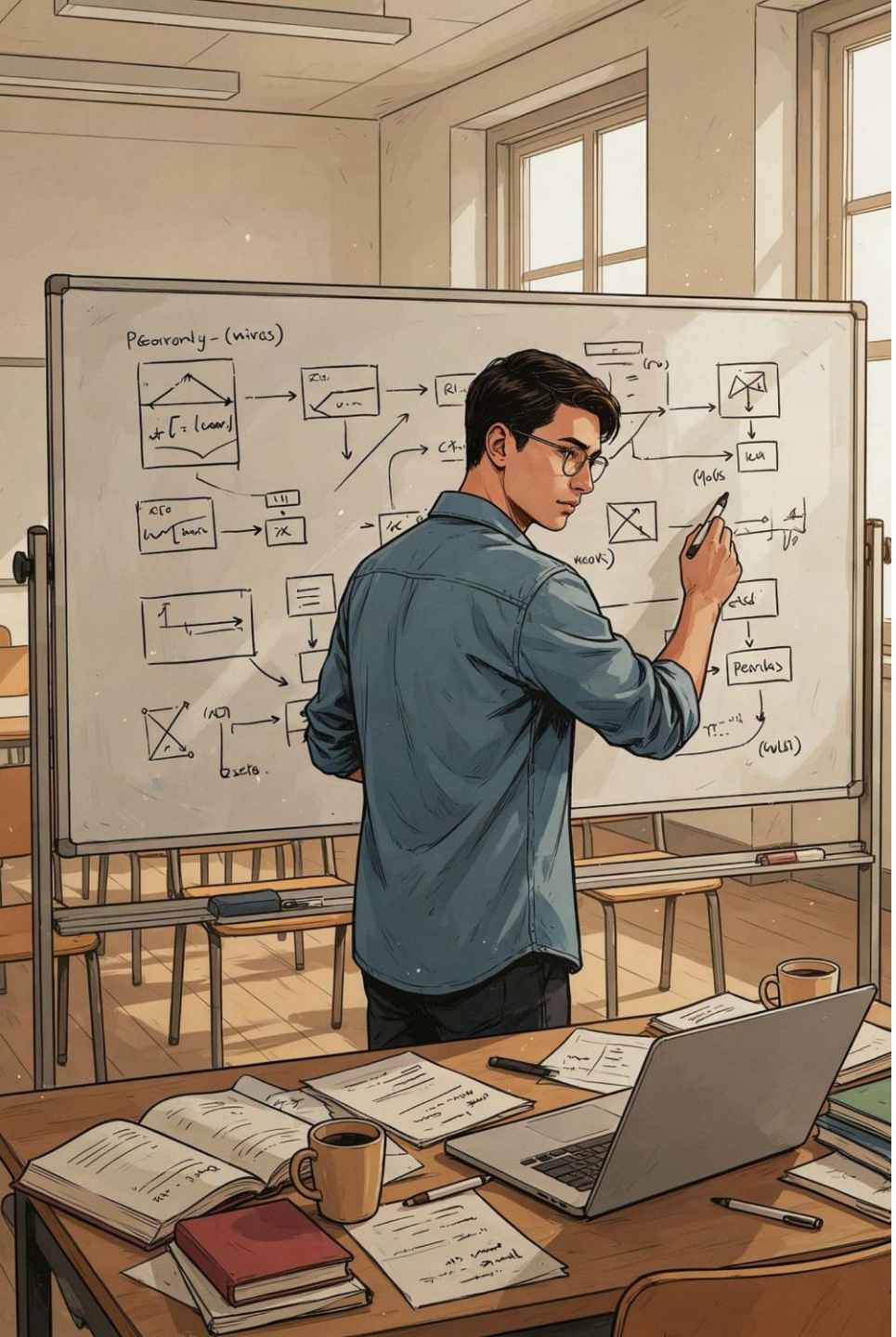
Un valor que cambia en cada iteración y está acotado por un límite.

3

Verificar el límite

La medida debe alcanzar ese límite en tiempo finito.

by ykcaicedo



EJERCICIO GUIADO

Revisar un algoritmo

Analice el siguiente fragmento

```
int i = 0;
int contador = 0;
while (i < numeros.length) {
    if (numeros[i] > 0) {
        contador++;
    }
    i++;
}
```

1. ¿Qué representa `contador`?

2. Proponga un invariante.

3. ¿Qué garantiza que el ciclo termina?

by ykcaicedo

SOLUCIÓN ESPERADA

Análisis del ejercicio guiado

Significado de `contador`

Representa la cantidad de números positivos entre los elementos ya revisados en el arreglo.

Invariante

Al inicio de cada iteración, `contador` es igual al número de positivos en las posiciones de 0 hasta `i - 1`.

Inicialización

Con `i = 0` no se ha procesado ningún elemento y `contador = 0`: el invariante se cumple trivialmente.

Preservación

Se incrementa `contador` solo si el elemento actual es positivo; después aumenta `i`.

Terminación

`i` aumenta en cada iteración y el ciclo finaliza cuando `i = numeros.length`.

by ykcaicedo

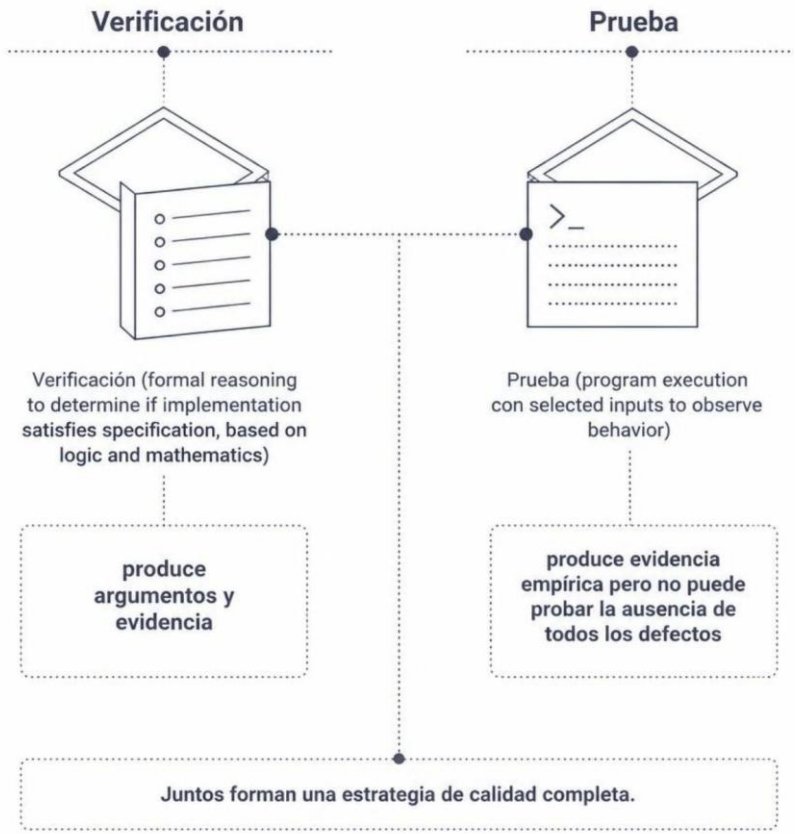
PRODUCTO DE CLASE

Ficha de verificación del equipo

1	**Funcionalidad y propósito** Nombre de la operación y descripción breve de qué hace.
2	**PRE** Condiciones que deben cumplirse antes de ejecutar la operación.
3	**POST** Resultado esperado al finalizar correctamente la operación.
4	**Estructuras presentes** Indicar si hay secuencia, condicional, ciclo o recursividad.
5	**Invariante y terminación** Invariante del ciclo (si aplica) y argumento que garantiza la terminación.
6	**Conclusión** ¿Qué se pudo justificar formalmente? ¿Qué queda pendiente?

DISTINCIÓN ESENCIAL

Verificación no es lo mismo que prueba



Una prueba exitosa aporta **evidencia empírica**, pero no demuestra por sí sola la ausencia de defectos ante todas las entradas posibles.

La siguiente etapa utilizará **casos normales, límite e inválidos** para complementar lo que la verificación ha argumentado formalmente.



CIERRE · UNIDAD 3

Cierre y conexión con la siguiente etapa

Especificación + implementación + argumentos

Los tres elementos deben estar relacionados y documentados.

Ciclos: invariante y terminación

No basta con el invariante; también se debe justificar que el ciclo finaliza.

Funciones y clases: contratos

Se revisan los contratos y la conservación de los estados válidos del sistema.

Próximo paso: pruebas

Diseñar y ejecutar pruebas para encontrar defectos y observar el comportamiento real.

by ykcaicedo