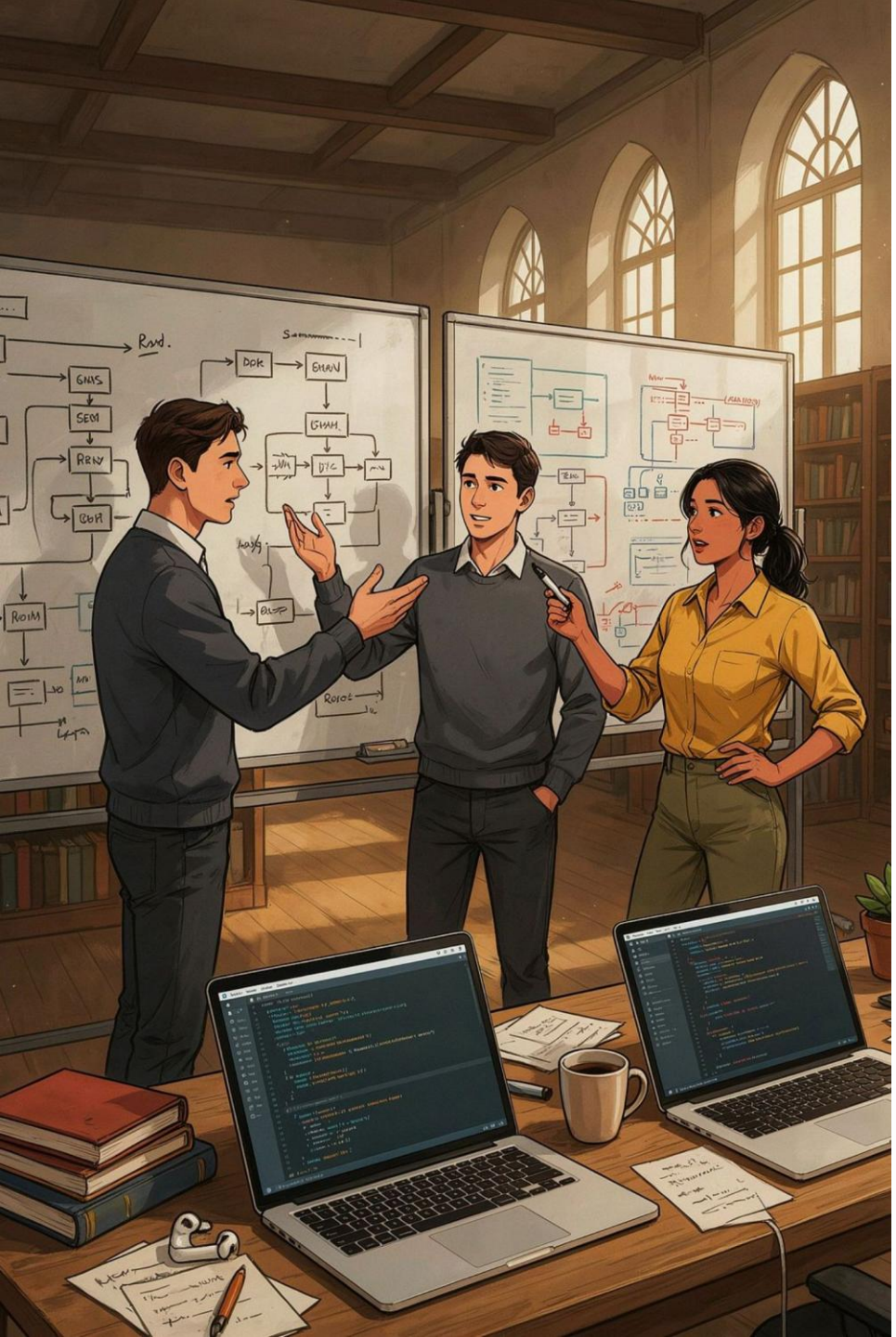


Calidad y Construcción de Software

DESARROLLO FORMAL DE SOFTWARE

Relación entre especificación, construcción, verificación y pruebas.

Cierre del Capítulo 1 + Desarrollo del Capítulo 2

INGENIERÍA DE SISTEMAS
UNIVERSIDAD DEL PACÍFICO

Mg. Yowanna Karina Caicedo Guerrero

Ubicación de la clase

01

Unidad 1

Fundamentos del desarrollo formal y calidad del software

02

Capítulo 1

Introducción al Desarrollo Formal — **Cierre conceptual**

03

Capítulo 2

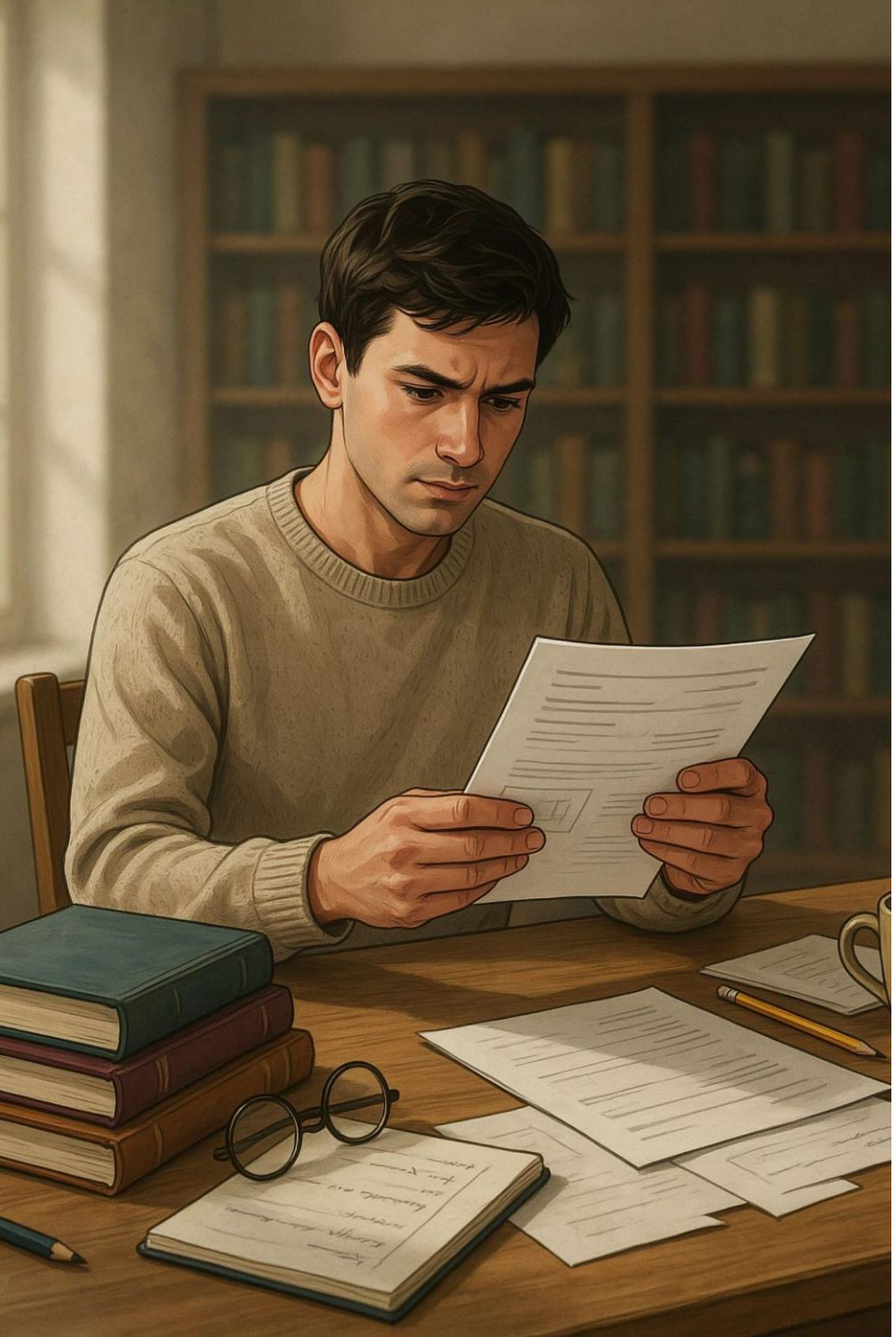
Calidad y construcción de software — **Desarrollo principal de la sesión**

04

Capítulo 3

Lógica proposicional — puente hacia la próxima clase

Hoy no profundizaremos todavía en lógica proposicional; será el puente hacia el Capítulo 3.



Objetivo específico

Relacionar la especificación, construcción, verificación y pruebas con la **calidad del software**, reconociendo cómo las prácticas de construcción contribuyen al desarrollo de productos **confiables, mantenibles y robustos**.

¿Dónde quedamos en la clase anterior?

Error, Defecto y Fallo

Diferenciamos tres conceptos clave en la terminología de calidad.

Verificación vs. Validación

Distinguimos qué se analiza en cada actividad y cuándo se aplica.

Tipos de Casos de Prueba

Analizamos casos normal, límite e inválido con ejemplos concretos.

“¿Cómo sabemos que una funcionalidad que estamos construyendo realmente cumple lo que esperamos de ella?"

ESPECIFICACIÓN

Retomamos

La relación que mencionamos rápidamente al final de la sesión anterior:

Especificación → Construcción → Verificación → Pruebas

¿Por qué necesitamos una Especificación?

Claridad antes de construir

Antes de implementar una funcionalidad, debemos saber con precisión **qué esperamos de ella**.

Descripción del comportamiento esperado

La especificación define el comportamiento y las **condiciones que deben cumplirse**.

Referencia compartida

Sin ella, es difícil determinar si la implementación es correcta. Se convierte en la referencia para **construir, verificar y probar**.

Tenemos una funcionalidad: **Registrar solicitud de transporte.**

Podemos decir: Un estudiante activo puede registrar una solicitud cuando existen cupos disponibles.

Especificación → Construcción

La correspondencia es obligatoria

La construcción debe mantener **coherencia directa** con lo especificado. Cualquier desviación introduce riesgo de error.

Ejemplo: si una solicitud solo puede registrarse con cupos disponibles, el código **debe** contemplar y verificar esa condición.

Construcción → Verificación

Verificar significa analizar si lo construido **cumple las condiciones establecidas**. No basta con observar que el programa ejecuta sin errores visibles.

¿Qué preguntamos?

¿El comportamiento observado corresponde con lo que se especificó?

¿Cómo verificamos?

Mediante diferentes técnicas; más adelante estudiaremos métodos formales con mayor profundidad.

Verificación → Pruebas

Caso normal

Comprueba un comportamiento esperado en **condiciones habituales**.

Caso límite

Explora una **frontera relevante** del comportamiento definido.

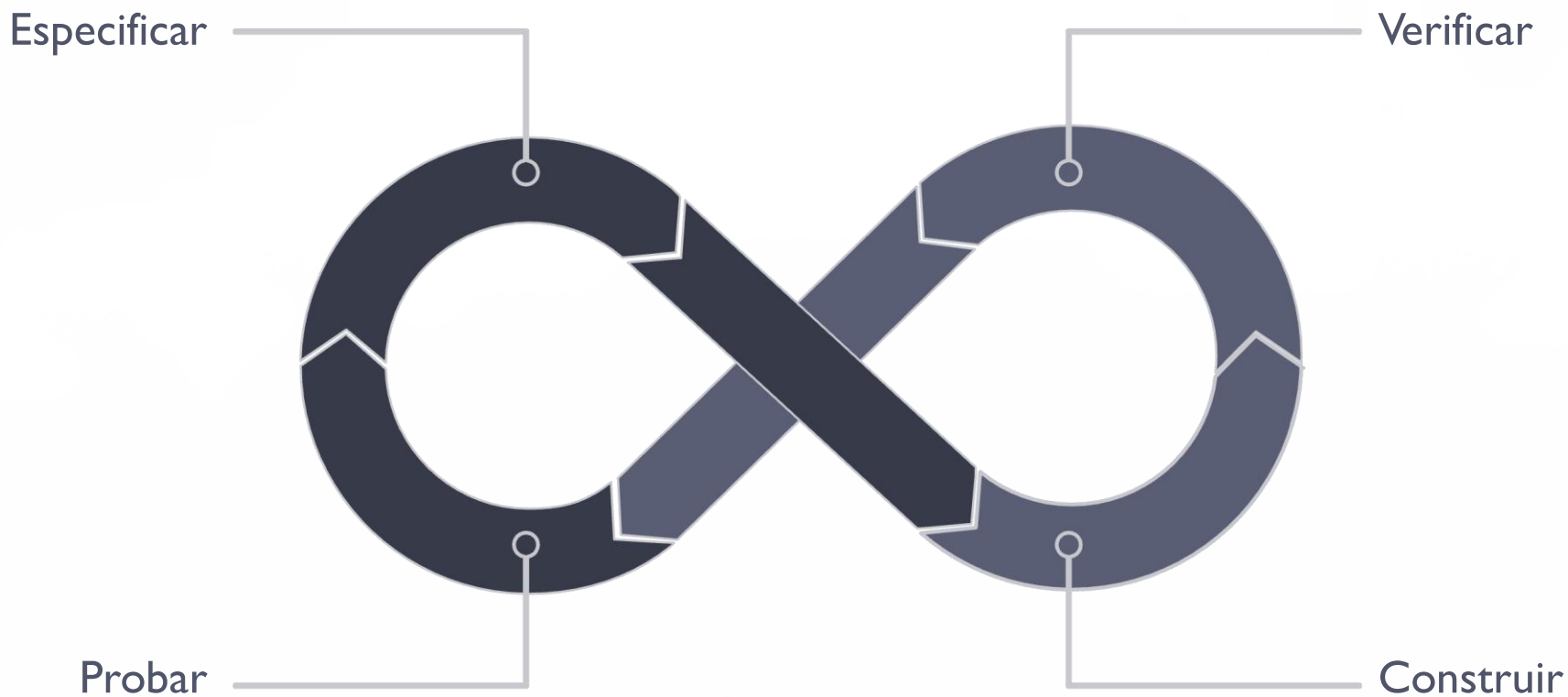
Caso inválido

Comprueba cómo responde el programa ante una situación que **no debe aceptarse**.

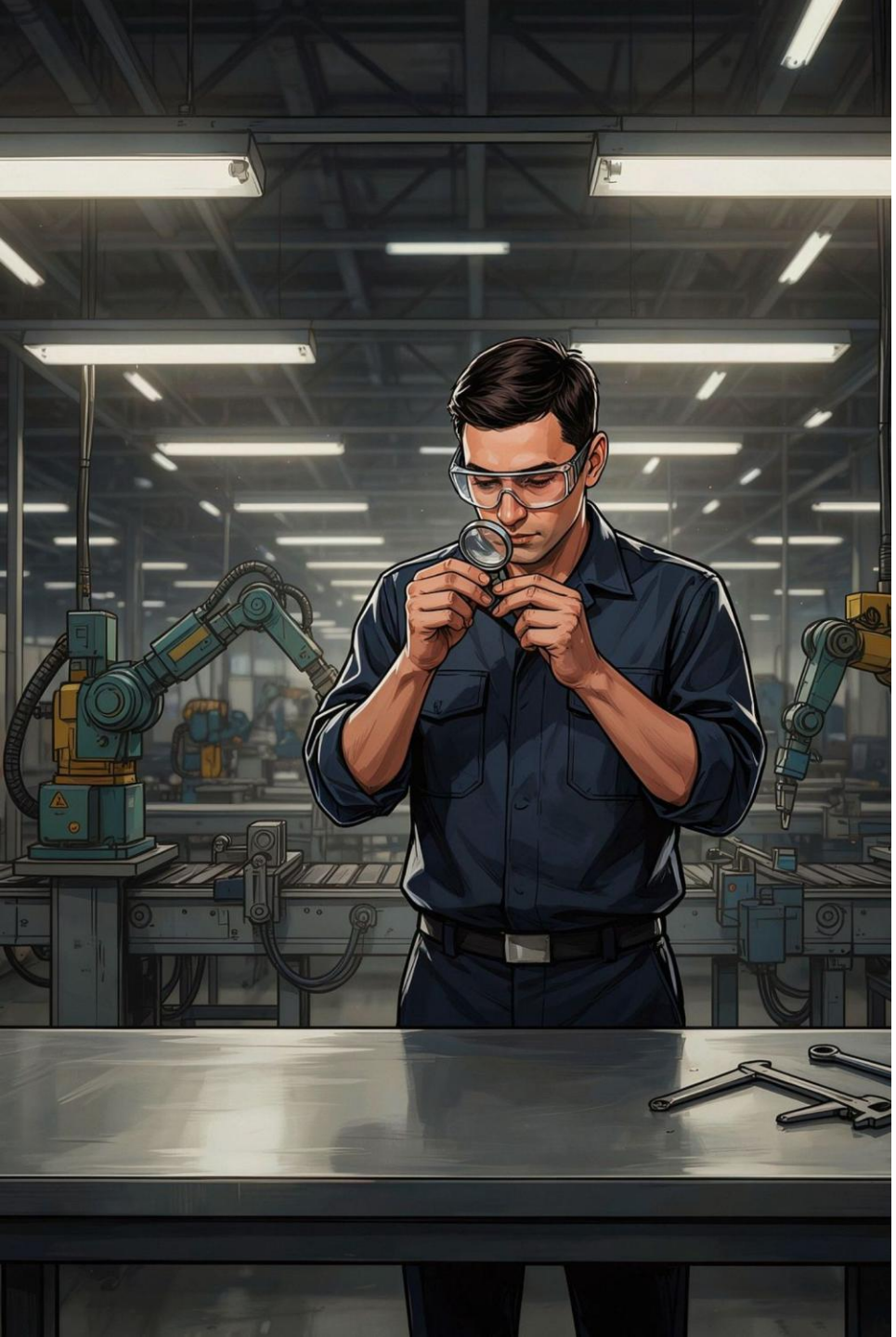
Una prueba aporta evidencia sobre los casos ejecutados; no demuestra por sí sola la corrección de todos los casos posibles.

Las pruebas no aparecen aisladas. Las diseñamos teniendo en cuenta lo que esperamos que haga la funcionalidad."

La relación no es una línea rígida



En un proyecto real, estas actividades se **repiten**. Una prueba puede revelar un comportamiento incorrecto, lo que obliga a **analizar, corregir y verificar** de nuevo. El ciclo garantiza que cada corrección se valide.



¿Qué entendemos por calidad del software?

> Calidad no significa únicamente que el programa "*funcione*". El producto debe cumplir las **condiciones y características esperadas**, comportarse correctamente ante entradas normales, límites e inesperadas, y poder sustentarse con evidencias.

En DFS nos interesa especialmente una calidad que pueda **analizarse y justificarse formalmente**.

Características de calidad que trabajaremos

Calidad funcional

Realiza correctamente lo que debe hacer según lo especificado.

Confiabilidad

Mantiene un comportamiento adecuado bajo las condiciones previstas.

Mantenibilidad

Puede comprenderse, modificarse y corregirse con menor esfuerzo.

Robustez

Responde adecuadamente ante entradas inesperadas o condiciones de error.

Calidad funcional

Pregunta central

¿La funcionalidad hace **lo que se espera**, según lo que se especificó?

Conecta directamente con especificación, verificación y pruebas: si no hay especificación clara, no podemos evaluar la calidad funcional.

Ejemplo

La función debe permitir registrar una solicitud cuando:

- El estudiante está activo; hay disponibilidad.

Si permite registrar una solicitud cuando no hay disponibilidad: tenemos un problema de **calidad funcional**.

"La calidad funcional depende de que el comportamiento del software corresponda con lo que se espera de él."

Confiabilidad

La confiabilidad tiene que ver con que el software mantenga el comportamiento esperado cuando se utiliza repetidamente y bajo las condiciones previstas.

> Si una funcionalidad funciona una vez, eso no necesariamente nos dice mucho.

Ejemplo: Reservar Cupo

Funcionalidad:
Reservar cupo para transporte

La funcionalidad debe permitir reservar un cupo cuando hay disponibilidad y rechazar la solicitud cuando no se cumplen las condiciones

Prueba	Condición	Resultado esperado
1	Hay 10 cupos y se solicita 1	Reserva realizada. Quedan 9 cupos.
2	Hay 9 cupos y se solicita 1	Reserva realizada. Quedan 8 cupos.
3	Hay 5 cupos y se solicitan 2	Reserva realizada. Quedan 3 cupos.
4	Hay 1 cupo y se solicita 1	Reserva realizada. Quedan 0 cupos.
5	Hay 0 cupos y se solicita 1	Reserva rechazada. No se realiza la reserva.

Mantenibilidad y Legibilidad

¿Qué pasa si dentro de dos meses necesitamos cambiar una parte del programa?. ¿Será fácil modificarlo?

Nombres **claros**

Facilitan comprender el propósito del código sin necesidad de comentarios adicionales.

Responsabilidades bien definidas

Las funciones y métodos deben hacer **una sola cosa** y hacerla bien.

Sin duplicación innecesaria

Evitar repetición facilita modificaciones y reduce el riesgo al corregir.

Legible por otras personas

Un programa correcto pero ilegible es difícil de mantener a largo plazo. Estructura comprensible.

Robustez

¿Qué pasa si el usuario hace algo que nosotros no esperábamos?

> Ejemplo:
>
> La funcionalidad espera:
>
> cantidad=número positivo
>
> Pero el usuario escribe
>
> -5 o deja el campo vacío

¿Qué debería hacer un software bien construido?

Construcción **del software**

La calidad no solamente se revisa al final. También depende de **cómo** construimos el software.

¿Cómo **puede una mala construcción afectar la calidad?**

Python

```python
def f(a,b):
    if a > 0 and b > 0:
        return a*b
    return 0
```

Python

```python
def calcular_total(cantidad, precio):
    if cantidad > 0 and precio > 0:
        return cantidad * precio
    return 0
```

Java

```java
public static int f(int a, int b) {
    if (a > 0 && b > 0) {
        return a * b;
    }
    return 0;
}
```

Java

```java
public static int calcularTotal(int cantidad, int precio) {
    if (cantidad > 0 && precio > 0) {
        return cantidad * precio;
    }
    return 0;
}
```

Revisión del código

Revisar código significa analizarlo para detectar **problemas, inconsistencias u oportunidades de mejora** antes de que se conviertan en fallos.

Al revisar el código nos preguntamos

- ¿Los nombres son claros?
- ¿La responsabilidad está bien definida?
- ¿Se manejan los errores?
- ¿Se contemplan casos límite?
- ¿Corresponde con lo especificado?

Casos límite y condiciones de error

Los casos límite ayudan a identificar comportamientos que pueden pasar **desapercibidos en condiciones normales**. Un análisis sistemático cubre el espectro completo.

La robustez de un programa se evidencia especialmente ante los casos límite e inválidos.

Disponibilidad = 10

Situación **normal**

Disponibilidad = 1

Frontera **relevante**

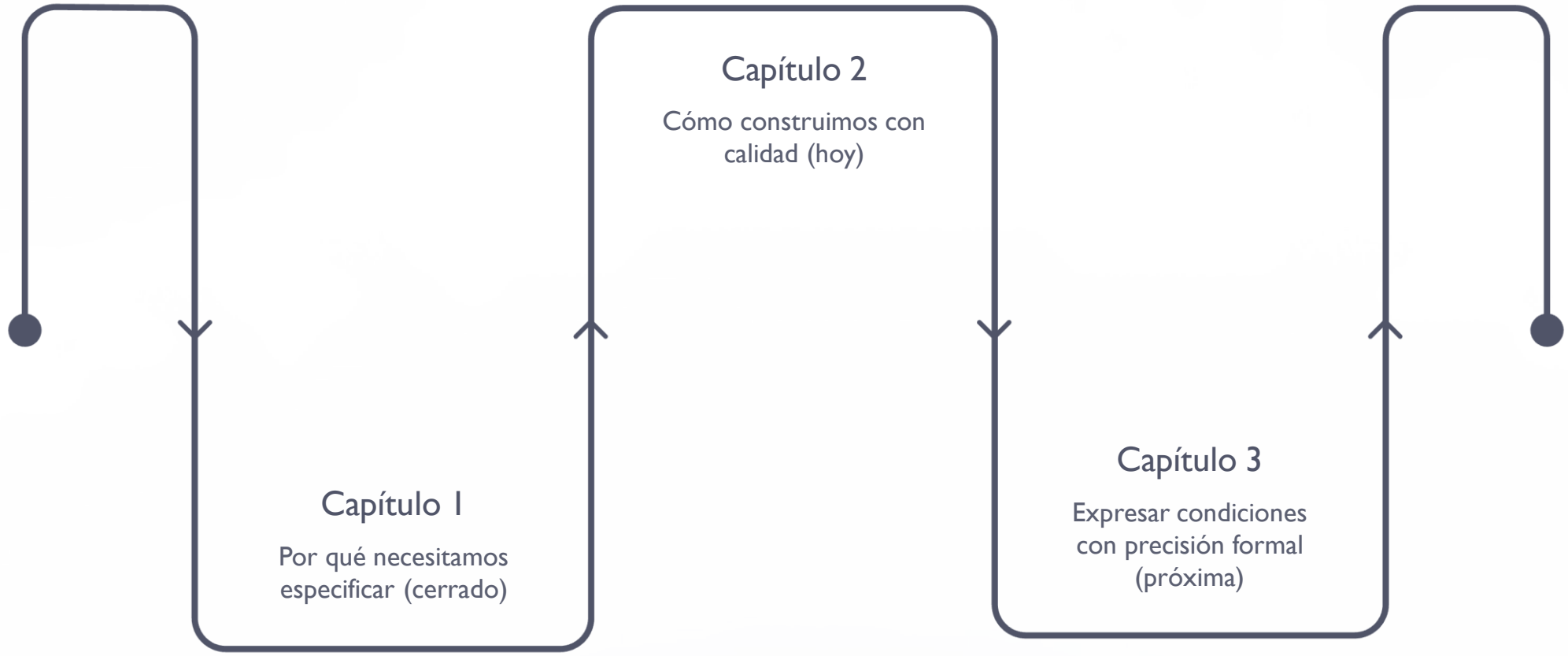
Disponibilidad = 0

Cambia el comportamiento **esperado**

Disponibilidad = -2

Situación **inválida** a manejar

Cierre del Capítulo 1 e inicio del Capítulo 3



En la próxima clase iniciaremos el **Capítulo 3: Fundamentos de especificación formal**. Trabajaremos lógica proposicional, proposiciones, operadores lógicos y tablas de verdad.

ACTIVIDAD PRÁCTICA

Del Código a la Calidad

¿Qué vamos a hacer?

Vamos a recuperar el programa reservar(cupos) que construyeron en la primera clase y analizarlo.

01

Especificación

¿Qué debería cumplir?

Identifiquen: entrada; condiciones; comportamiento esperado; resultado.

02

Validación

¿Las reglas representan lo que necesitamos?
Pregúntense: ¿Tiene sentido reservar con cupos disponibles?
¿Qué ocurre con cupos = 0?
¿Qué debería pasar con cupos < 0?

03

Construcción

¿Cómo está construido el código?
Revisen: nombres claros; código organizado; validación de condiciones; manejo de errores; robustez; facilidad de mantenimiento.

04

Calidad

¿Qué característica es importante?
Calidad funcional
Confiabilidad
Mantenibilidad
Robustez
Justifiquen su elección.

05

Verificación

¿El código cumple lo que especificamos?
Comparen: Lo que debería hacer ⇔ Lo que realmente hace

06

Retomemos las pruebas

Retomemos las pruebas de la clase anterior.
Utilicen: Normal → Límite → Inválido
Expliquen:¿Qué evidencia aporta cada prueba?