Desarrollo Basado en Pruebas (TDD

Material Completo del Curso

Total de Unidades: 6

31 de December de 2025

Índice de Contenidos

Desarrollo Basado en Pruebas (TDD)

Unidad I: Introducción a TDD

Bienvenidos a la primera unidad del curso. Hoy comenzamos nuestro viaje en el mundo del Test-Driven Development.

Competencias y Resultados de Aprendizaje

Competencia Específica CE13

Especificar y ejecutar pruebas de calidad de los prototipos (diseño), simulaciones y los componentes de software desarrollados para garantizar su fidelidad y precisión en todos los niveles (Test units) y garantizando una asertiva aceptación de los usuarios (UCT).

Resultados de Aprendizaje

- Comprender los conceptos fundamentales de TDD y su importancia en el desarrollo de software
- Identificar los beneficios y ventajas de aplicar TDD en proyectos de software
- Comparar TDD con enfoques tradicionales de desarrollo
- Reconocer cuándo es apropiado utilizar TDD en un proyecto
- Entender el impacto de TDD en la calidad del código y la productividad del equipo

Estos resultados de aprendizaje guiarán toda la unidad. Asegúrense de entender cada uno.

Agenda de la Unidad

1. **Conceptos fundamentales de TDD**
    - ¿Qué es TDD?
    - Historia y evolución
    - Principios básicos
2. **Beneficios y ventajas de TDD**
    - Mejora en la calidad del código
    - Reducción de bugs
    - Documentación viva
    - Refactoring seguro
3. **Comparación con enfoques tradicionales**
    - TDD vs Desarrollo tradicional
    - Cuándo usar TDD
    - Cuándo no usar TDD

Esta agenda nos llevará aproximadamente 3 horas de clase.

¿Qué es TDD?

TDD (Test-Driven Development) o **Desarrollo Guiado por Pruebas** es una metodología de desarrollo de software que consiste en escribir primero las pruebas (tests) antes de escribir el código de producción.

> **Definición clave:** TDD invierte el orden tradicional: primero escribes qué debe hacer el código (test), luego escribes el código que cumpla esa expectativa.

Los tres pilares de TDD:

1. **Red:** Escribir una prueba que falle
2. **Green:** Escribir el código mínimo para que pase
3. **Refactor:** Mejorar el código sin cambiar su comportamiento

TDD no es solo testing, es un enfoque de diseño de software.

Historia y Evolución de TDD

Orígenes (1990s)

- Kent Beck desarrolla la metodología
- Influenciado por Extreme Programming (XP)
- Primera publicación: "Test-Driven Development: By Example" (2003)

Evolución (2000s)

- Adopción en comunidades ágiles
- Framework xUnit para múltiples lenguajes
- Integración con CI/CD

Estado Actual

- Estándar en desarrollo ágil
- Herramientas modernas (Jest, pytest, JUnit 5)
- Prácticas avanzadas (BDD, ATDD)

Figuras Clave

- **Kent Beck:** Creador de TDD
- **Martin Fowler:** Refactoring y patrones
- **Robert C. Martin:** Clean Code y TDD

Entender la historia ayuda a comprender por qué TDD es tan importante hoy en día.

Principios Básicos de TDD

1. Las pruebas son código de primera clase

Las pruebas deben mantenerse con el mismo cuidado que el código de producción.

2. Escribe pruebas antes del código

Esto fuerza a pensar en el diseño antes de implementar.

3. Mantén las pruebas simples

Una prueba debe verificar una sola cosa (principio de responsabilidad única).

4. Refactoriza constantemente

Después de hacer pasar la prueba, mejora el código sin cambiar su comportamiento.

5. Mantén el ciclo rápido

Red-Green-Refactor debe ser un ciclo de minutos, no horas.

Estos principios guían todas las prácticas de TDD.

En la Vida Real: Conceptos Fundamentales

Escenario: Desarrollo de un sistema de facturación

Situación tradicional:

- Desarrollas la clase `Factura`
- Implementas métodos: `calcularTotal()`, `aplicarDescuento()`
- Al final, escribes pruebas para verificar que funciona
- **Problema:** Si las pruebas fallan, ya tienes mucho código escrito que puede necesitar cambios grandes

Con TDD:

- Primero escribes: "La factura debe calcular el total sumando los items"
- Escribes la prueba que verifica esto
- Escribes el código mínimo para que pase
- **Ventaja:** El código está diseñado desde el principio para ser testeable

Este ejemplo muestra la diferencia fundamental en el enfoque mental.

Ejemplo: Primer Test en TDD

Supongamos que necesitamos una calculadora simple:

```
// Paso 1: RED - Escribir la prueba (falla porque no existe la clase) describe('Calculadora', () => { test('debe sumar dos números correctamente', () => { const calc = new Calculadora(); expect(calc.sumar(2, 3)).toBe(5); }); }); // Paso 2: GREEN - Código mínimo para pasar class Calculadora { sumar(a, b) { return a + b; } } // Paso 3: REFACTOR - Mejorar (si es necesario) // En este caso, el código ya es simple y claro
```

Observaciones:

- La prueba define el comportamiento esperado
- El código es mínimo pero suficiente
- No hay código innecesario

Este es el ciclo básico que repetiremos constantemente.

Beneficios y Ventajas de TDD

1. Mejora en la Calidad del Código

- Código más limpio y mantenible
- Mejor diseño (alta cohesión, bajo acoplamiento)
- Menos código innecesario

2. Reducción de Bugs

- Detección temprana de errores
- Pruebas como red de seguridad
- Menos regresiones

3. Documentación Viva

- Las pruebas documentan el comportamiento
- Siempre actualizadas
- Ejemplos de uso del código

4. Refactoring Seguro

- Confianza para mejorar código
- Detección inmediata de problemas
- Evolución continua del diseño

Estos beneficios se acumulan con el tiempo. No son inmediatos.

Más Beneficios de TDD

5. Mejor Diseño de Arquitectura

TDD fuerza a pensar en interfaces y dependencias desde el inicio, resultando en mejor arquitectura.

6. Mayor Confianza del Equipo

Los desarrolladores tienen más confianza para hacer cambios sabiendo que las pruebas detectarán problemas.

7. Desarrollo más Rápido a Largo Plazo

Aunque inicialmente puede ser más lento, a largo plazo acelera el desarrollo al reducir tiempo en debugging.

8. Mejor Colaboración

Las pruebas sirven como contrato entre desarrolladores, facilitando la integración de código.

9. Reducción de Deuda Técnica

El código escrito con TDD tiende a tener menos deuda técnica acumulada.

Estos beneficios son especialmente notables en proyectos de mediano y largo plazo.

En la Vida Real: Beneficios de TDD

Caso: Refactorización de un módulo crítico

Sin TDD:

- El módulo tiene 2000 líneas de código
- Nadie se atreve a refactorizarlo por miedo a romper algo
- Se acumulan parches y workarounds
- El código se vuelve cada vez más difícil de mantener

Con TDD:

- El módulo tiene pruebas que cubren todos los casos importantes
- Puedes refactorizar con confianza
- Si algo se rompe, las pruebas lo detectan inmediatamente
- El código mejora continuamente

Resultado: Con TDD, el código mejora con el tiempo. Sin TDD, el código se degrada con el tiempo.

Este es uno de los beneficios más importantes pero menos obvios de TDD.

TDD vs Desarrollo Tradicional

Desarrollo Tradicional

1. Escribir código
2. Probar manualmente
3. Encontrar bugs
4. Corregir bugs
5. Repetir

Problemas:

- Bugs detectados tarde
- Código difícil de testear
- Miedo a refactorizar

Desarrollo con TDD

1. Escribir prueba (RED)
2. Escribir código mínimo (GREEN)
3. Refactorizar (REFACTOR)
4. Repetir

Ventajas:

- Bugs detectados temprano
- Código testeable por diseño
- Refactoring seguro

La diferencia clave está en el orden y la mentalidad.

¿Cuándo Usar TDD?

✅ Ideal para TDD:

- **Lógica de negocio compleja:** Cálculos, validaciones, algoritmos
- **APIs y servicios:** Contratos claros, comportamiento definido
- **Código que cambiará frecuentemente:** Necesitas seguridad para refactorizar
- **Proyectos de largo plazo:** La inversión inicial se paga con el tiempo
- **Equipos colaborativos:** Las pruebas facilitan la integración

❌ No ideal para TDD:

- **Prototipos rápidos:** Necesitas validar ideas rápidamente
- **Interfaces de usuario:** Difícil de testear automáticamente
- **Código legacy sin estructura:** Primero necesitas refactorizar
- **Proyectos muy pequeños:** El overhead puede no valer la pena

TDD es una herramienta poderosa, pero no siempre es la mejor opción.

Errores Comunes al Empezar con TDD

1. Escribir demasiadas pruebas a la vez

Error: Escribir 10 pruebas antes de implementar nada.

Correcto: Una prueba, implementar, refactorizar, siguiente prueba.

2. Pruebas demasiado complejas

Error: Una prueba que verifica 5 cosas diferentes.

Correcto: Una prueba verifica un comportamiento específico.

3. Saltarse el refactor

Error: Hacer pasar la prueba y seguir con la siguiente.

Correcto: Siempre refactorizar después de hacer pasar la prueba.

4. Pruebas que dependen unas de otras

Error: La prueba 2 depende del resultado de la prueba 1.

Correcto: Cada prueba es independiente y puede ejecutarse sola.

Estos errores son muy comunes. Estén atentos para evitarlos.

Buenas Prácticas en TDD

1. Nombres descriptivos para las pruebas

```
// ❌ Mal test('test1', () => { ... }); // ✅ Bien test('debe calcular el descuento del 10% cuando el total es mayor a 1000', () => { ... });
```

2. Arrange-Act-Assert (AAA)

```
test('ejemplo AAA', () => { // Arrange: Preparar const calculadora = new Calculadora(); // Act: Ejecutar const resultado = calculadora.sumar(2, 3); // Assert: Verificar expect(resultado).toBe(5); });
```

3. Una aserción por prueba (idealmente)

Facilita identificar qué falló cuando la prueba no pasa.

4. Mantén las pruebas rápidas

Las pruebas deben ejecutarse en milisegundos, no segundos.

Estas prácticas hacen que TDD sea más efectivo y mantenible.

Práctica Guiada #1: Calculadora Simple

Objetivo:

Implementar una calculadora básica usando TDD que pueda sumar, restar, multiplicar y dividir.

Paso 1: Configurar el entorno

```
// Crear archivo calculadora.test.js // Instalar dependencias: npm install --save-dev jest
```

Paso 2: Escribir primera prueba (RED)

```
describe('Calculadora', () => { test('debe sumar dos números positivos', () => { const calc = new Calculadora(); expect(calc.sumar(5, 3)).toBe(8); }); });
```

Paso 3: Ejecutar y verificar que falla

Ejecuta: `npm test` - Debe fallar porque Calculadora no existe.

Vamos paso a paso. No se apresuren.

Práctica Guiada #1: Continuación

Paso 4: Implementar código mínimo (GREEN)

```
// Crear archivo calculadora.js class Calculadora { sumar(a, b) { return a + b; } } module.exports = Calculadora;
```

Paso 5: Ejecutar prueba - Debe pasar

Ejecuta: `npm test` - Ahora debe pasar ✅

Paso 6: Agregar siguiente funcionalidad (restar)

```
// Primero la prueba test('debe restar dos números', () => { const calc = new Calculadora(); expect(calc.restar(10, 4)).toBe(6); }); // Luego la implementación restar(a, b) { return a - b; }
```

Paso 7: Repetir para multiplicar y dividir

Sigue el mismo patrón: RED → GREEN → REFACTOR

Este es el ciclo básico. Repítanlo para cada nueva funcionalidad.

Práctica Guiada #2: Validador de Email

Objetivo:

Crear un validador de emails usando TDD que verifique el formato correcto.

Paso 1: Escribir pruebas primero

```
describe('ValidadorEmail', () => { test('debe aceptar un email válido', () => { const validador = new ValidadorEmail(); expect(validador.esValido('usuario@dominio.com')).toBe(true); }); test('debe rechazar un email sin @', () => { const validador = new ValidadorEmail(); expect(validador.esValido('usuariodominio.com')).toBe(false); }); test('debe rechazar un email sin dominio', () => { const validador = new ValidadorEmail(); expect(validador.esValido('usuario@')).toBe(false); }); });
```

Escriban todas las pruebas primero, luego implementen.

Práctica Guiada #2: Solución

Implementación del ValidadorEmail

```
class ValidadorEmail { esValido(email) { if (!email || typeof email !== 'string') { return false; } // Verificar que tenga @ if (!email.includes('@')) { return false; } const partes = email.split('@'); // Debe tener exactamente dos partes if (partes.length !== 2) { return false; } const [usuario, dominio] = partes; // Usuario y dominio no pueden estar vacíos if (!usuario || !dominio) { return false; } // Dominio debe tener al menos un punto if (!dominio.includes('.')) { return false; } return true; } }
```

Refactorización sugerida:

Podrías usar expresiones regulares para hacer el código más conciso, pero mantén la legibilidad.

Noten cómo las pruebas guiaron el diseño de la clase.

Práctica Evaluada

Enunciado:

Implementa una clase `GestorInventario` usando TDD que gestione productos en un inventario. Debe cumplir con los siguientes requisitos:

1. Agregar un producto con nombre, precio y cantidad
2. Obtener un producto por nombre
3. Actualizar la cantidad de un producto
4. Eliminar un producto
5. Calcular el valor total del inventario (suma de precio × cantidad de todos los productos)
6. Validar que no se puedan agregar productos con precio negativo
7. Validar que no se puedan agregar productos con cantidad negativa

Criterios de Evaluación:

- **Uso correcto de TDD (40%):** Pruebas escritas antes del código, ciclo RED-GREEN-REFACTOR seguido
- **Calidad de las pruebas (30%):** Nombres descriptivos, una aserción por prueba, cobertura completa
- **Calidad del código (20%):** Código limpio, bien estructurado, sin código innecesario

- **Funcionalidad (10%):** Todos los requisitos implementados correctamente

Rúbrica:

- **Excelente (90-100):** TDD aplicado perfectamente, pruebas exhaustivas, código impecable
- **Bueno (75-89):** TDD aplicado correctamente, pruebas adecuadas, código funcional
- **Satisfactorio (60-74):** TDD aplicado parcialmente, algunas pruebas, código funciona
- **Necesita mejorar (<60):** TDD no aplicado, pocas o ninguna prueba, código incompleto

Esta práctica evalúa la comprensión completa de TDD.

Mini-Quiz

1. ¿Qué significa TDD?

a) Test-Driven Development
b) Test-Defined Development
c) Test-Directed Development
d) Test-Design Development

2. ¿Cuál es el orden correcto del ciclo TDD?

a) Green → Red → Refactor
b) Red → Green → Refactor
c) Refactor → Red → Green
d) Green → Refactor → Red

3. ¿Cuál es el principal beneficio de TDD?

a) Escribir código más rápido
b) Mejor calidad del código y menos bugs
c) Menos código en general
d) No necesita documentación

Mini-Quiz (Continuación)

4. ¿Cuándo NO es recomendable usar TDD?

a) En proyectos grandes
b) En prototipos rápidos
c) En código legacy
d) b y c

5. ¿Qué significa "Red" en el ciclo TDD?

a) Escribir código que funciona
b) Escribir una prueba que falla
c) Refactorizar el código
d) Eliminar código rojo

6. Verdadero o Falso: TDD es solo escribir pruebas.

Verdadero
Falso - TDD es un enfoque de diseño

Mini-Quiz (Continuación)

7. ¿Cuál es una buena práctica en TDD?

a) Escribir todas las pruebas antes de implementar
b) Una prueba por vez, implementar, refactorizar
c) Escribir código primero y luego pruebas
d) Solo escribir pruebas para código complejo

8. ¿Qué patrón se recomienda para estructurar pruebas?

a) AAA (Arrange-Act-Assert)
b) BDD (Behavior-Driven Development)
c) DDD (Domain-Driven Design)
d) SOLID

9. Verdadero o Falso: Las pruebas en TDD deben ser independientes entre sí.

Verdadero
Falso

Respuestas del Mini-Quiz

1. **a) Test-Driven Development** - Desarrollo Guiado por Pruebas
2. **b) Red → Green → Refactor** - Este es el ciclo fundamental
3. **b) Mejor calidad del código y menos bugs** - Aunque puede ser más lento inicialmente
4. **d) b y c** - Prototipos rápidos y código legacy sin estructura
5. **b) Escribir una prueba que falla** - La prueba debe fallar primero
6. **Falso** - TDD es un enfoque de diseño, no solo testing
7. **b) Una prueba por vez** - Mantener el ciclo rápido
8. **a) AAA** - Arrange-Act-Assert es el patrón recomendado
9. **Verdadero** - Cada prueba debe poder ejecutarse independientemente

Si tuvieron dudas en alguna pregunta, revisen ese tema.

Resumen de la Unidad

Puntos Clave:

- **TDD** es Desarrollo Guiado por Pruebas: escribir pruebas antes del código
- El ciclo fundamental es **Red-Green-Refactor**
- TDD mejora la calidad del código, reduce bugs y permite refactoring seguro
- Las pruebas sirven como documentación viva del código
- TDD es ideal para lógica de negocio y código que cambiará frecuentemente
- No es recomendable para prototipos rápidos o interfaces de usuario complejas
- Buenas prácticas: nombres descriptivos, patrón AAA, pruebas independientes
- TDD requiere disciplina pero los beneficios se acumulan con el tiempo

Este resumen cubre los conceptos más importantes de la unidad.

Tarea para Casa

Actividad:

Implementa un sistema de gestión de biblioteca usando TDD con las siguientes funcionalidades:

1. Clase `Libro` con propiedades: título, autor, ISBN, disponible
2. Clase `Biblioteca` que pueda:
    - Agregar libros
    - Buscar libro por ISBN
    - Prestar un libro (marcar como no disponible)
    - Devolver un libro (marcar como disponible)
    - Listar todos los libros disponibles
    - Validar que no se pueda prestar un libro ya prestado

Entregables:

- Código fuente completo (archivos .js y .test.js)
- Capturas de pantalla mostrando que todas las pruebas pasan
- Documento breve (1-2 páginas) explicando:
    - El proceso TDD seguido
    - Decisiones de diseño tomadas

- Reflexiones sobre la experiencia

Fecha de entrega:

Una semana después de esta clase

Formato de entrega:

Repositorio Git (GitHub/GitLab) con commits que muestren el proceso TDD, o archivo ZIP con todos los archivos.

Esta tarea les permitirá practicar TDD en un proyecto completo.

Recursos Adicionales

📚 Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Videos sobre TDD y sus beneficios
- Documentación de frameworks de testing
- Ejemplos de proyectos con TDD
- Artículos y tutoriales

Ver Recursos Adicionales →

¡Gracias!

Unidad I: Introducción a TDD

Próxima clase: Unidad II - Ciclo de desarrollo en TDD

Preguntas y dudas

Asegúrense de hacer la tarea y venir preparados para la siguiente unidad.

Recursos Adicionales - Unidad 01

Introducción a TDD - Enlaces, videos y material complementario

📚 Documentación y Libros

[Test-Driven Development: By Example - Kent Beck Libro]

El libro fundacional de TDD escrito por Kent Beck, el creador de la metodología. Lectura esencial para entender TDD.

[Is TDD Dead? - Martin Fowler Artículo]

Serie de debates sobre TDD con expertos como Kent Beck, David Heinemeier Hansson y Martin Fowler.

[TDD en Agile Alliance Referencia]

Definición y recursos sobre TDD desde la perspectiva de metodologías ágiles.

🎥 Videos Educativos

[TDD: Where Did It All Go Wrong? - Ian Cooper Video]

Charla sobre los malentendidos comunes de TDD y cómo aplicarlo correctamente.

[Test-Driven Development (TDD) Tutorial Tutorial]

Tutorial práctico sobre cómo aplicar TDD en un proyecto real.

💡 Artículos Especializados

[The Cycles of TDD - Robert C. Martin Artículo]

Explicación detallada del ciclo Red-Green-Refactor por "Uncle Bob" Martin.

[Common Mistakes with Testing Artículo]

Errores comunes al escribir pruebas y cómo evitarlos.

💡 Consejo de Estudio

Para esta unidad introductoria: 1) Lee el libro de Kent Beck sobre TDD para entender los fundamentos, 2) Observa videos de expertos aplicando TDD en proyectos reales, 3) Comprende que TDD es más un enfoque de diseño que solo testing, 4) Practica el ciclo Red-Green-Refactor con ejercicios simples. La mentalidad es tan importante como la técnica.

Desarrollo Basado en Pruebas (TDD)

Unidad II: Ciclo de desarrollo en TDD

Bienvenidos a la segunda unidad. Hoy profundizaremos en el ciclo fundamental de TDD: Red-Green-Refactor.

Competencias y Resultados de Aprendizaje

Competencia Específica CE13

Especificar y ejecutar pruebas de calidad de los prototipos (diseño), simulaciones y los componentes de software desarrollados para garantizar su fidelidad y precisión en todos los niveles (Test units) y garantizando una asertiva aceptación de los usuarios (UCT).

Resultados de Aprendizaje

- Dominar el ciclo Red-Green-Refactor de TDD
- Aplicar correctamente cada fase del ciclo TDD
- Escribir pruebas automatizadas que fallen primero (Red)
- Implementar código mínimo para hacer pasar las pruebas (Green)
- Refactorizar código manteniendo las pruebas pasando (Refactor)

El ciclo Red-Green-Refactor es el corazón de TDD. Deben dominarlo completamente.

Agenda de la Unidad

1. **Red: Prueba automatizada**
    - Escribir prueba que falle primero
    - Definir comportamiento esperado
    - Verificar que la prueba falla por la razón correcta
2. **Green: Código mínimo**
    - Implementar código mínimo para pasar
    - No optimizar prematuramente
    - Verificar que la prueba pasa
3. **Refactor: Mejorar código**
    - Mejorar sin cambiar comportamiento
    - Mantener pruebas pasando
    - Principios de código limpio

Esta agenda cubre las tres fases fundamentales del ciclo TDD.

El Ciclo Red-Green-Refactor

🔴 RED

Escribir una prueba que falle

Define qué debe hacer el código

🟢 GREEN

Escribir código mínimo para pasar

Implementa solo lo necesario

🔵 REFACTOR

Mejorar el código sin cambiar comportamiento

Mantén las pruebas pasando

Ciclo Continuo
Repetir para cada nueva funcionalidad

Este ciclo se repite constantemente durante el desarrollo.

Fase RED: Escribir Prueba que Falle

Objetivo:

Escribir una prueba automatizada que defina el comportamiento deseado y que **falle** porque la funcionalidad aún no existe.

Pasos:

1. Pensar en qué debe hacer el código
2. Escribir la prueba que verifica ese comportamiento
3. Ejecutar la prueba y verificar que **falla**
4. Verificar que falla por la razón correcta (no por un error de sintaxis)

¿Por qué debe fallar primero?

- Confirma que la prueba realmente verifica algo
- Evita falsos positivos
- Define claramente el comportamiento esperado

La fase RED es crucial. Si la prueba no falla primero, puede que no esté probando nada.

Ejemplo: Fase RED

Queremos crear una función que calcule el área de un círculo:

```
// Paso 1: Escribir la prueba (RED) describe('CalculadoraGeometrica', () => { test('debe calcular el área de un círculo dado su radio', () => { const calc = new CalculadoraGeometrica(); const area = calc.areaCirculo(5); expect(area).toBeCloseTo(78.54, 2); // π * r² = π * 25 ≈ 78.54 }); }); // Ejecutar: npm test // Resultado esperado: ❌ FAIL - CalculadoraGeometrica is not defined
```

> **Importante:** La prueba debe fallar porque la clase no existe, no por un error de sintaxis o lógica en la prueba misma.

Este es un ejemplo claro de cómo escribir una prueba en la fase RED.

Fase GREEN: Código Mínimo para Pasar

Objetivo:

Escribir el código **mínimo necesario** para hacer que la prueba pase. No optimizar, no agregar funcionalidades extra.

Principios:

- **Simplicidad:** Código más simple que funcione
- **Mínimo:** Solo lo necesario para pasar la prueba
- **Sin optimización prematura:** No pensar en casos futuros
- **Verificación:** Ejecutar la prueba y confirmar que pasa

¿Por qué código mínimo?

- Evita código innecesario
- Mantiene el foco en el comportamiento actual
- Facilita el refactoring posterior

La fase GREEN requiere disciplina para no agregar código innecesario.

Ejemplo: Fase GREEN

Ahora implementamos el código mínimo para hacer pasar la prueba:

```
// Paso 2: Implementar código mínimo (GREEN) class CalculadoraGeometrica { areaCirculo(radio) { return Math.PI * radio * radio; } } // Ejecutar: npm test // Resultado esperado: ✅ PASS
```

Observaciones:

- Implementamos solo lo necesario
- No agregamos validaciones ni casos especiales aún
- El código es simple y directo
- La prueba ahora pasa ✅

Este código es mínimo pero suficiente. No hay código innecesario.

Fase REFACTOR: Mejorar el Código

Objetivo:

Mejorar la calidad del código **sin cambiar su comportamiento**. Las pruebas deben seguir pasando.

¿Qué mejorar?

- **Nombres:** Variables y funciones más descriptivas
- **Estructura:** Organizar mejor el código
- **Duplicación:** Eliminar código duplicado
- **Complejidad:** Reducir complejidad ciclomática
- **Principios SOLID:** Aplicar principios de diseño

Regla de oro:

Si las pruebas dejan de pasar durante el refactor, has cambiado el comportamiento. Revierte y vuelve a intentar.

El refactor es donde mejoramos el código sin romper funcionalidad.

Ejemplo: Fase REFACTOR

Mejoramos el código manteniendo las pruebas pasando:

```
// Paso 3: Refactorizar (REFACTOR) class CalculadoraGeometrica { areaCirculo(radio) { const PI = Math.PI; const radioAlCuadrado = Math.pow(radio, 2); return PI * radioAlCuadrado; } } // O mejor aún, usando constantes: class CalculadoraGeometrica { static PI = Math.PI; areaCirculo(radio) { return this.constructor.PI * Math.pow(radio, 2); } } // Ejecutar: npm test // Resultado esperado: ✅ PASS (comportamiento igual, código mejor)
```

> **Nota:** El comportamiento es idéntico, pero el código es más legible y mantenible.

El refactor mejora el código sin cambiar qué hace.

Ciclo Completo: Ejemplo Práctico

Escenario: Crear una clase ValidarContraseña

🔴 **RED - Prueba 1:**

```
test('debe rechazar contraseña menor a 8 caracteres', () => { const validador = new ValidarContraseña(); expect(validador.esValida('abc123')).toBe(false); });
```

🟢 **GREEN - Implementación mínima:**

```
class ValidarContraseña { esValida(contraseña) { return contraseña.length >= 8; } }
```

🔵 **REFACTOR - Mejorar:**

```
class ValidarContraseña { static LONGITUD_MINIMA = 8; esValida(contraseña) { return contraseña && contraseña.length >= this.constructor.LONGITUD_MINIMA; } }
```

Este ejemplo muestra el ciclo completo en acción.

Continuación del Ciclo

Agregando más funcionalidad:

🔴 **RED - Prueba 2:**

```
test('debe rechazar contraseña sin mayúsculas', () => { const validador = new ValidarContraseña(); expect(validador.esValida('abc12345')).toBe(false); });
```

🟢 **GREEN - Actualizar implementación:**

```
esValida(contraseña) { if (!contraseña || contraseña.length < 8) return false; return /[A-Z]/.test(contraseña); }
```

🔵 **REFACTOR - Mejorar estructura:**

```
esValida(contraseña) { return this.tieneLongitudMinima(contraseña) && this.tieneMayuscula(contraseña); } tieneLongitudMinima(contraseña) { return contraseña && contraseña.length >= 8; } tieneMayuscula(contraseña) { return /[A-Z]/.test(contraseña); }
```

El ciclo continúa agregando funcionalidad incrementalmente.

En la Vida Real: Fase RED

Escenario: Sistema de descuentos en e-commerce

Situación: Necesitas implementar un sistema que aplique descuentos según el monto total de la compra.

Fase RED - Pensar primero:

- ¿Qué descuentos necesito? (5% para compras > $100, 10% para > $500)
- ¿Qué casos límite hay? (compra exactamente $100, $500)
- ¿Qué validaciones necesito? (montos negativos, cero)

```
// RED: Escribir todas las pruebas primero test('debe aplicar 5% de descuento para compras mayores a $100', () => { const calculadora = new CalculadoraDescuentos(); expect(calculadora.calcularDescuento(150)).toBe(7.5); // 5% de 150 }); test('debe aplicar 10% de descuento para compras mayores a $500', () => { const calculadora = new CalculadoraDescuentos(); expect(calculadora.calcularDescuento(600)).toBe(60); // 10% de 600 }); test('no debe aplicar descuento para compras menores a $100', () => { const calculadora = new CalculadoraDescuentos(); expect(calculadora.calcularDescuento(50)).toBe(0); });
```

En la vida real, pensar primero en todas las pruebas ayuda a entender mejor el problema.

En la Vida Real: Fase GREEN

Implementación mínima para pasar todas las pruebas:

```
class CalculadoraDescuentos { calcularDescuento(monto) { if (monto <= 0) return 0; if (monto > 500) return monto * 0.10; if (monto > 100) return monto * 0.05; return 0; } }
```

Observaciones:

- Código simple y directo
- Pasa todas las pruebas
- No hay optimización prematura
- Fácil de entender

Ahora que las pruebas pasan, podemos refactorizar con confianza.

El código mínimo funciona y pasa todas las pruebas. Ahora podemos mejorarlo.

En la Vida Real: Fase REFACTOR

Mejorando el código manteniendo las pruebas pasando:

```
class CalculadoraDescuentos { static UMBRAL_DESCUENTO_BASICO = 100; static UMBRAL_DESCUENTO_PREMIUM = 500; static PORCENTAJE_DESCUENTO_BASICO = 0.05; static PORCENTAJE_DESCUENTO_PREMIUM = 0.10; calcularDescuento(monto) { if (!this.esMontoValido(monto)) return 0; if (this.esDescuentoPremium(monto)) { return this.aplicarDescuento(monto, this.constructor.PORCENTAJE_DESCUENTO_PREMIUM); } if (this.esDescuentoBasico(monto)) { return this.aplicarDescuento(monto, this.constructor.PORCENTAJE_DESCUENTO_BASICO); } return 0; } esMontoValido(monto) { return monto > 0; } esDescuentoPremium(monto) { return monto > this.constructor.UMBRAL_DESCUENTO_PREMIUM; } esDescuentoBasico(monto) { return monto > this.constructor.UMBRAL_DESCUENTO_BASICO; } aplicarDescuento(monto, porcentaje) { return monto * porcentaje; } }
```

Mejoras:

- Código más legible y mantenible
- Constantes para valores mágicos
- Métodos pequeños y con responsabilidad única
- **Las pruebas siguen pasando** ✅

El refactor mejora significativamente el código sin cambiar su comportamiento.

Errores Comunes en el Ciclo TDD

1. Saltarse la fase RED

Error: Escribir código primero y luego la prueba.

Correcto: Siempre escribir la prueba primero y verificar que falla.

2. Escribir código excesivo en GREEN

Error: Implementar funcionalidades que la prueba no requiere.

Correcto: Solo el código mínimo necesario para pasar la prueba.

3. Saltarse el REFACTOR

Error: Hacer pasar la prueba y seguir con la siguiente.

Correcto: Siempre refactorizar después de hacer pasar la prueba.

4. Cambiar comportamiento en REFACTOR

Error: Agregar nuevas funcionalidades durante el refactor.

Correcto: Refactorizar solo mejora código, no cambia comportamiento.

5. Ciclos muy largos

Error: Pasar mucho tiempo en cada fase.

Correcto: Mantener ciclos cortos (minutos, no horas).

Estos errores son muy comunes. Estén atentos para evitarlos.

Buenas Prácticas del Ciclo TDD

1. Mantén el ciclo rápido

Cada iteración del ciclo debe tomar minutos, no horas. Esto mantiene el flujo y la concentración.

2. Una funcionalidad a la vez

No intentes implementar múltiples funcionalidades en un solo ciclo. Mantén el foco.

3. Verifica que la prueba falla primero

En RED, asegúrate de que la prueba realmente falla y por la razón correcta.

4. No optimices prematuramente

En GREEN, implementa solo lo necesario. La optimización viene en REFACTOR.

5. Refactoriza constantemente

No acumules deuda técnica. Refactoriza después de cada ciclo GREEN.

6. Confía en las pruebas

Si las pruebas pasan, puedes refactorizar con confianza. Si fallan, sabes exactamente qué rompiste.

Estas prácticas hacen que TDD sea efectivo y sostenible.

Práctica Guiada #1: Calculadora de Impuestos

Objetivo:

Implementar una calculadora de impuestos usando el ciclo Red-Green-Refactor completo.

Requisitos:

- Calcular impuesto del 10% para montos menores a $1000
- Calcular impuesto del 15% para montos mayores o iguales a $1000
- Validar que el monto sea positivo

Paso 1: RED - Escribir primera prueba

```
describe('CalculadoraImpuestos', () => { test('debe calcular 10% de impuesto para montos menores a $1000', () => { const calc = new CalculadoraImpuestos(); expect(calc.calcularImpuesto(500)).toBe(50); // 10% de 500 }); });
```

Paso 2: Ejecutar y verificar que falla

Ejecuta: `npm test` - Debe fallar porque CalculadoraImpuestos no existe.

Vamos paso a paso siguiendo el ciclo completo.

Práctica Guiada #1: Continuación

Paso 3: GREEN - Implementar código mínimo

```
class CalculadoraImpuestos { calcularImpuesto(monto) { return monto * 0.10; } }
```

Paso 4: Ejecutar - Debe pasar ✅

Ejecuta: `npm test` - Ahora debe pasar.

Paso 5: REFACTOR - Mejorar código

```
class CalculadoraImpuestos { static TASA_IMPUESTO_BASICO = 0.10; calcularImpuesto(monto) { return monto * this.constructor.TASA_IMPUESTO_BASICO; } }
```

Paso 6: Agregar siguiente funcionalidad (RED)

```
test('debe calcular 15% de impuesto para montos >= $1000', () => { const calc = new CalculadoraImpuestos(); expect(calc.calcularImpuesto(1000)).toBe(150); // 15% de 1000 });
```

Continúen el ciclo para completar todos los requisitos.

Práctica Guiada #2: Sistema de Puntos

Objetivo:

Crear un sistema de puntos de fidelidad usando TDD completo.

Requisitos:

- Acumular 1 punto por cada $10 gastados
- Canjear puntos (1 punto = $0.10 de descuento)
- Validar que no se puedan canjear más puntos de los disponibles

RED - Escribir todas las pruebas primero:

```
describe('SistemaPuntos', () => { test('debe acumular 1 punto por cada $10 gastados', () => { const sistema = new SistemaPuntos(); sistema.agregarCompra(50); expect(sistema.obtenerPuntos()).toBe(5); // 50 / 10 = 5 puntos }); test('debe canjear puntos por descuento', () => { const sistema = new SistemaPuntos(); sistema.agregarCompra(100); // 10 puntos const descuento = sistema.canjearPuntos(5); expect(descuento).toBe(0.50); // 5 puntos * $0.10 expect(sistema.obtenerPuntos()).toBe(5); // Quedan 5 puntos }); test('no debe permitir canjear más puntos de los disponibles', () => { const sistema = new SistemaPuntos(); sistema.agregarCompra(50); // 5 puntos expect(() => sistema.canjearPuntos(10)).toThrow('Puntos insuficientes'); }); });
```

Escriban todas las pruebas primero, luego implementen.

Práctica Guiada #2: Solución

GREEN - Implementación mínima:

```
class SistemaPuntos { constructor() { this.puntos = 0; } agregarCompra(monto) { this.puntos += Math.floor(monto / 10); } obtenerPuntos() { return this.puntos; } canjearPuntos(cantidad) { if (cantidad > this.puntos) { throw new Error('Puntos insuficientes'); } this.puntos -= cantidad; return cantidad * 0.10; } }
```

REFACTOR - Mejorar código:

```
class SistemaPuntos { static PUNTOS_POR_DOLAR = 10; static VALOR_PUNTO = 0.10; constructor() { this.puntos = 0; } agregarCompra(monto) { this.puntos += this.calcularPuntos(monto); } calcularPuntos(monto) { return Math.floor(monto / this.constructor.PUNTOS_POR_DOLAR); } obtenerPuntos() { return this.puntos; } canjearPuntos(cantidad) { this.validarPuntosSuficientes(cantidad); this.puntos -= cantidad; return this.calcularDescuento(cantidad); } validarPuntosSuficientes(cantidad) { if (cantidad > this.puntos) { throw new Error('Puntos insuficientes'); } } calcularDescuento(cantidad) { return cantidad * this.constructor.VALOR_PUNTO; } }
```

Noten cómo el refactor mejora la legibilidad y mantenibilidad.

Práctica Evaluada

Enunciado:

Implementa una clase `GestorTareas` usando el ciclo completo Red-Green-Refactor que gestione una lista de tareas con las siguientes funcionalidades:

1. Agregar una tarea con título y descripción
2. Marcar una tarea como completada
3. Eliminar una tarea
4. Obtener todas las tareas pendientes
5. Obtener todas las tareas completadas
6. Contar el total de tareas
7. Validar que no se puedan agregar tareas con título vacío

Criterios de Evaluación:

- **Ciclo Red-Green-Refactor (50%):** Cada funcionalidad debe seguir el ciclo completo, pruebas escritas primero, código mínimo, refactor aplicado
- **Calidad de las pruebas (25%):** Pruebas claras, una aserción por prueba, nombres descriptivos
- **Calidad del código refactorizado (15%):** Código limpio, bien estructurado, sin duplicación
- **Funcionalidad completa (10%):** Todos los requisitos implementados correctamente

Rúbrica:

- **Excelente (90-100):** Ciclo perfectamente aplicado, código impecable después del refactor
- **Bueno (75-89):** Ciclo aplicado correctamente, código funcional y bien refactorizado
- **Satisfactorio (60-74):** Ciclo aplicado parcialmente, algunas funcionalidades sin refactor
- **Necesita mejorar (<60):** Ciclo no aplicado, código sin refactorizar

Esta práctica evalúa el dominio completo del ciclo Red-Green-Refactor.

Mini-Quiz

1. ¿Cuál es el orden correcto del ciclo TDD?

a) Green → Red → Refactor
b) Red → Green → Refactor
c) Refactor → Red → Green
d) Red → Refactor → Green

2. En la fase RED, ¿por qué es importante que la prueba falle primero?

a) Para ahorrar tiempo
b) Para confirmar que la prueba realmente verifica algo
c) Para evitar escribir código
d) No es importante

3. En la fase GREEN, ¿qué principio debemos seguir?

a) Escribir código perfecto desde el inicio
b) Escribir código mínimo para hacer pasar la prueba
c) Optimizar el código inmediatamente
d) Agregar funcionalidades extra

Mini-Quiz (Continuación)

4. En la fase REFACTOR, ¿qué podemos hacer?

a) Cambiar el comportamiento del código
b) Agregar nuevas funcionalidades
c) Mejorar el código sin cambiar su comportamiento
d) Eliminar pruebas

5. ¿Qué pasa si las pruebas fallan durante el REFACTOR?

a) Es normal, podemos continuar
b) Hemos cambiado el comportamiento, debemos revertir
c) Debemos escribir nuevas pruebas
d) Debemos eliminar las pruebas que fallan

6. Verdadero o Falso: El ciclo TDD debe ser rápido (minutos, no horas).

Verdadero
Falso

Mini-Quiz (Continuación)

7. ¿Cuál es un error común en la fase GREEN?

a) Escribir código mínimo
b) Implementar funcionalidades que la prueba no requiere
c) Verificar que la prueba pasa
d) Seguir el ciclo correctamente

8. ¿Qué debemos hacer después de hacer pasar una prueba?

a) Escribir la siguiente prueba inmediatamente
b) Refactorizar el código
c) Optimizar el código
d) Documentar el código

9. Verdadero o Falso: En REFACTOR podemos agregar nuevas funcionalidades.

Verdadero
Falso - REFACTOR solo mejora código existente

10. ¿Cuál es la duración ideal de un ciclo completo Red-Green-Refactor?

a) Horas
b) Minutos
c) Días
d) Semanas

Respuestas del Mini-Quiz

1. **b) Red → Green → Refactor** - Este es el orden correcto del ciclo
2. **b) Para confirmar que la prueba realmente verifica algo** - Evita falsos positivos
3. **b) Escribir código mínimo para hacer pasar la prueba** - Sin optimización prematura
4. **c) Mejorar el código sin cambiar su comportamiento** - REFACTOR solo mejora
5. **b) Hemos cambiado el comportamiento, debemos revertir** - REFACTOR no debe cambiar comportamiento
6. **Verdadero** - Los ciclos deben ser rápidos para mantener el flujo
7. **b) Implementar funcionalidades que la prueba no requiere** - Error común
8. **b) Refactorizar el código** - Siempre refactorizar después de GREEN
9. **Falso** - REFACTOR solo mejora código, no agrega funcionalidades
10. **b) Minutos** - Los ciclos deben ser rápidos

Si tuvieron dudas, revisen esos conceptos antes de continuar.

Resumen de la Unidad

Puntos Clave:

- El ciclo **Red-Green-Refactor** es el corazón de TDD
- **RED:** Escribir prueba que falle primero - define comportamiento esperado
- **GREEN:** Código mínimo para pasar - sin optimización prematura
- **REFACTOR:** Mejorar código sin cambiar comportamiento - mantener pruebas pasando
- El ciclo debe ser rápido (minutos, no horas)
- Cada fase tiene un propósito específico y no debe saltarse
- Las pruebas son la red de seguridad durante el refactor
- El ciclo se repite constantemente durante el desarrollo
- Confiar en las pruebas permite refactorizar con seguridad

Este resumen cubre los conceptos fundamentales del ciclo TDD.

Tarea para Casa

Actividad:

Implementa un sistema de gestión de inventario usando el ciclo completo Red-Green-Refactor con las siguientes funcionalidades:

1. Agregar productos con nombre, precio y cantidad
2. Buscar producto por nombre
3. Actualizar cantidad de un producto
4. Eliminar producto
5. Calcular valor total del inventario
6. Obtener productos con stock bajo (< 10 unidades)
7. Validar que no se puedan agregar productos con precio negativo

Requisitos Específicos:

- Cada funcionalidad debe seguir el ciclo completo: RED → GREEN → REFACTOR
- Documentar cada fase del ciclo con comentarios
- Incluir capturas de pantalla mostrando las pruebas fallando (RED) y pasando (GREEN)
- Mostrar el código antes y después del refactor

Entregables:

- Código fuente completo con commits que muestren el proceso

- Documento explicando el proceso seguido en cada funcionalidad
- Capturas de pantalla del proceso Red-Green-Refactor
- Reflexión sobre la experiencia (1-2 páginas)

Fecha de entrega:

Una semana después de esta clase

Esta tarea les permitirá practicar el ciclo completo múltiples veces.

Recursos Adicionales

📚 Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Videos sobre el ciclo RED-GREEN-REFACTOR
- Ejemplos prácticos de TDD
- Frameworks de testing
- Mejores prácticas

Ver Recursos Adicionales →

¡Gracias!

Unidad II: Ciclo de desarrollo en TDD

Próxima clase: Unidad III - Diseño de pruebas

Preguntas y dudas

Practiquen el ciclo Red-Green-Refactor hasta que sea natural.

Recursos Adicionales - Unidad 02

Ciclo de Desarrollo en TDD - Enlaces, videos y material complementario

📚 Documentación

[The Cycles of TDD - Uncle Bob Artículo]

Explicación detallada del ciclo Red-Green-Refactor y sus variaciones.

[TDD: Red, Green, Refactor - James Shore Tutorial]

Tutorial interactivo sobre el ciclo de TDD con ejemplos prácticos.

🎥 Videos Educativos

[Red-Green-Refactor en Acción Video]

Demostración práctica del ciclo completo de TDD desarrollando una funcionalidad.

[TDD: Flujo de Trabajo Completo Tutorial]

Video tutorial mostrando el flujo completo de trabajo con TDD.

💡 Consejo de Estudio

Para dominar el ciclo de TDD: 1) Practica el ciclo Red-Green-Refactor con problemas simples primero, 2) No te saltes el paso de Refactor - es crucial para mantener código limpio, 3) Escribe la prueba más simple que pueda fallar, 4) Escribe el código más simple que haga pasar la prueba. La disciplina del ciclo es lo que hace que TDD funcione.

Desarrollo Basado en Pruebas (TDD)

Unidad III: Diseño de pruebas

Bienvenidos a la tercera unidad. Hoy aprenderemos cómo diseñar pruebas efectivas en TDD.

Competencias y Resultados de Aprendizaje

Competencia Específica CE13

Especificar y ejecutar pruebas de calidad de los prototipos (diseño), simulaciones y los componentes de software desarrollados para garantizar su fidelidad y precisión en todos los niveles (Test units) y garantizando una asertiva aceptación de los usuarios (UCT).

Resultados de Aprendizaje

- Identificar las características de una buena prueba en TDD
- Utilizar aserciones efectivamente en las pruebas
- Organizar y estructurar las pruebas de manera clara
- Escribir pruebas mantenibles y legibles
- Aplicar patrones de diseño de pruebas

El diseño de pruebas es fundamental para mantener suites de pruebas efectivas y mantenibles.

Agenda de la Unidad

1. **Características de una buena prueba en TDD**
    - F.I.R.S.T. principles
    - Readability y mantenibilidad
    - Independencia y aislación
2. **Utilización de aserciones**
    - Tipos de aserciones
    - Uso efectivo de aserciones
    - Mensajes de error claros
3. **Organización y estructura de las pruebas**
    - Estructura de archivos de pruebas
    - Nomenclatura y organización
    - Patrón Arrange-Act-Assert

Esta agenda cubre todos los aspectos del diseño de pruebas en TDD.

Características de una Buena Prueba en TDD

Principios F.I.R.S.T.

- **F**ast (Rápida): Las pruebas deben ejecutarse rápidamente
- **I**ndependent (Independiente): Cada prueba debe poder ejecutarse sola
- **R**epeatable (Repetible): Debe dar el mismo resultado siempre
- **S**elf-validating (Auto-validante): Debe pasar o fallar claramente
- **T**imely (Oportuna): Debe escribirse en el momento adecuado (antes del código)

Características Adicionales:

- **Legible:** Fácil de entender qué verifica
- **Mantenible:** Fácil de modificar cuando cambia la funcionalidad
- **Específica:** Verifica un solo comportamiento
- **Confiable:** No falla intermitentemente

Los principios FIRST son fundamentales para escribir buenas pruebas.

F.I.R.S.T. en Detalle

Fast (Rápida)

Las pruebas deben ejecutarse en milisegundos. Si una prueba tarda segundos, algo está mal.

Ejemplo: Una prueba unitaria simple debe ejecutarse en menos de 10ms.

Independent (Independiente)

Cada prueba debe poder ejecutarse sola, sin depender de otras pruebas.

Problema común: Prueba 2 depende del estado dejado por Prueba 1.

Repeatable (Repetible)

Debe dar el mismo resultado cada vez que se ejecuta, sin importar el orden o el entorno.

Self-validating (Auto-validante)

El resultado debe ser binario: pasa o falla. No requiere interpretación humana.

Mal: "Verificar manualmente que el resultado es correcto"

Timely (Oportuna)

Debe escribirse justo antes del código que valida (en la fase RED).

No escribir: Pruebas después de implementar el código.

Cada principio FIRST tiene un propósito específico y debe seguirse.

Ejemplo: Prueba Mal Diseñada

❌ Prueba con problemas:

```
describe('Usuario', () => { let usuario; beforeEach(() => { usuario = new Usuario(); usuario.nombre = 'Juan'; usuario.apellido = 'Pérez'; usuario.email = 'juan@email.com'; usuario.edad = 25; usuario.guardar(); // Guarda en base de datos }); test('test1', () => { // Modifica el usuario global usuario.edad = 30; usuario.actualizar(); expect(usuario.edad).toBe(30); }); test('test2', () => { // Depende de test1 - si test1 falla, este también falla expect(usuario.edad).toBe(30); // Espera el valor de test1 }); });
```

Problemas:

- ❌ Nombres no descriptivos (test1, test2)
- ❌ Dependencia entre pruebas (test2 depende de test1)
- ❌ Lenta (accede a base de datos)
- ❌ Estado compartido entre pruebas

Esta prueba tiene múltiples problemas. Veamos cómo mejorarla.

Ejemplo: Prueba Bien Diseñada

✅ Prueba mejorada:

```
describe('Usuario', () => { test('debe actualizar la edad cuando se proporciona un nuevo valor', () => { // Arrange: Preparar datos de prueba aislados const usuario = new Usuario(); usuario.nombre = 'Juan'; usuario.edad = 25; // Act: Ejecutar la operación usuario.actualizarEdad(30); // Assert: Verificar resultado expect(usuario.edad).toBe(30); }); test('debe mantener la edad original si se proporciona un valor inválido', () => { // Arrange: Cada prueba tiene su propio estado const usuario = new Usuario(); usuario.edad = 25; // Act usuario.actualizarEdad(-5); // Assert: Verificación independiente expect(usuario.edad).toBe(25); }); });
```

Mejoras:

- ✅ Nombres descriptivos
- ✅ Pruebas independientes
- ✅ Sin dependencias externas (base de datos)
- ✅ Estado aislado para cada prueba
- ✅ Estructura AAA clara

Noten la diferencia. Esta prueba sigue todos los principios FIRST.

Utilización de Aserciones

¿Qué es una Aserción?

Una **aserción** es una declaración que verifica que algo es verdadero. Si la aserción falla, la prueba falla.

Tipos de Aserciones Comunes:

- **Igualdad:** `expect(valor).toBe(esperado)`
- **Desigualdad:** `expect(valor).not.toBe(esperado)`
- **Verdadero/Falso:** `expect(valor).toBeTruthy()`
- **Contiene:** `expect(array).toContain(elemento)`
- **Excepciones:** `expect(() => funcion()).toThrow()`
- **Mayor/Menor:** `expect(valor).toBeGreaterThan(limite)`

Las aserciones son la herramienta fundamental para verificar comportamiento.

Mensajes de Error en Aserciones

❌ Aserción sin mensaje:

```
test('prueba descuento', () => { const precio = calcularDescuento(100, 10); expect(precio).toBe(90); }); // Si falla: "Expected 90 but got 95" // ¿Qué descuento? ¿Qué precio?
```

✅ Aserción con mensaje:

```
test('debe aplicar 10% de descuento a precio de $100', () => { const precio = calcularDescuento(100, 10); expect(precio).toBe(90); }); // Si falla: "Expected 90 but got 95" // El nombre del test ya explica qué se prueba
```

> **Recomendación:** El nombre del test debe ser suficientemente descriptivo para entender qué falló sin necesidad de agregar mensajes adicionales en las aserciones.

Un buen nombre de prueba es mejor que un mensaje de error detallado.

Ejemplos de Aserciones Comunes

```
// Igualdad exacta expect(5).toBe(5); expect('texto').toBe('texto'); // Igualdad aproximada (para números decimales) expect(0.1 + 0.2).toBeCloseTo(0.3, 5); // Verdadero/Falso expect(true).toBeTruthy(); expect(false).toBeFalsy(); expect(usuario.activo).toBe(true); // Arrays y objetos expect(['a', 'b', 'c']).toContain('b'); expect(objeto).toHaveProperty('nombre'); expect(objeto).toEqual({nombre: 'Juan', edad: 25}); // Excepciones expect(() => dividir(10, 0)).toThrow(); expect(() => dividir(10, 0)).toThrow('División por cero'); // Comparaciones numéricas expect(10).toBeGreaterThan(5); expect(5).toBeLessThan(10); expect(10).toBeGreaterThanOrEqual(10); // Strings expect('Hola Mundo').toContain('Mundo'); expect('email@dominio.com').toMatch(/^[\w-]+@[\w-]+\.\w+$/); // Null/Undefined expect(null).toBeNull(); expect(undefined).toBeUndefined(); // Negación expect(5).not.toBe(10); expect(usuario).not.toBeNull();
```

Estos son ejemplos comunes de aserciones. Elijan la más apropiada para cada caso.

Patrón Arrange-Act-Assert (AAA)

El patrón AAA estructura las pruebas en tres fases:

1. Arrange (Preparar)

Configurar todos los datos y objetos necesarios para la prueba.

```
// Arrange const calculadora = new Calculadora(); const numero1 = 10; const numero2 = 5;
```

2. Act (Actuar)

Ejecutar la operación que se está probando.

```
// Act const resultado = calculadora.sumar(numero1, numero2);
```

3. Assert (Verificar)

Verificar que el resultado es el esperado.

```
// Assert expect(resultado).toBe(15);
```

El patrón AAA hace que las pruebas sean más legibles y mantenibles.

Ejemplo Completo: Patrón AAA

```
describe('CalculadoraDescuentos', () => { test('debe aplicar 15% de descuento cuando el monto es mayor a $500', () => { // Arrange: Preparar todos los datos necesarios const calculadora = new CalculadoraDescuentos(); const montoSinDescuento = 600; const descuentoEsperado = 90; // 15% de 600 // Act: Ejecutar la operación que se está probando const resultado = calculadora.calcularDescuento(montoSinDescuento); // Assert: Verificar que el resultado es correcto expect(resultado).toBe(descuentoEsperado); }); test('debe lanzar error cuando el monto es negativo', () => { // Arrange const calculadora = new CalculadoraDescuentos(); const montoInvalido = -100; // Act & Assert: En este caso, Act y Assert están juntos expect(() => { calculadora.calcularDescuento(montoInvalido); }).toThrow('El monto no puede ser negativo'); }); });
```

Ventajas del patrón AAA:

- Hace explícito qué se está probando
- Facilita identificar dónde falla la prueba
- Mejora la legibilidad
- Facilita el mantenimiento

Siempre sigan el patrón AAA. Hace que las pruebas sean mucho más claras.

Organización y Estructura de Pruebas

Estructura de Archivos:

- **Un archivo de prueba por archivo de código:**
    - `calculadora.js` → `calculadora.test.js`
    - `usuario.js` → `usuario.test.js`
- **Carpetas paralelas:**
    - `src/calculadora.js`
    - `tests/calculadora.test.js`
- **Misma estructura:**
    - `src/models/usuario.js`
    - `tests/models/usuario.test.js`

Agrupación con describe():

Usa `describe()` para agrupar pruebas relacionadas y crear una estructura jerárquica clara.

La organización es clave para mantener suites de pruebas grandes.

Ejemplo de Organización con describe()

```
// Estructura jerárquica clara describe('Calculadora', () => { describe('operaciones básicas', () => { test('debe sumar dos números', () => { const calc = new Calculadora(); expect(calc.sumar(2, 3)).toBe(5); }); test('debe restar dos números', () => { const calc = new Calculadora(); expect(calc.restar(5, 3)).toBe(2); }); }); describe('validaciones', () => { test('debe lanzar error al dividir por cero', () => { const calc = new Calculadora(); expect(() => calc.dividir(10, 0)).toThrow(); }); test('debe validar que los operandos sean números', () => { const calc = new Calculadora(); expect(() => calc.sumar('a', 5)).toThrow(); }); }); describe('operaciones avanzadas', () => { test('debe calcular potencia', () => { const calc = new Calculadora(); expect(calc.potencia(2, 3)).toBe(8); }); }); });
```

Resultado en la consola:

```
Calculadora
operaciones básicas
✓ debe sumar dos números
✓ debe restar dos números
validaciones
✓ debe lanzar error al dividir por cero
```

Esta organización hace que sea fácil encontrar pruebas específicas.

Nomenclatura de Pruebas

❌ Malos Nombres:

- `test1()`
- `testCalculadora()`
- `prueba()`
- `test_suma()`

✅ Buenos Nombres:

- `debe sumar dos números positivos`
- `debe lanzar error cuando el divisor es cero`
- `debe aplicar descuento del 10% para montos mayores a $100`
- `no debe aceptar emails inválidos`

Convenciones de Nomenclatura:

- **Formato:** "debe [acción] [condición] [resultado esperado]"
- **Ejemplo:** "debe calcular el área de un círculo dado su radio"
- **Lenguaje:** Español claro y descriptivo
- **Longitud:** No tener miedo a nombres largos si son descriptivos

Un buen nombre de prueba dice exactamente qué se está probando.

Errores Comunes en el Diseño de Pruebas

1. Pruebas que verifican múltiples cosas

❌ **Error:**

```
test('debe crear usuario y validar email y calcular edad', () => { const usuario = new Usuario('Juan', 'juan@email.com', '1990-01-01'); expect(usuario.nombre).toBe('Juan'); expect(usuario.email).toBe('juan@email.com'); expect(usuario.edad).toBe(34); });
```

✅ **Correcto:** Una prueba por comportamiento

2. Pruebas dependientes

❌ **Error:** Prueba 2 depende del estado de Prueba 1.

✅ **Correcto:** Cada prueba es independiente y puede ejecutarse sola.

3. Pruebas lentas

❌ **Error:** Acceder a base de datos o servicios externos en pruebas unitarias.

✅ **Correcto:** Usar mocks o stubs para dependencias externas.

4. Nombres genéricos

❌ **Error:** Nombres como "test1", "prueba", "ejemplo".

✅ **Correcto:** Nombres descriptivos que explican qué se prueba.

Estos errores son muy comunes. Estén atentos para evitarlos.

Buenas Prácticas en el Diseño de Pruebas

1. Una aserción por prueba (idealmente)

Facilita identificar qué falló cuando la prueba no pasa.

2. Nombres descriptivos

El nombre debe explicar qué comportamiento se está probando.

3. Pruebas aisladas

Cada prueba debe poder ejecutarse independientemente, en cualquier orden.

4. Datos de prueba simples

Usa los datos más simples posibles que demuestren el comportamiento.

5. Pruebas rápidas

Las pruebas unitarias deben ejecutarse en milisegundos, no segundos.

6. Sin efectos secundarios

Una prueba no debe modificar el estado que afecte a otras pruebas.

7. Estructura AAA clara

Siempre separa Arrange, Act y Assert con comentarios si es necesario.

Estas prácticas hacen que las pruebas sean mantenibles a largo plazo.

Práctica Guiada #1: Rediseñar Pruebas

Objetivo:

Mejorar el diseño de pruebas existentes aplicando los principios FIRST y el patrón AAA.

Prueba original (con problemas):

```
describe('Usuario', () => { let usuario; beforeEach(() => { usuario = new Usuario(); }); test('test1', () => { usuario.nombre = 'Juan'; usuario.guardar(); expect(usuario.nombre).toBe('Juan'); }); test('test2', () => { expect(usuario.nombre).toBe('Juan'); }); });
```

Tareas:

1. Renombrar las pruebas con nombres descriptivos
2. Eliminar dependencias entre pruebas
3. Aplicar patrón AAA
4. Hacer cada prueba independiente

Vamos paso a paso mejorando esta prueba.

Práctica Guiada #1: Solución

Prueba mejorada:

```
describe('Usuario', () => { test('debe establecer el nombre cuando se proporciona un valor válido', () => { // Arrange const usuario = new Usuario(); const nombreEsperado = 'Juan'; // Act usuario.nombre = nombreEsperado; // Assert expect(usuario.nombre).toBe(nombreEsperado); }); test('debe mantener el nombre establecido independientemente de otras pruebas', () => { // Arrange: Cada prueba crea su propio usuario const usuario = new Usuario(); const nombreEsperado = 'María'; // Act usuario.nombre = nombreEsperado; // Assert expect(usuario.nombre).toBe(nombreEsperado); }); });
```

Mejoras aplicadas:

- ✅ Nombres descriptivos que explican qué se prueba
- ✅ Cada prueba es independiente
- ✅ Patrón AAA aplicado claramente
- ✅ Sin dependencias entre pruebas
- ✅ Comentarios AAA para claridad

Noten cómo cada prueba ahora es clara e independiente.

Práctica Guiada #2: Diseñar Pruebas desde Cero

Objetivo:

Diseñar pruebas bien estructuradas para una clase `ValidadorTarjeta`.

Requisitos:

- Validar que el número de tarjeta tenga 16 dígitos
- Validar que todos los caracteres sean numéricos
- Validar que el número no esté vacío
- Lanzar errores descriptivos cuando la validación falla

Tarea:

Diseñar pruebas que sigan todos los principios FIRST y el patrón AAA. Escribir nombres descriptivos y estructurar las pruebas adecuadamente.

Usen todos los principios aprendidos para diseñar estas pruebas.

Práctica Guiada #2: Solución

Pruebas bien diseñadas:

```
describe('ValidadorTarjeta', () => { describe('validación de longitud', () => { test('debe aceptar un número de tarjeta con exactamente 16 dígitos', () => { // Arrange const validador = new ValidadorTarjeta(); const numeroValido = '1234567890123456'; // Act const resultado = validador.esValido(numeroValido); // Assert expect(resultado).toBe(true); }); test('debe rechazar un número de tarjeta con menos de 16 dígitos', () => { // Arrange const validador = new ValidadorTarjeta(); const numeroInvalido = '123456789012345'; // Act & Assert expect(() => validador.esValido(numeroInvalido)) .toThrow('El número de tarjeta debe tener 16 dígitos'); }); }); describe('validación de formato', () => { test('debe rechazar números que contengan letras', () => { // Arrange const validador = new ValidadorTarjeta(); const numeroConLetras = '123456789012345a'; // Act & Assert expect(() => validador.esValido(numeroConLetras)) .toThrow('El número de tarjeta solo puede contener dígitos'); }); test('debe rechazar números vacíos', () => { // Arrange const validador = new ValidadorTarjeta(); const numeroVacio = ''; // Act & Assert expect(() => validador.esValido(numeroVacio)) .toThrow('El número de tarjeta no puede estar vacío'); }); }); });
```

Características del diseño:

- ✅ Agrupación lógica con describe()
- ✅ Nombres descriptivos y claros
- ✅ Patrón AAA aplicado
- ✅ Pruebas independientes
- ✅ Una cosa por prueba

Este es un ejemplo excelente de diseño de pruebas bien estructurado.

Práctica Evaluada

Enunciado:

Diseña y escribe pruebas bien estructuradas para una clase `GestorPedidos` con las siguientes funcionalidades:

1. Agregar producto a un pedido
2. Eliminar producto de un pedido
3. Calcular el total del pedido
4. Aplicar descuento por código
5. Validar que no se puedan agregar productos con cantidad negativa
6. Validar que no se pueda eliminar un producto que no existe

Criterios de Evaluación:

- **Diseño de pruebas (40%):** Principios FIRST aplicados, patrón AAA, pruebas independientes
- **Nomenclatura (25%):** Nombres descriptivos, formato consistente
- **Organización (20%):** Agrupación lógica con describe(), estructura clara
- **Aserciones (15%):** Uso apropiado de aserciones, mensajes claros

Rúbrica:

- **Excelente (90-100):** Todos los principios aplicados perfectamente, diseño impecable
- **Bueno (75-89):** Principios aplicados correctamente, diseño claro y mantenible
- **Satisfactorio (60-74):** Algunos principios aplicados, diseño funcional
- **Necesita mejorar (<60):** Principios no aplicados, diseño deficiente

Esta práctica evalúa el dominio completo del diseño de pruebas.

Mini-Quiz

1. ¿Qué significa F.I.R.S.T. en el contexto de pruebas?

a) Fast, Independent, Repeatable, Self-validating, Timely
b) Fast, Integrated, Reliable, Simple, Testable
c) Functional, Independent, Repeatable, Simple, Timely
d) Flexible, Independent, Reliable, Simple, Testable

2. ¿Qué patrón se recomienda para estructurar pruebas?

a) AAA (Arrange-Act-Assert)
b) BDD (Behavior-Driven Development)
c) DDD (Domain-Driven Design)
d) SOLID

3. ¿Cuál es una característica de una buena prueba?

a) Debe ejecutarse en segundos
b) Debe ser independiente y ejecutarse sola
c) Debe depender de otras pruebas
d) Debe requerir interpretación humana

Mini-Quiz (Continuación)

4. ¿Cuál es un buen nombre para una prueba?

a) test1
b) debe calcular el descuento del 10% cuando el monto es mayor a $100
c) pruebaCalculadora
d) test

5. ¿Qué significa "Self-validating" en FIRST?

a) La prueba se valida automáticamente
b) Debe tener un resultado binario claro (pasa o falla)
c) Debe validarse manualmente
d) Debe validar múltiples cosas

6. Verdadero o Falso: Una prueba puede verificar múltiples comportamientos.

Verdadero
Falso - Idealmente una prueba verifica un comportamiento

Mini-Quiz (Continuación)

7. ¿Para qué se usa describe() en las pruebas?

a) Para describir la prueba
b) Para agrupar pruebas relacionadas
c) Para comentar código
d) Para documentar

8. ¿Cuál es el tiempo ideal de ejecución de una prueba unitaria?

a) Segundos
b) Milisegundos
c) Minutos
d) No importa

9. Verdadero o Falso: Las pruebas deben poder ejecutarse en cualquier orden.

Verdadero
Falso

10. ¿Cuál es un error común en el diseño de pruebas?

a) Usar nombres descriptivos
b) Hacer pruebas dependientes entre sí
c) Aplicar patrón AAA
d) Hacer pruebas independientes

Respuestas del Mini-Quiz

1. **a) Fast, Independent, Repeatable, Self-validating, Timely** - Los principios FIRST
2. **a) AAA (Arrange-Act-Assert)** - Patrón recomendado para estructurar pruebas
3. **b) Debe ser independiente y ejecutarse sola** - Independencia es clave
4. **b) debe calcular el descuento del 10% cuando el monto es mayor a $100** - Nombre descriptivo
5. **b) Debe tener un resultado binario claro (pasa o falla)** - Self-validating
6. **Falso** - Idealmente una prueba verifica un comportamiento
7. **b) Para agrupar pruebas relacionadas** - Organización lógica
8. **b) Milisegundos** - Las pruebas deben ser rápidas
9. **Verdadero** - Las pruebas deben ser independientes
10. **b) Hacer pruebas dependientes entre sí** - Error común

Si tuvieron dudas, revisen esos conceptos antes de continuar.

Resumen de la Unidad

Puntos Clave:

- Las pruebas deben seguir los principios **F.I.R.S.T.**
- El patrón **Arrange-Act-Assert (AAA)** estructura las pruebas claramente
- Las pruebas deben ser **independientes** y poder ejecutarse en cualquier orden
- Los **nombres descriptivos** son fundamentales para entender qué se prueba
- Las pruebas deben ser **rápidas** (milisegundos, no segundos)
- La **organización** con describe() mejora la estructura y legibilidad
- Las **aserciones** deben ser apropiadas y claras
- Una prueba debe verificar **un comportamiento** específico

Este resumen cubre los conceptos fundamentales del diseño de pruebas.

Tarea para Casa

Actividad:

Rediseña las pruebas de un proyecto existente (o crea pruebas desde cero) aplicando todos los principios de diseño aprendidos:

1. Elige una clase o módulo de código
2. Escribe pruebas siguiendo principios FIRST
3. Aplica patrón AAA en todas las pruebas
4. Organiza las pruebas con describe() apropiadamente
5. Usa nombres descriptivos para todas las pruebas
6. Asegúrate de que las pruebas sean independientes
7. Verifica que las pruebas sean rápidas

Entregables:

- Código de pruebas completo (archivos .test.js)
- Documento explicando:
    - Cómo se aplicaron los principios FIRST
    - Cómo se estructuró el código con AAA
    - Decisiones de organización tomadas
    - Mejoras realizadas respecto a pruebas anteriores
- Capturas de pantalla mostrando la ejecución de las pruebas

Fecha de entrega:

Una semana después de esta clase

Esta tarea les permitirá practicar el diseño de pruebas en un proyecto real.

Recursos Adicionales

📚 Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Videos sobre diseño de pruebas
- Patrones de testing
- Ejemplos de pruebas unitarias
- Herramientas de testing

Ver Recursos Adicionales →

¡Gracias!

Unidad III: Diseño de pruebas

Próxima clase: Unidad IV - Cobertura de pruebas

Preguntas y dudas

Practiquen el diseño de pruebas hasta que sea natural.

Recursos Adicionales - Unidad 03

Diseño de Pruebas - Enlaces, videos y material complementario

📚 Documentación

[Errores Comunes al Escribir Pruebas Artículo]

Artículo sobre errores comunes al diseñar pruebas y cómo evitarlos.

[Testing JavaScript Curso]

Curso completo sobre diseño y escritura de pruebas efectivas.

🎥 Videos Educativos

[Cómo Diseñar Pruebas Efectivas Video]

Guía sobre estrategias para diseñar pruebas que sean útiles y mantenibles.

💡 Consejo de Estudio

> Para diseñar buenas pruebas: 1) Piensa en los casos límite y edge cases, 2) Una prueba debe verificar un comportamiento específico, 3) Nombra tus pruebas de forma descriptiva, 4) Organiza tus pruebas siguiendo el patrón Arrange-Act-Assert. El diseño de pruebas es un arte que mejora con la práctica.

Desarrollo Basado en Pruebas (TDD)

Unidad IV: Cobertura de pruebas

Bienvenidos a la cuarta unidad. Hoy aprenderemos sobre la importancia de la cobertura de pruebas en TDD.

Competencias y Resultados de Aprendizaje

Competencia Específica CE13

Especificar y ejecutar pruebas de calidad de los prototipos (diseño), simulaciones y los componentes de software desarrollados para garantizar su fidelidad y precisión en todos los niveles (Test units) y garantizando una asertiva aceptación de los usuarios (UCT).

Resultados de Aprendizaje

- Comprender la importancia de la cobertura de pruebas en TDD
- Medir la cobertura de pruebas usando herramientas
- Aplicar estrategias para lograr cobertura apropiada
- Interpretar métricas de cobertura
- Equilibrar cobertura con mantenibilidad

La cobertura de pruebas es fundamental para garantizar la calidad del código.

Agenda de la Unidad

1. **Importancia de la cobertura de pruebas en TDD**
    - ¿Qué es la cobertura de pruebas?
    - Por qué es importante
    - Beneficios de una buena cobertura
2. **Métodos para medir la cobertura de pruebas**
    - Tipos de cobertura (líneas, ramas, funciones)
    - Herramientas de medición
    - Interpretación de reportes
3. **Estrategias para una apropiada cobertura de pruebas**
    - Niveles de cobertura recomendados
    - Equilibrio entre cobertura y mantenibilidad
    - Áreas críticas vs áreas menos críticas

Esta agenda cubre todos los aspectos de la cobertura de pruebas en TDD.

¿Qué es la Cobertura de Pruebas?

Definición:

La **cobertura de pruebas** es una métrica que indica qué porcentaje del código fuente ha sido ejecutado durante la ejecución de las pruebas automatizadas.

Objetivo:

Identificar qué partes del código están siendo probadas y cuáles no, ayudando a detectar áreas sin pruebas adecuadas.

Tipos de Cobertura:

- **Cobertura de líneas:** Porcentaje de líneas ejecutadas
- **Cobertura de ramas:** Porcentaje de ramas condicionales probadas
- **Cobertura de funciones:** Porcentaje de funciones llamadas
- **Cobertura de sentencias:** Porcentaje de sentencias ejecutadas

La cobertura de pruebas nos ayuda a identificar qué código está siendo probado.

Importancia de la Cobertura de Pruebas en TDD

Beneficios:

- **Confianza:** Mayor confianza en que el código funciona correctamente
- **Detección temprana:** Encuentra bugs antes de producción
- **Documentación:** Las pruebas documentan el comportamiento esperado
- **Refactorización segura:** Facilita cambiar código sin temor
- **Calidad:** Mejora la calidad general del código

Riesgos de baja cobertura:

- **Bugs ocultos:** Errores no detectados hasta producción
- **Regresiones:** Cambios rompen funcionalidad existente
- **Incertidumbre:** No se sabe qué código realmente funciona
- **Mantenimiento:** Difícil mantener código sin pruebas
- **Costos:** Bugs en producción son más costosos

La cobertura adecuada es fundamental para proyectos de calidad.

Tipos de Cobertura de Pruebas

1. Cobertura de Líneas (Line Coverage)

Mide el porcentaje de líneas de código ejecutadas al menos una vez durante las pruebas.

```
function calcularDescuento(monto) { if (monto > 100) { // ← Línea 1 (ejecutada si monto > 100) return monto * 0.1; // ← Línea 2 (ejecutada si monto > 100) } return 0; // ← Línea 3 (ejecutada si monto <= 100) } // Si solo probamos con monto = 150: // Cobertura de líneas: 2/3 = 66.7% (líneas 1 y 2)
```

2. Cobertura de Ramas (Branch Coverage)

Mide el porcentaje de ramas condicionales (if/else, switch) probadas.

En el ejemplo anterior: Para 100% de cobertura de ramas, necesitamos probar tanto el caso `monto > 100` como `monto <= 100`.

La cobertura de ramas es más estricta que la cobertura de líneas.

Más Tipos de Cobertura

3. Cobertura de Funciones (Function Coverage)

Mide el porcentaje de funciones llamadas al menos una vez durante las pruebas.

```
function procesarUsuario(usuario) { validarUsuario(usuario); // ← Función 1 guardarUsuario(usuario); // ← Función 2 enviarNotificacion(usuario); // ← Función 3 } // Si solo probamos validarUsuario: // Cobertura de funciones: 1/3 = 33.3%
```

4. Cobertura de Sentencias (Statement Coverage)

Similar a cobertura de líneas, pero mide sentencias ejecutadas (una línea puede tener múltiples sentencias).

5. Cobertura de Condiciones (Condition Coverage)

Evalúa todas las combinaciones posibles de condiciones booleanas.

```
if (a > 0 && b < 10) { // Para 100% de cobertura de condiciones: // - a > 0 AND b < 10 (true, true) // - a > 0 AND b >= 10 (true, false) // - a <= 0 AND b < 10 (false, true) // - a <= 0 AND b >= 10 (false, false) }
```

Cada tipo de cobertura tiene su propósito y nivel de detalle.

En la Vida Real: Cobertura de Pruebas

Escenario Real:

Una empresa de e-commerce tiene un sistema de cálculo de descuentos para clientes VIP. El código tiene 1000 líneas relacionadas con descuentos.

Sin medir cobertura:

- Se escriben pruebas "a ojo"
- Se asume que todo está probado
- Un bug en producción afecta a 500 clientes
- El costo de reparación: $10,000

Con cobertura medida:

- Se identifica que solo 60% está probado
- Se escriben pruebas para el 40% faltante
- Se detectan 3 bugs antes de producción
- El costo de prevención: $500

> **Conclusión:** Medir la cobertura ayuda a identificar áreas críticas sin pruebas y prevenir bugs costosos en producción.

Este ejemplo muestra el valor real de medir la cobertura.

Herramientas para Medir Cobertura

Herramientas Populares:

- **Istanbul (JavaScript/Node.js):**

  `npm install --save-dev nyc nyc npm test // Genera reporte HTML con cobertura`

- **Jest (JavaScript/React):** Incluye cobertura integrada

  `npm test -- --coverage // Genera reporte automáticamente`

- **Coverlet (.NET):** Para proyectos .NET

  `dotnet test /p:CollectCoverage=true // Genera reporte de cobertura`

- **JaCoCo (Java):** Para proyectos Java
- **Coverage.py (Python):** Para proyectos Python

Hay herramientas para cada lenguaje y tecnología.

Ejemplo de Reporte de Cobertura

Reporte generado por Jest:

```
File | % Stmts | % Branch | % Funcs | % Lines -------------------|---------|----------|---------|-------- calculadora.js | 85.71 | 75 | 100 | 85.71 Calcular | 85.71 | 75 | 100 | 85.71 sumar | 100 | 100 | 100 | 100 restar | 100 | 100 | 100 | 100 multiplicar | 100 | 100 | 100 | 100 dividir | 50 | 0 | 100 | 50 | dividir por cero| 0 | 0 | | 0 usuario.js | 66.67 | 50 | 66.67| 66.67 -------------------|---------|----------|---------|-------- TOTAL | 76.19 | 62.5 | 83.33 | 76.19
```

Análisis:

- ✅ `sumar`, `restar`, `multiplicar` tienen 100% de cobertura
- ⚠️ `dividir` solo tiene 50% - falta probar división por cero
- ⚠️ `usuario.js` tiene baja cobertura (66.67%)

Los reportes muestran exactamente dónde necesitamos más pruebas.

Niveles de Cobertura Recomendados

Guías Generales:

80% - 90% Cobertura (Recomendado)

Excelente: Balance entre calidad y esfuerzo

- Proyectos comerciales estándar
- Mayoría de código crítico probado
- Buen balance costo-beneficio

70% - 80% Cobertura (Aceptable)

Bueno: Cobertura adecuada para proyectos normales

- Proyectos internos no críticos
- Proyectos en desarrollo temprano
- Áreas críticas deben tener 90%+

< 70% Cobertura (Inadecuado)

Insuficiente: Necesita mejorar

- Muchas áreas sin probar
- Alto riesgo de bugs
- Se requiere aumentar cobertura

100% Cobertura (Ideal pero no siempre práctico)

Perfecto: Todo el código probado, pero puede ser costoso

- Sistemas críticos (médicos, financieros)
- Alto costo de mantenimiento

- Puede no ser necesario para todo el código

80-90% es un buen objetivo para la mayoría de proyectos.

Estrategias para Mejorar la Cobertura

1. Priorizar Áreas Críticas

Enfocarse primero en código crítico para el negocio:

- Cálculos financieros
- Validaciones de seguridad
- Procesamiento de pagos
- Lógica de negocio compleja

2. Identificar Código sin Cobertura

Usar herramientas para identificar líneas específicas sin cobertura:

```
// Ejemplo: Reporte muestra línea 45 sin cobertura function procesarPago(monto) { if (monto > 0) { // Línea 45: No probada - caso monto <= 0 return calcularDescuento(monto); } return 0; } // Solución: Agregar prueba para monto <= 0 test('debe retornar 0 cuando monto es negativo', () => { expect(procesarPago(-10)).toBe(0); });
```

3. Cubrir Casos Edge

Probar casos límite y excepcionales:

- Valores nulos o undefined
- Valores en los límites
- Arrays vacíos
- Strings vacíos

Las estrategias ayudan a mejorar la cobertura de manera eficiente.

Equilibrio: Cobertura vs Mantenibilidad

✅ Buen Equilibrio:

- 80-90% cobertura general
- 100% en código crítico
- Pruebas mantenibles y claras
- No probar código trivial
- Enfoque en comportamiento, no implementación

❌ Mal Equilibrio:

- Buscar 100% a toda costa
- Pruebas complejas y difíciles de mantener
- Probar código trivial innecesariamente
- Ignorar código crítico
- Pruebas que duplican lógica

Regla de Oro:

"No todo código necesita el mismo nivel de cobertura. Prioriza calidad sobre cantidad."

- ✅ **Código crítico:** Alta cobertura (90-100%)
- ✅ **Código estándar:** Cobertura media (70-90%)
- ✅ **Código trivial:** Cobertura baja aceptable (50-70%)
- ✅ **Código legacy:** Mejorar gradualmente

El equilibrio es clave para proyectos sostenibles.

Errores Comunes en Cobertura de Pruebas

1. Obsesión con el 100%

❌ **Error:** Intentar alcanzar 100% de cobertura sin importar el costo.

✅ **Correcto:** Buscar 80-90% con enfoque en código crítico.

2. Ignorar la Cobertura

❌ **Error:** No medir la cobertura porque "sabemos que funciona".

✅ **Correcto:** Medir constantemente y usar métricas para guiar las pruebas.

3. Cobertura Falsa

❌ **Error:** Escribir pruebas que ejecutan código pero no verifican comportamiento.

```
// ❌ Prueba que ejecuta código pero no verifica nada test('ejecutar calcular', () => { calcularDescuento(100); // Solo ejecuta, no verifica resultado }); // ✅ Prueba que verifica comportamiento test('debe calcular 10% de descuento para $100', () => { expect(calcularDescuento(100)).toBe(10); });
```

4. Ignorar Áreas Críticas

❌ **Error:** Tener alta cobertura en código simple pero baja en código crítico.

✅ **Correcto:** Priorizar código crítico incluso si baja la cobertura general.

Estos errores son muy comunes. Estén atentos para evitarlos.

Buenas Prácticas en Cobertura de Pruebas

1. Medir Regularmente

Ejecutar reportes de cobertura en cada commit o al menos diariamente.

2. Establecer Umbrales

Definir mínimos aceptables y hacer que el build falle si no se alcanzan:

```
// jest.config.js module.exports = { coverageThresholds: { global: { branches: 80, functions: 80, lines: 80, statements: 80 }, './src/critical/': { branches: 95, functions: 95, lines: 95, statements: 95 } } };
```

3. Revisar Reportes

No solo mirar el porcentaje, sino identificar qué código falta probar.

4. Integrar en CI/CD

Hacer que la cobertura sea parte del proceso de integración continua.

5. Priorizar Calidad sobre Cantidad

Mejor tener 80% de cobertura con buenas pruebas que 100% con pruebas malas.

6. Revisar Código sin Cobertura

Si hay código sin cobertura, preguntarse: ¿Es necesario? ¿Debe eliminarse?

Estas prácticas hacen que la cobertura sea útil y sostenible.

Práctica Guiada #1: Mejorar Cobertura

Objetivo:

Identificar y mejorar la cobertura de pruebas en un módulo existente.

Código a probar:

```
// calculadora.js function calcularDescuento(monto, esVIP) { if (monto > 100) { if (esVIP) { return monto * 0.15; } return monto * 0.10; } return 0; } // Pruebas actuales (cobertura: 33%) test('debe calcular descuento VIP', () => { expect(calcularDescuento(150, true)).toBe(22.5); });
```

Tareas:

1. Ejecutar reporte de cobertura
2. Identificar líneas sin cobertura
3. Agregar pruebas para cubrir todos los casos
4. Verificar que la cobertura aumente a 100%

Vamos paso a paso mejorando la cobertura.

Práctica Guiada #1: Solución

Pruebas completas para 100% cobertura:

```
describe('calcularDescuento', () => { test('debe calcular descuento VIP del 15% para montos mayores a $100', () => { expect(calcularDescuento(150, true)).toBe(22.5); }); test('debe calcular descuento estándar del 10% para montos mayores a $100', () => { expect(calcularDescuento(150, false)).toBe(15); }); test('debe retornar 0 para montos menores o iguales a $100 (VIP)', () => { expect(calcularDescuento(100, true)).toBe(0); }); test('debe retornar 0 para montos menores o iguales a $100 (estándar)', () => { expect(calcularDescuento(50, false)).toBe(0); }); });
```

Análisis:

- ✅ Cobertura de líneas: 100% (todas las líneas ejecutadas)
- ✅ Cobertura de ramas: 100% (todas las ramas if/else probadas)
- ✅ Cobertura de funciones: 100% (función probada)
- ✅ Casos edge cubiertos (monto = 100, monto < 100)

Ahora tenemos 100% de cobertura con pruebas bien diseñadas.

Práctica Guiada #2: Interpretar Reporte

Objetivo:

Interpretar un reporte de cobertura y crear un plan de acción.

Reporte de cobertura:

```
 File | % Stmts | % Branch | % Funcs | % Lines -------------------|---------|----------|---------|-------- auth.js | 45.45 | 25 | 66.67 | 45.45 validarUsuario | 100 | 100 | 100 | 100 hashPassword | 100 | 100 | 100 | 100 verificarToken | 0 | 0 | 0 | 0 payment.js | 83.33 | 75 | 100 | 83.33 procesarPago | 100 | 100 | 100 | 100 calcularImpuesto | 50 | 0 | 100 | 50 -------------------|---------|----------|---------|-------- TOTAL | 64.29 | 50 | 83.33 | 64.29
```

Tareas:

1. Identificar módulos con baja cobertura
2. Priorizar qué mejorar primero
3. Crear plan de acción con pruebas específicas
4. Establecer objetivo de cobertura

Practiquen interpretar reportes y crear planes de acción.

Práctica Guiada #2: Solución

Análisis del Reporte:

🔴 Crítico - auth.js (45.45% cobertura)

- **verificarToken:** 0% - ¡Ninguna prueba! (CRÍTICO para seguridad)
- **Prioridad:** ALTA - Código de seguridad debe tener 100%
- **Acción:** Escribir pruebas para verificarToken inmediatamente

🟡 Medio - payment.js (83.33% cobertura)

- **calcularImpuesto:** 50% - Falta probar alguna rama
- **Prioridad:** MEDIA - Código financiero importante
- **Acción:** Identificar rama faltante y agregar prueba

✅ Bueno - Funciones individuales

- **validarUsuario, hashPassword, procesarPago:** 100%
- Mantenimiento: Continuar con este nivel

Plan de Acción:

1. **Día 1:** Escribir pruebas para verificarToken (aumentar auth.js a 100%)
2. **Día 2:** Completar pruebas para calcularImpuesto (aumentar payment.js a 100%)

3. **Objetivo:** Alcanzar 85%+ cobertura general

Este tipo de análisis ayuda a priorizar el trabajo de pruebas.

Práctica Evaluada

Enunciado:

Analiza un proyecto existente y mejora su cobertura de pruebas siguiendo las mejores prácticas:

1. Ejecuta reporte de cobertura en un proyecto real
2. Identifica módulos con cobertura menor a 80%
3. Prioriza módulos críticos
4. Escribe pruebas para aumentar la cobertura
5. Documenta el análisis y las mejoras realizadas
6. Establece umbrales de cobertura mínimos

Criterios de Evaluación:

- **Análisis (30%):** Identificación correcta de áreas con baja cobertura, priorización adecuada
- **Mejoras (40%):** Pruebas bien diseñadas que aumentan cobertura, principios FIRST aplicados
- **Documentación (20%):** Reporte claro del antes/después, explicación de decisiones
- **Umbrales (10%):** Configuración de umbrales apropiados

Rúbrica:

- **Excelente (90-100):** Análisis completo, cobertura mejorada significativamente, documentación excelente

- **Bueno (75-89):** Análisis correcto, cobertura mejorada, documentación adecuada
- **Satisfactorio (60-74):** Análisis básico, algunas mejoras, documentación básica
- **Necesita mejorar (<60):** Análisis incompleto, pocas mejoras, documentación insuficiente

Esta práctica evalúa el dominio completo de la cobertura de pruebas.

Mini-Quiz

1. ¿Qué mide la cobertura de líneas?

a) Porcentaje de funciones probadas
b) Porcentaje de líneas de código ejecutadas
c) Porcentaje de pruebas que pasan
d) Porcentaje de bugs encontrados

2. ¿Cuál es un nivel de cobertura recomendado para proyectos comerciales?

a) 50-60%
b) 80-90%
c) 100% siempre
d) No importa la cobertura

3. ¿Qué es más importante en cobertura de pruebas?

a) Alcanzar 100% sin importar cómo
b) Equilibrar cobertura con mantenibilidad y priorizar código crítico
c) Solo probar código simple
d) Ignorar la cobertura

Mini-Quiz (Continuación)

4. ¿Cuál es la diferencia entre cobertura de líneas y cobertura de ramas?

a) No hay diferencia
b) Cobertura de ramas es más estricta y requiere probar todas las condiciones
c) Cobertura de líneas es más difícil
d) Son sinónimos

5. Verdadero o Falso: 100% de cobertura garantiza que no hay bugs.

Verdadero
Falso - La cobertura solo indica qué código se ejecutó, no garantiza corrección

6. ¿Qué debe tener prioridad en cobertura de pruebas?

a) Código simple y trivial
b) Código crítico del negocio
c) Código que nunca se usa
d) Solo código nuevo

7. ¿Qué herramienta es común para medir cobertura en JavaScript?

a) Jest con flag --coverage
b) npm test
c) console.log
d) No hay herramientas

8. ¿Cuál es un error común en cobertura de pruebas?

a) Medir la cobertura regularmente
b) Obsesionarse con 100% sin considerar el costo
c) Priorizar código crítico

d) Usar herramientas de medición

Respuestas del Mini-Quiz

1. **b) Porcentaje de líneas de código ejecutadas** - Cobertura de líneas mide líneas ejecutadas
2. **b) 80-90%** - Nivel recomendado para mayoría de proyectos
3. **b) Equilibrar cobertura con mantenibilidad y priorizar código crítico** - El equilibrio es clave
4. **b) Cobertura de ramas es más estricta y requiere probar todas las condiciones** - Ramas es más completo
5. **Falso** - La cobertura solo indica ejecución, no garantiza corrección
6. **b) Código crítico del negocio** - Prioridad en código crítico
7. **a) Jest con flag --coverage** - Jest incluye cobertura integrada
8. **b) Obsesionarse con 100% sin considerar el costo** - Error común

Si tuvieron dudas, revisen esos conceptos antes de continuar.

Resumen de la Unidad

Puntos Clave:

- La **cobertura de pruebas** mide qué porcentaje del código se ejecuta durante las pruebas
- Los **tipos principales** son: líneas, ramas, funciones y sentencias
- Un nivel de **80-90%** es recomendado para la mayoría de proyectos
- Es importante **priorizar código crítico** sobre código trivial
- Las **herramientas** como Jest, Istanbul, Coverlet ayudan a medir cobertura
- El **equilibrio** entre cobertura y mantenibilidad es fundamental
- 100% de cobertura **no garantiza** código sin bugs
- Debemos **medir regularmente** y usar métricas para guiar mejoras

Este resumen cubre los conceptos fundamentales de cobertura de pruebas.

Tarea para Casa

Actividad:

Mejora la cobertura de pruebas de un proyecto existente aplicando todas las técnicas aprendidas:

1. Selecciona un proyecto con código existente
2. Instala y configura herramientas de cobertura
3. Ejecuta reporte de cobertura inicial
4. Analiza el reporte e identifica áreas de mejora
5. Prioriza módulos críticos
6. Escribe pruebas para aumentar la cobertura
7. Ejecuta reporte final y compara
8. Configura umbrales mínimos de cobertura

Entregables:

- Reporte de cobertura inicial (captura de pantalla)
- Análisis escrito identificando áreas de mejora
- Código de pruebas agregadas
- Reporte de cobertura final (captura de pantalla)
- Documento explicando:
    - Metodología utilizada
    - Decisiones de priorización
    - Mejoras realizadas (antes/después)

    - Umbrales configurados y justificación
- Configuración de herramientas de cobertura

Fecha de entrega:

Una semana después de esta clase

Esta tarea les permitirá practicar la cobertura de pruebas en un proyecto real.

Recursos Adicionales

📚 Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Videos sobre cobertura de código
- Herramientas de análisis de cobertura
- Métricas y reportes
- Mejores prácticas de cobertura

Ver Recursos Adicionales →

¡Gracias!

Unidad IV: Cobertura de pruebas

Próxima clase: Unidad V - Integración continua

Preguntas y dudas

Sigan midiendo y mejorando la cobertura de pruebas regularmente.

Recursos Adicionales - Unidad 04

Cobertura de Pruebas - Enlaces, videos y material complementario

📚 Documentación

[Test Coverage - Martin Fowler Artículo]

Artículo sobre qué es la cobertura de pruebas y cómo interpretarla correctamente.

[Testing Without Mocks Artículo]

Estrategias para lograr buena cobertura sin depender excesivamente de mocks.

🛠️ Herramientas de Cobertura

[Coverlet - Herramienta de Cobertura Herramienta]

Herramienta de código abierto para medir la cobertura de código en .NET.

[dotCover - JetBrains Herramienta]

> Herramienta profesional para análisis de cobertura de código en .NET.

💡 Consejo de Estudio

Para entender la cobertura de pruebas: 1) No te obsesiones con el 100% de cobertura - calidad sobre cantidad, 2) Enfócate en cubrir código crítico y casos de negocio importantes, 3) Usa herramientas de cobertura para identificar áreas sin pruebas, 4) Recuerda que alta cobertura no garantiza calidad - las pruebas deben ser significativas. La cobertura es una métrica, no un objetivo.

Desarrollo Basado en Pruebas (TDD)

Unidad V: Integración continua

Bienvenidos a la quinta unidad. Hoy aprenderemos cómo integrar TDD con prácticas de integración continua.

Competencias y Resultados de Aprendizaje

Competencia Específica CE13

Especificar y ejecutar pruebas de calidad de los prototipos (diseño), simulaciones y los componentes de software desarrollados para garantizar su fidelidad y precisión en todos los niveles (Test units) y garantizando una asertiva aceptación de los usuarios (UCT).

Resultados de Aprendizaje

- Integrar TDD con prácticas de integración continua
- Automatizar pruebas y ejecución continua
- Comprender los beneficios de la integración continua
- Configurar pipelines de CI/CD con TDD
- Gestionar pruebas en entornos de integración continua

La integración continua es fundamental para proyectos modernos de software.

Agenda de la Unidad

1. **Integración de TDD con prácticas de integración continua**
    - ¿Qué es CI/CD?
    - Cómo TDD se integra con CI/CD
    - Flujo de trabajo con TDD y CI
2. **Automatización de pruebas y ejecución continua**
    - Configuración de pipelines
    - Ejecución automática de pruebas
    - Reportes y notificaciones
3. **Beneficios de la integración continua en el desarrollo de software**
    - Detección temprana de problemas
    - Mejora en la calidad
    - Reducción de riesgos

Esta agenda cubre todos los aspectos de la integración continua con TDD.

¿Qué es CI/CD?

CI - Continuous Integration (Integración Continua):

Práctica de desarrollo donde los desarrolladores integran código frecuentemente (varias veces al día) en un repositorio compartido. Cada integración es verificada automáticamente mediante pruebas automatizadas.

CD - Continuous Deployment/Delivery:

- **Continuous Delivery:** Código siempre listo para desplegarse a producción
- **Continuous Deployment:** Despliegue automático a producción después de pasar pruebas

Pipeline CI/CD:

Serie de pasos automatizados que ejecutan cuando se hace commit:

1. Build (compilar código)
2. Test (ejecutar pruebas)
3. Deploy (desplegar si todo pasa)

CI/CD es fundamental para desarrollo moderno de software.

TDD y CI/CD: Integración Perfecta

✅ Cómo TDD se integra con CI/CD:

- **Pruebas primero:** TDD crea pruebas antes del código
- **CI ejecuta pruebas:** Cada commit ejecuta todas las pruebas
- **Feedback inmediato:** Si pruebas fallan, CI lo detecta
- **Confianza:** Código siempre probado antes de integrar

Flujo TDD + CI/CD:

1. Desarrollador escribe prueba (RED)
2. Implementa código (GREEN)
3. Refactoriza
4. Hace commit
5. CI ejecuta todas las pruebas
6. Si pasa → merge/despliegue
7. Si falla → notificación al desarrollador

TDD y CI/CD trabajan juntos perfectamente.

Beneficios de la Integración Continua

1. Detección Temprana de Problemas

Los bugs se detectan minutos después de introducirse, no días o semanas después.

2. Reducción de Riesgos

Integraciones pequeñas y frecuentes reducen el riesgo de conflictos grandes.

3. Mejora en la Calidad

El código siempre está probado y listo para producción.

4. Feedback Rápido

Los desarrolladores saben inmediatamente si su código funciona correctamente.

5. Confianza en el Código

Saber que todas las pruebas pasan da confianza para hacer cambios.

6. Automatización

Reduce trabajo manual y errores humanos.

Los beneficios de CI son enormes para proyectos de software.

Herramientas de CI/CD

Plataformas Populares:

- **GitHub Actions:** Integrado con GitHub, fácil de usar
- **GitLab CI:** Integrado con GitLab, muy completo
- **Jenkins:** Open source, muy flexible y configurable
- **Azure DevOps:** Microsoft, integrado con herramientas Azure
- **CircleCI:** Cloud-based, fácil configuración
- **Travis CI:** Popular para proyectos open source

Ejemplo: GitHub Actions

```
# .github/workflows/test.yml name: Tests on: [push, pull_request] jobs: test: runs-on: ubuntu-latest steps: - uses: actions/checkout@v2 - uses: actions/setup-node@v2 - run: npm install - run: npm test - run: npm run coverage
```

Hay muchas herramientas disponibles, elijan la que mejor se adapte a su proyecto.

Pipeline CI/CD Básico con TDD

Pasos del Pipeline:

1. **Checkout:** Obtener código del repositorio
2. **Setup:** Instalar dependencias y herramientas
3. **Build:** Compilar el proyecto
4. **Test:** Ejecutar todas las pruebas (TDD)
5. **Coverage:** Generar reporte de cobertura
6. **Lint:** Verificar calidad de código
7. **Deploy:** Desplegar si todo pasa (opcional)

Ejemplo Completo:

```
# .github/workflows/ci.yml name: CI Pipeline on: push: branches: [ main, develop ] pull_request: branches: [ main ] jobs: test: runs-on: ubuntu-latest steps: - name: Checkout code uses: actions/checkout@v3 - name: Setup Node.js uses: actions/setup-node@v3 with: node-version: '18' - name: Install dependencies run: npm ci - name: Run tests run: npm test - name: Generate coverage run: npm run test:coverage - name: Upload coverage uses: codecov/codecov-action@v3
```

Este es un pipeline básico que ejecuta pruebas en cada commit.

En la Vida Real: CI/CD con TDD

Escenario Real:

Un equipo de 5 desarrolladores trabajando en una aplicación web. Cada desarrollador hace commits varias veces al día.

❌ Sin CI/CD:

- Desarrollador A hace commit con bug
- Desarrollador B integra código 2 días después
- Bug descubierto durante integración manual
- Tiempo perdido: 4 horas buscando el problema
- Costo: $500 en tiempo de desarrollo

✅ Con CI/CD:

- Desarrollador A hace commit con bug
- CI ejecuta pruebas automáticamente
- Bug detectado en 2 minutos
- Notificación inmediata al desarrollador
- Tiempo perdido: 5 minutos
- Costo: $10 en tiempo de desarrollo

Ahorro: 50x más rápido en detectar problemas, 50x más económico.

Este ejemplo muestra el valor real de CI/CD.

Configuración Práctica: GitHub Actions

Paso 1: Crear archivo de workflow

Crear `.github/workflows/test.yml` en el repositorio:

```
name: Test Suite on: push: branches: [ main, develop ] pull_request: branches: [ main ] jobs: test: runs-on: ubuntu-latest strategy: matrix: node-version: [16.x, 18.x, 20.x] steps: - uses: actions/checkout@v3 - name: Use Node.js ${{ matrix.node-version }} uses: actions/setup-node@v3 with: node-version: ${{ matrix.node-version }} - run: npm ci - run: npm test - run: npm run test:coverage
```

Paso 2: Configurar package.json

```
{ "scripts": { "test": "jest", "test:coverage": "jest --coverage", "test:watch": "jest --watch" } }
```

Esta configuración ejecuta pruebas en cada push y pull request.

Badges y Estado del Build

Badges de Estado:

Los badges muestran el estado del build en el README del proyecto:

```
# README.md [![Tests]
(https://github.com/usuario/proyecto/workflows/Tests/badge.svg)]
(https://github.com/usuario/proyecto/actions) [![Coverage]
(https://codecov.io/gh/usuario/proyecto/branch/main/graph/badge.svg)]
(https://codecov.io/gh/usuario/proyecto)
```

Estados del Build:

- ✅ **Passing (Verde):** Todas las pruebas pasan
- ❌ **Failing (Rojo):** Alguna prueba falla
- 🟡 **Pending (Amarillo):** Build en progreso
- ⚪ **Not Run (Gris):** Build no ejecutado

Beneficios:

- Visibilidad inmediata del estado del proyecto
- Confianza para contribuidores
- Identificación rápida de problemas

Los badges proporcionan visibilidad del estado del proyecto.

Gestionar Pruebas en CI

Mejores Prácticas:

1. Pruebas Rápidas

Las pruebas deben ejecutarse rápidamente. Si tardan mucho, dividirlas en suites paralelas.

2. Pruebas Determinísticas

Las pruebas deben dar el mismo resultado siempre. Evitar dependencias de tiempo o estado aleatorio.

3. Aislar Pruebas

Cada prueba debe ser independiente y poder ejecutarse en cualquier orden.

4. Limpiar Después

Limpiar recursos después de cada prueba (archivos temporales, base de datos, etc.).

5. Reportes Claros

Generar reportes claros que muestren qué pruebas fallaron y por qué.

Estas prácticas hacen que las pruebas funcionen bien en CI.

Errores Comunes en CI/CD

1. Pruebas que Solo Funcionan Localmente

❌ **Error:** Pruebas que pasan en tu máquina pero fallan en CI.

✅ **Solución:** Usar contenedores Docker o entornos consistentes.

2. Dependencias Externas

❌ **Error:** Pruebas que dependen de servicios externos que pueden estar caídos.

✅ **Solución:** Usar mocks o stubs para dependencias externas.

3. Pruebas Lentas

❌ **Error:** Pipeline que tarda horas en ejecutarse.

✅ **Solución:** Paralelizar pruebas, ejecutar solo pruebas relevantes.

4. Ignorar Fallos

❌ **Error:** Permitir que el build pase aunque las pruebas fallen.

✅ **Solución:** Hacer que el build falle si las pruebas fallan.

5. No Limpiar Recursos

❌ **Error:** Dejar archivos temporales o procesos ejecutándose.

✅ **Solución:** Siempre limpiar después de las pruebas.

Estos errores son muy comunes. Estén atentos para evitarlos.

Buenas Prácticas en CI/CD con TDD

1. Ejecutar Pruebas en Cada Commit

Todos los commits deben ejecutar el pipeline completo de pruebas.

2. Falla Rápida

Si una prueba falla, el build debe fallar inmediatamente y notificar al desarrollador.

3. Mantener Builds Rápidos

Los builds deben completarse en menos de 10 minutos idealmente.

4. Paralelizar Pruebas

Ejecutar pruebas en paralelo para reducir tiempo total.

5. Caché de Dependencias

Usar caché para dependencias y builds anteriores.

6. Notificaciones

Notificar al desarrollador cuando el build falla o pasa.

7. Ambientes Separados

Usar ambientes separados para desarrollo, staging y producción.

Estas prácticas hacen que CI/CD sea efectivo y útil.

Práctica Guiada #1: Configurar CI Básico

Objetivo:

Configurar un pipeline CI básico que ejecute pruebas automáticamente.

Tareas:

1. Crear repositorio en GitHub (o GitLab)
2. Crear archivo de workflow CI
3. Configurar ejecución de pruebas
4. Hacer commit y verificar que CI ejecuta
5. Agregar badge de estado al README

Archivo a crear:

`.github/workflows/test.yml`

Resultado esperado:

Al hacer push, CI debe ejecutar automáticamente las pruebas y mostrar el estado.

Vamos paso a paso configurando CI básico.

Práctica Guiada #1: Solución

Archivo .github/workflows/test.yml:

```
name: Run Tests on: push: branches: [ main ] pull_request: branches: [ main ] jobs: test: runs-on: ubuntu-latest steps: - name: Checkout code uses: actions/checkout@v3 - name: Setup Node.js uses: actions/setup-node@v3 with: node-version: '18' cache: 'npm' - name: Install dependencies run: npm ci - name: Run tests run: npm test - name: Generate coverage report run: npm run test:coverage continue-on-error: true
```

README.md con badge:

```
# Mi Proyecto [![Tests](https://github.com/usuario/proyecto/workflows/Run%20Tests/badge.svg)](https://github.com/usuario/proyecto/actions) ## Instalación npm install ## Pruebas npm test
```

Verificación:

- ✅ Hacer push al repositorio
- ✅ Ir a la pestaña "Actions" en GitHub
- ✅ Verificar que el workflow se ejecuta
- ✅ Verificar que las pruebas pasan

Esta configuración básica ejecuta pruebas en cada push.

Práctica Guiada #2: Pipeline Avanzado

Objetivo:

Configurar un pipeline avanzado con múltiples jobs y reportes.

Requisitos:

- Ejecutar pruebas en múltiples versiones de Node.js
- Generar reporte de cobertura
- Ejecutar linter
- Subir reporte de cobertura a Codecov
- Notificar en Slack si falla (opcional)

Tarea:

Crear un pipeline que incluya todos estos pasos y verificar que funciona correctamente.

Practiquen configurando pipelines más avanzados.

Práctica Guiada #2: Solución

Pipeline avanzado:

```
name: CI Pipeline on: push: branches: [ main, develop ] pull_request: branches: [ main ] jobs: lint: runs-on: ubuntu-latest steps: - uses: actions/checkout@v3 - uses: actions/setup-node@v3 with: node-version: '18' - run: npm ci - run: npm run lint test: runs-on: ubuntu-latest strategy: matrix: node-version: [16.x, 18.x, 20.x] steps: - uses: actions/checkout@v3 - uses: actions/setup-node@v3 with: node-version: ${{ matrix.node-version }} cache: 'npm' - run: npm ci - run: npm test - run: npm run test:coverage - name: Upload coverage uses: codecov/codecov-action@v3 with: token: ${{ secrets.CODECOV_TOKEN }} build: needs: [lint, test] runs-on: ubuntu-latest steps: - uses: actions/checkout@v3 - uses: actions/setup-node@v3 with: node-version: '18' - run: npm ci - run: npm run build
```

Características:

- ✅ Job de lint separado
- ✅ Pruebas en múltiples versiones de Node.js
- ✅ Reporte de cobertura subido a Codecov
- ✅ Build solo si lint y test pasan
- ✅ Paralelización de jobs

Este pipeline avanzado incluye múltiples jobs y reportes.

Práctica Evaluada

Enunciado:

Configura un pipeline CI/CD completo para un proyecto existente aplicando todas las mejores prácticas:

1. Configura CI/CD en un proyecto real
2. Ejecuta pruebas automáticamente en cada commit
3. Genera reportes de cobertura
4. Configura notificaciones
5. Agrega badges al README
6. Documenta el proceso

Criterios de Evaluación:

- **Configuración (40%):** Pipeline correctamente configurado, ejecuta pruebas automáticamente
- **Funcionalidad (30%):** Pruebas ejecutan correctamente, reportes generados
- **Mejores Prácticas (20%):** Uso de caché, paralelización, limpieza
- **Documentación (10%):** README actualizado, proceso documentado

Rúbrica:

- **Excelente (90-100):** Pipeline completo y funcional, todas las mejores prácticas aplicadas

- **Bueno (75-89):** Pipeline funcional, mayoría de mejores prácticas aplicadas
- **Satisfactorio (60-74):** Pipeline básico funcional
- **Necesita mejorar (<60):** Pipeline incompleto o no funcional

Esta práctica evalúa el dominio completo de CI/CD con TDD.

Mini-Quiz

1. ¿Qué significa CI?

a) Continuous Integration
b) Code Integration
c) Continuous Improvement
d) Code Inspection

2. ¿Cuándo debe ejecutarse el pipeline CI?

a) Solo al final del proyecto
b) En cada commit o pull request
c) Solo una vez al día
d) Manualmente cuando se necesite

3. ¿Cuál es un beneficio principal de CI/CD?

a) Detección temprana de problemas
b) Más trabajo manual
c) Menos pruebas
d) Código más complejo

Mini-Quiz (Continuación)

4. ¿Qué herramienta es popular para CI/CD?

a) GitHub Actions
b) Jenkins
c) GitLab CI
d) Todas las anteriores

5. Verdadero o Falso: TDD y CI/CD trabajan bien juntos.

Verdadero
Falso

6. ¿Qué debe hacer el pipeline si las pruebas fallan?

a) Continuar de todas formas
b) Fallar y notificar al desarrollador
c) Ignorar los errores
d) Ejecutar solo algunas pruebas

7. ¿Cuál es un error común en CI/CD?

a) Ejecutar pruebas en cada commit
b) Pruebas que solo funcionan localmente
c) Notificar cuando falla
d) Usar caché de dependencias

8. ¿Qué muestra un badge de estado verde?

a) El build está fallando
b) Todas las pruebas pasan
c) El build no se ha ejecutado
d) Hay errores de compilación

Respuestas del Mini-Quiz

1. **a) Continuous Integration** - Integración Continua
2. **b) En cada commit o pull request** - Ejecución automática frecuente
3. **a) Detección temprana de problemas** - Beneficio principal
4. **d) Todas las anteriores** - Todas son populares
5. **Verdadero** - TDD y CI/CD trabajan perfectamente juntos
6. **b) Fallar y notificar al desarrollador** - Feedback inmediato
7. **b) Pruebas que solo funcionan localmente** - Error común
8. **b) Todas las pruebas pasan** - Estado exitoso

Si tuvieron dudas, revisen esos conceptos antes de continuar.

Resumen de la Unidad

Puntos Clave:

- **CI/CD** automatiza la integración y despliegue de código
- **TDD y CI/CD** trabajan perfectamente juntos
- Los **pipelines** ejecutan pruebas automáticamente en cada commit
- La **detección temprana** de problemas es el beneficio principal
- Hay muchas **herramientas** disponibles (GitHub Actions, Jenkins, GitLab CI)
- Las **pruebas deben ser rápidas** y determinísticas para CI
- Los **badges** proporcionan visibilidad del estado del proyecto
- Las **mejores prácticas** incluyen paralelización y caché

Este resumen cubre los conceptos fundamentales de CI/CD con TDD.

Tarea para Casa

Actividad:

Configura un pipeline CI/CD completo para un proyecto aplicando todas las técnicas aprendidas:

1. Selecciona un proyecto existente o crea uno nuevo
2. Configura CI/CD usando GitHub Actions, GitLab CI u otra herramienta
3. Configura ejecución automática de pruebas
4. Agrega generación de reportes de cobertura
5. Configura notificaciones (email, Slack, etc.)
6. Agrega badges al README
7. Optimiza el pipeline (caché, paralelización)
8. Documenta el proceso

Entregables:

- Archivo de configuración del pipeline (YAML)
- README actualizado con badges
- Capturas de pantalla del pipeline ejecutándose
- Documento explicando:
    - Herramienta elegida y por qué
    - Configuración del pipeline
    - Optimizaciones aplicadas

    - Problemas encontrados y soluciones
- URL del repositorio con CI/CD funcionando

Fecha de entrega:

Una semana después de esta clase

Esta tarea les permitirá practicar CI/CD en un proyecto real.

Recursos Adicionales

📚 Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Videos sobre CI/CD con TDD
- Configuración de pipelines
- Integración continua
- Herramientas de CI/CD

Ver Recursos Adicionales →

¡Gracias!

Unidad V: Integración continua

Próxima clase: Unidad VI - Técnicas avanzadas en TDD

Preguntas y dudas

Sigan practicando CI/CD hasta que sea natural en sus proyectos.

Recursos Adicionales - Unidad 05

Integración Continua - Enlaces, videos y material complementario

📚 Documentación

Continuous Integration - Martin Fowler Artículo

Artículo fundacional sobre integración continua y sus beneficios.

Continuous Integration - ThoughtWorks Guía

Guía completa sobre integración continua y mejores prácticas.

🛠️ Herramientas CI/CD

GitHub Actions Herramienta

Plataforma de CI/CD integrada con GitHub para automatizar workflows.

Jenkins Herramienta

Servidor de automatización open source para CI/CD.

Azure DevOps Pipelines Herramienta

> Plataforma de CI/CD de Microsoft para automatizar builds y deployments.

💡 Consejo de Estudio

Para integrar TDD con CI/CD: 1) Configura tu pipeline para ejecutar todas las pruebas automáticamente, 2) Asegúrate de que el build falle si alguna prueba falla, 3) Integra herramientas de cobertura de código en tu pipeline, 4) Automatiza la ejecución de pruebas en cada commit. CI/CD hace que TDD sea aún más valioso al detectar problemas temprano.

Desarrollo Basado en Pruebas (TDD)

Unidad VI: Técnicas avanzadas en TDD

Bienvenidos a la última unidad. Hoy exploraremos técnicas avanzadas que llevan TDD al siguiente nivel.

Competencias y Resultados de Aprendizaje

Competencia Específica CE13

Especificar y ejecutar pruebas de calidad de los prototipos (diseño), simulaciones y los componentes de software desarrollados para garantizar su fidelidad y precisión en todos los niveles (Test units) y garantizando una asertiva aceptación de los usuarios (UCT).

Resultados de Aprendizaje

- Utilizar dobles de prueba (mocks, stubs) efectivamente
- Aplicar TDD orientado a objetos
- Implementar pruebas de regresión en TDD
- Dominar técnicas avanzadas de testing
- Resolver problemas complejos usando TDD avanzado

Las técnicas avanzadas permiten manejar casos complejos en TDD.

Agenda de la Unidad

1. **Utilización de dobles de prueba (mocks, stubs)**
    - ¿Qué son mocks y stubs?
    - Cuándo usar cada uno
    - Ejemplos prácticos
2. **TDD orientado a objetos**
    - Pruebas de clases y objetos
    - Herencia y polimorfismo
    - Diseño orientado a pruebas
3. **Pruebas de regresión en TDD**
    - ¿Qué son pruebas de regresión?
    - Cómo prevenir regresiones
    - Mantenimiento de suites de regresión

Esta agenda cubre técnicas avanzadas esenciales en TDD.

Dobles de Prueba: Mocks y Stubs

¿Qué son los Dobles de Prueba?

Objetos que reemplazan dependencias reales durante las pruebas para aislar el código que se está probando.

Tipos de Dobles:

- **Stub:** Proporciona respuestas predefinidas a llamadas
- **Mock:** Verifica que se llamaron métodos específicos
- **Spy:** Registra llamadas pero delega al objeto real
- **Fake:** Implementación simplificada pero funcional

¿Por qué usar Dobles?

- Aislar código bajo prueba
- Evitar dependencias externas (BD, APIs, archivos)
- Hacer pruebas más rápidas
- Controlar comportamiento de dependencias

Los dobles de prueba son fundamentales para pruebas aisladas.

Stubs: Respuestas Predefinidas

¿Qué es un Stub?

Un **stub** es un objeto que proporciona respuestas predefinidas a las llamadas durante las pruebas. No verifica cómo se llama, solo devuelve valores.

Ejemplo: Stub de Servicio de Email

```
// Servicio real (lento, requiere conexión) class EmailService { enviarEmail(destinatario, mensaje) { // Conexión real a servidor SMTP // Tarda segundos } } // Stub para pruebas class EmailServiceStub { enviarEmail(destinatario, mensaje) { // Respuesta inmediata, sin conexión real return { exito: true, id: '12345' }; } } // Prueba usando stub test('debe enviar email de bienvenida', () => { const emailStub = new EmailServiceStub(); const usuario = new Usuario(emailStub); usuario.registrar('juan@email.com'); // Prueba rápida sin conexión real expect(usuario.emailEnviado).toBe(true); });
```

Los stubs proporcionan respuestas rápidas sin dependencias reales.

Mocks: Verificación de Comportamiento

¿Qué es un Mock?

Un **mock** es un objeto que verifica que se llamaron métodos específicos con parámetros específicos. Se enfoca en el comportamiento, no solo en el resultado.

Ejemplo: Mock con Jest

```
// Código a probar class ProcesadorPagos { constructor(servicioPago) { this.servicioPago = servicioPago; } procesar(monto) { if (monto > 0) { return this.servicioPago.cobrar(monto); } throw new Error('Monto inválido'); } } // Prueba con mock test('debe llamar servicioPago.cobrar con monto correcto', () => { // Crear mock const mockServicioPago = { cobrar: jest.fn().mockReturnValue({ exito: true }) }; const procesador = new ProcesadorPagos(mockServicioPago); // Act procesador.procesar(100); // Assert: Verificar que se llamó el método expect(mockServicioPago.cobrar).toHaveBeenCalled(); expect(mockServicioPago.cobrar).toHaveBeenCalledWith(100); expect(mockServicioPago.cobrar).toHaveBeenCalledTimes(1); });
```

Los mocks verifican comportamiento, no solo resultados.

Cuándo Usar Stubs vs Mocks

✅ Usar Stub cuando:

- Necesitas respuesta rápida
- No te importa cómo se llama
- Solo necesitas un valor de retorno
- Quieres simular dependencias lentas
- Ejemplo: Servicio de base de datos

✅ Usar Mock cuando:

- Necesitas verificar comportamiento
- Quieres asegurar que se llamó un método
- Necesitas verificar parámetros
- Quieres contar llamadas
- Ejemplo: Servicio de logging

⚠️ Regla de Oro:

"Usa el doble más simple que cumpla tu necesidad. Prefiere stubs sobre mocks cuando sea posible."

Elegir el doble correcto es importante para pruebas claras.

Ejemplo Práctico: Stub vs Mock

Escenario: Sistema de Notificaciones

```
// Código a probar class GestorNotificaciones { constructor(servicioEmail, logger) { this.servicioEmail = servicioEmail; this.logger = logger; } notificarUsuario(email, mensaje) { this.logger.info(`Enviando notificación a ${email}`); const resultado = this.servicioEmail.enviar(email, mensaje); this.logger.info(`Notificación enviada: ${resultado.exito}`); return resultado; } } // Prueba usando Stub y Mock test('debe enviar email y registrar en log', () => { // Stub: Solo necesitamos respuesta const emailStub = { enviar: jest.fn().mockReturnValue({ exito: true }) }; // Mock: Necesitamos verificar que se llamó const loggerMock = { info: jest.fn() }; const gestor = new GestorNotificaciones(emailStub, loggerMock); gestor.notificarUsuario('juan@email.com', 'Hola'); // Verificar stub fue llamado expect(emailStub.enviar).toHaveBeenCalledWith('juan@email.com', 'Hola'); // Verificar mock fue llamado correctamente expect(loggerMock.info).toHaveBeenCalledTimes(2); expect(loggerMock.info).toHaveBeenCalledWith('Enviando notificación a juan@email.com'); });
```

Este ejemplo muestra cuándo usar cada tipo de doble.

TDD Orientado a Objetos

Pruebas de Clases y Objetos:

En TDD orientado a objetos, probamos el comportamiento de clases y objetos, no solo funciones.

Ejemplo: Prueba de Clase

```
// Prueba primero (RED) describe('Calculadora', () => { test('debe sumar dos números', () => { const calc = new Calculadora(); expect(calc.sumar(2, 3)).toBe(5); }); test('debe mantener historial de operaciones', () => { const calc = new Calculadora(); calc.sumar(2, 3); expect(calc.historial).toHaveLength(1); expect(calc.historial[0]).toEqual({ operacion: 'sumar', a: 2, b: 3, resultado: 5 }); }); }); // Implementación (GREEN) class Calculadora { constructor() { this.historial = []; } sumar(a, b) { const resultado = a + b; this.historial.push({ operacion: 'sumar', a, b, resultado }); return resultado; } }
```

TDD orientado a objetos prueba comportamiento de objetos.

Herencia y Polimorfismo en TDD

Pruebas de Herencia:

```
// Clase base class Animal { hacerSonido() { throw new Error('Método abstracto'); } } // Clase derivada class Perro extends Animal { hacerSonido() { return 'Guau'; } } // Pruebas describe('Perro', () => { test('debe hacer sonido de perro', () => { const perro = new Perro(); expect(perro.hacerSonido()).toBe('Guau'); }); test('debe ser instancia de Animal', () => { const perro = new Perro(); expect(perro).toBeInstanceOf(Animal); }); }); // Prueba polimórfica describe('Polimorfismo', () => { test('debe tratar diferentes animales polimórficamente', () => { const animales = [ new Perro(), new Gato(), new Pajaro() ]; const sonidos = animales.map(a => a.hacerSonido()); expect(sonidos).toEqual(['Guau', 'Miau', 'Pío']); }); });
```

TDD funciona bien con herencia y polimorfismo.

Pruebas de Regresión en TDD

¿Qué son las Pruebas de Regresión?

Pruebas que verifican que funcionalidad existente sigue funcionando después de cambios en el código.

Cómo TDD Previene Regresiones:

- **Suite completa:** Todas las pruebas se ejecutan en cada cambio
- **Detección inmediata:** Si algo se rompe, las pruebas fallan
- **Confianza:** Saber que el código existente sigue funcionando
- **Documentación:** Las pruebas documentan comportamiento esperado

Ejemplo de Regresión Detectada:

```
// Código original (funciona) function calcularDescuento(monto) { if (monto > 100) { return monto * 0.1; } return 0; } // Cambio introduce bug function calcularDescuento(monto) { if (monto >= 100) { // ← Cambio: ahora incluye 100 return monto * 0.1; } return 0; } // Prueba de regresión detecta el cambio test('debe retornar 0 para monto exactamente 100', () => { expect(calcularDescuento(100)).toBe(0); // ← Falla después del cambio });
```

Las pruebas de regresión son esenciales para mantener estabilidad.

Mantenimiento de Suites de Regresión

Mejores Prácticas:

1. Ejecutar Todas las Pruebas Regularmente

No solo las pruebas nuevas. Ejecutar toda la suite en cada cambio.

2. Mantener Pruebas Actualizadas

Cuando cambia el comportamiento esperado, actualizar las pruebas correspondientes.

3. Organizar por Funcionalidad

Agrupar pruebas relacionadas para facilitar mantenimiento.

4. Pruebas Rápidas

Mantener pruebas rápidas para ejecutar suite completa frecuentemente.

5. Documentar Cambios

Cuando se actualiza una prueba, documentar por qué cambió el comportamiento.

El mantenimiento adecuado mantiene las suites útiles.

En la Vida Real: Técnicas Avanzadas

Escenario Real:

Un sistema de e-commerce necesita integrar con múltiples servicios externos (pagos, envíos, inventario).

❌ Sin técnicas avanzadas:

- Pruebas dependen de servicios reales
- Pruebas lentas (segundos por prueba)
- Pruebas fallan si servicios están caídos
- Difícil probar casos edge
- Tiempo total: 30 minutos

✅ Con técnicas avanzadas:

- Stubs para servicios externos
- Pruebas rápidas (milisegundos)
- Pruebas aisladas y confiables
- Fácil probar casos edge
- Tiempo total: 2 minutos

> **Resultado:** 15x más rápido, pruebas más confiables, mejor cobertura de casos edge.

Este ejemplo muestra el valor real de técnicas avanzadas.

Errores Comunes en Técnicas Avanzadas

1. Abusar de Mocks

❌ **Error:** Mockear todo, incluso objetos simples.

✅ **Correcto:** Usar mocks solo cuando es necesario verificar comportamiento.

2. Mocks Demasiado Específicos

❌ **Error:** Verificar cada detalle de implementación.

✅ **Correcto:** Verificar comportamiento, no implementación.

3. Ignorar Pruebas de Regresión

❌ **Error:** Solo ejecutar pruebas nuevas.

✅ **Correcto:** Ejecutar toda la suite en cada cambio.

4. Stubs que No Reflejan Realidad

❌ **Error:** Stubs que devuelven valores irreales.

✅ **Correcto:** Stubs que simulan comportamiento realista.

5. No Limpiar Mocks

❌ **Error:** Estado de mocks persiste entre pruebas.

✅ **Correcto:** Limpiar/resetear mocks en beforeEach.

Estos errores son muy comunes. Estén atentos para evitarlos.

Buenas Prácticas en Técnicas Avanzadas

1. Usar el Doble Más Simple

Preferir stubs sobre mocks cuando sea posible. Solo usar mocks cuando necesites verificar comportamiento.

2. Aislar Dependencias

Usar dobles para aislar código bajo prueba de dependencias externas.

3. Mantener Pruebas Legibles

Las pruebas deben ser fáciles de entender, incluso con mocks y stubs.

4. Verificar Comportamiento, No Implementación

Probar qué hace el código, no cómo lo hace internamente.

5. Ejecutar Suite Completa

Ejecutar todas las pruebas de regresión en cada cambio.

6. Documentar Decisiones

Documentar por qué se usa un mock o stub específico.

7. Revisar y Refactorizar

Revisar pruebas regularmente y refactorizar cuando sea necesario.

Estas prácticas hacen que las técnicas avanzadas sean efectivas.

Práctica Guiada #1: Crear Stubs

Objetivo:

Crear stubs para aislar código de dependencias externas.

Código a probar:

```
class GestorUsuarios { constructor(repositorioUsuarios) { this.repositorio = repositorioUsuarios; } obtenerUsuarioPorId(id) { return this.repositorio.buscarPorId(id); } } // Repositorio real (lento, requiere BD) class RepositorioUsuarios { buscarPorId(id) { // Consulta real a base de datos // Tarda segundos } }
```

Tareas:

1. Crear stub de RepositorioUsuarios
2. Escribir prueba usando el stub
3. Verificar que la prueba es rápida
4. Probar diferentes casos (usuario existe, no existe)

Vamos paso a paso creando stubs.

Práctica Guiada #1: Solución

Stub y Pruebas:

```
// Stub del repositorio class RepositorioUsuariosStub { constructor() { this.usuarios = { '1': { id: '1', nombre: 'Juan', email: 'juan@email.com' }, '2': { id: '2', nombre: 'María', email: 'maria@email.com' } }; } buscarPorId(id) { return this.usuarios[id] || null; } } // Pruebas usando stub describe('GestorUsuarios', () => { test('debe obtener usuario existente', () => { const stubRepo = new RepositorioUsuariosStub(); const gestor = new GestorUsuarios(stubRepo); const usuario = gestor.obtenerUsuarioPorId('1'); expect(usuario).toEqual({ id: '1', nombre: 'Juan', email: 'juan@email.com' }); }); test('debe retornar null para usuario inexistente', () => { const stubRepo = new RepositorioUsuariosStub(); const gestor = new GestorUsuarios(stubRepo); const usuario = gestor.obtenerUsuarioPorId('999'); expect(usuario).toBeNull(); }); });
```

Ventajas del Stub:

- ✅ Prueba rápida (milisegundos vs segundos)
- ✅ No requiere base de datos
- ✅ Control total sobre datos de prueba
- ✅ Prueba aislada

Este stub hace las pruebas rápidas y aisladas.

Práctica Guiada #2: Usar Mocks

Objetivo:

Usar mocks para verificar comportamiento de dependencias.

Código a probar:

```
class ServicioNotificaciones { constructor(logger, emailService) { this.logger = logger; this.emailService = emailService; } enviarNotificacion(email, mensaje) { this.logger.info(`Enviando a ${email}`); const resultado = this.emailService.enviar(email, mensaje); this.logger.info(`Resultado: ${resultado.exito}`); return resultado; } }
```

Tarea:

Escribir pruebas usando mocks para verificar que:

1. Se llama logger.info dos veces
2. Se llama emailService.enviar con parámetros correctos
3. Se registra el resultado correctamente

Practiquen usando mocks para verificar comportamiento.

Práctica Guiada #2: Solución

Pruebas con Mocks:

```
describe('ServicioNotificaciones', () => { test('debe registrar y enviar email correctamente', () => { // Crear mocks const loggerMock = { info: jest.fn() }; const emailServiceMock = { enviar: jest.fn().mockReturnValue({ exito: true }) }; const servicio = new ServicioNotificaciones(loggerMock, emailServiceMock); // Act const resultado = servicio.enviarNotificacion('juan@email.com', 'Hola'); // Assert: Verificar comportamiento expect(loggerMock.info).toHaveBeenCalledTimes(2); expect(loggerMock.info).toHaveBeenNthCalledWith(1, 'Enviando a juan@email.com'); expect(loggerMock.info).toHaveBeenNthCalledWith(2, 'Resultado: true'); expect(emailServiceMock.enviar).toHaveBeenCalledTimes(1); expect(emailServiceMock.enviar).toHaveBeenCalledWith('juan@email.com', 'Hola'); expect(resultado).toEqual({ exito: true }); }); });
```

Verificaciones del Mock:

- ✅ Verifica que logger.info se llamó 2 veces
- ✅ Verifica parámetros específicos de cada llamada
- ✅ Verifica que emailService.enviar se llamó correctamente
- ✅ Verifica el resultado retornado

Este ejemplo muestra cómo usar mocks para verificar comportamiento.

Práctica Evaluada

Enunciado:

Implementa pruebas avanzadas para un sistema complejo aplicando todas las técnicas aprendidas:

1. Identifica dependencias externas en un módulo
2. Crea stubs para servicios lentos o externos
3. Usa mocks para verificar comportamiento
4. Escribe pruebas de regresión
5. Aplica TDD orientado a objetos
6. Documenta decisiones de diseño

Criterios de Evaluación:

- **Uso de Dobles (40%):** Stubs y mocks usados correctamente, justificación clara
- **TDD OOP (25%):** Pruebas orientadas a objetos bien diseñadas
- **Regresión (20%):** Pruebas de regresión implementadas
- **Calidad (15%):** Pruebas mantenibles, legibles, bien organizadas

Rúbrica:

- **Excelente (90-100):** Todas las técnicas aplicadas correctamente, diseño impecable

- **Bueno (75-89):** Técnicas aplicadas correctamente, diseño claro
- **Satisfactorio (60-74):** Algunas técnicas aplicadas, diseño funcional
- **Necesita mejorar (<60):** Técnicas no aplicadas correctamente

Esta práctica evalúa el dominio completo de técnicas avanzadas.

Mini-Quiz

1. ¿Qué es un stub?

a) Un objeto que verifica comportamiento
b) Un objeto que proporciona respuestas predefinidas
c) Un objeto real
d) Un tipo de prueba

2. ¿Cuál es la diferencia principal entre stub y mock?

a) No hay diferencia
b) Stub proporciona respuestas, mock verifica comportamiento
c) Stub es más rápido
d) Mock es más simple

3. ¿Qué son las pruebas de regresión?

a) Pruebas nuevas
b) Pruebas que verifican que funcionalidad existente sigue funcionando
c) Pruebas rápidas
d) Pruebas de integración

Mini-Quiz (Continuación)

4. ¿Cuándo deberías usar un stub?

a) Cuando necesitas verificar comportamiento
b) Cuando solo necesitas una respuesta rápida sin verificar cómo se llama
c) Siempre
d) Nunca

5. Verdadero o Falso: TDD funciona bien con programación orientada a objetos.

Verdadero
Falso

6. ¿Qué verifica un mock?

a) Solo valores de retorno
b) Comportamiento: qué métodos se llamaron y con qué parámetros
c) Velocidad
d) Cobertura

7. ¿Por qué usar dobles de prueba?

a) Para hacer pruebas más lentas
b) Para aislar código y evitar dependencias externas
c) Para complicar las pruebas
d) Para evitar escribir pruebas

8. ¿Cuál es un error común con mocks?

a) Usarlos cuando son necesarios
b) Verificar detalles de implementación en lugar de comportamiento
c) Limpiarlos entre pruebas
d) Documentar su uso

Respuestas del Mini-Quiz

1. **b) Un objeto que proporciona respuestas predefinidas** - Definición de stub
2. **b) Stub proporciona respuestas, mock verifica comportamiento** - Diferencia clave
3. **b) Pruebas que verifican que funcionalidad existente sigue funcionando** - Definición
4. **b) Cuando solo necesitas una respuesta rápida sin verificar cómo se llama** - Uso de stub
5. **Verdadero** - TDD funciona excelentemente con OOP
6. **b) Comportamiento: qué métodos se llamaron y con qué parámetros** - Verificación de mock
7. **b) Para aislar código y evitar dependencias externas** - Propósito principal
8. **b) Verificar detalles de implementación en lugar de comportamiento** - Error común

Si tuvieron dudas, revisen esos conceptos.

Resumen de la Unidad

Puntos Clave:

- Los **dobles de prueba** (stubs, mocks) aíslan código de dependencias
- **Stubs** proporcionan respuestas predefinidas
- **Mocks** verifican comportamiento y llamadas
- **TDD orientado a objetos** prueba comportamiento de clases y objetos
- Las **pruebas de regresión** previenen que cambios rompan funcionalidad existente
- Usar el **doble más simple** que cumpla la necesidad
- Verificar **comportamiento, no implementación**
- Ejecutar **suite completa** de regresión en cada cambio

Este resumen cubre los conceptos fundamentales de técnicas avanzadas.

Tarea para Casa

Actividad:

Implementa técnicas avanzadas de TDD en un proyecto aplicando todas las técnicas aprendidas:

1. Identifica dependencias externas en tu proyecto
2. Crea stubs para servicios lentos o externos
3. Usa mocks para verificar comportamiento crítico
4. Escribe pruebas de regresión para funcionalidad existente
5. Aplica TDD orientado a objetos donde sea apropiado
6. Documenta decisiones de diseño
7. Compara velocidad y confiabilidad antes/después

Entregables:

- Código de pruebas con stubs y mocks
- Pruebas de regresión implementadas
- Documento explicando:
    - Dependencias identificadas
    - Decisión de usar stub vs mock
    - Pruebas de regresión creadas
    - Mejoras en velocidad y confiabilidad
    - Lecciones aprendidas
- Métricas antes/después (velocidad, cobertura)

Fecha de entrega:

Una semana después de esta clase

Esta tarea les permitirá practicar técnicas avanzadas en un proyecto real.

Recursos Adicionales

📚 Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Videos sobre mocks y stubs
- Frameworks de mocking
- Técnicas avanzadas de testing
- Refactoring con TDD

Ver Recursos Adicionales →

¡Gracias!

Unidad VI: Técnicas avanzadas en TDD

Fin del Curso

Preguntas y dudas

¡Éxito en sus proyectos con TDD!

Han completado el curso completo de TDD. Sigan practicando estas técnicas.

Recursos Adicionales - Unidad 06

Técnicas Avanzadas en TDD - Enlaces, videos y material complementario

📚 Documentación Avanzada

[Mocks Aren't Stubs - Martin Fowler Artículo]

Artículo clásico sobre la diferencia entre mocks, stubs y fakes, y cuándo usar cada uno.

[Testing Without Mocks Artículo]

Estrategias avanzadas para escribir pruebas sin depender excesivamente de mocks.

[Avoid Nesting When You're Testing Artículo]

Mejores prácticas para estructurar pruebas complejas y mantenerlas legibles.

🎥 Videos Educativos

[Técnicas Avanzadas de TDD Video]

Charla sobre técnicas avanzadas y patrones para aplicar TDD en proyectos complejos.

[Mocking y Stubbing en TDD Tutorial]

Tutorial sobre cómo usar mocks y stubs efectivamente en TDD.

🛠️ Frameworks y Herramientas

[Moq - Framework de Mocking Herramienta]

Framework popular para crear mocks en .NET y C#.

[xUnit - Framework de Testing Herramienta]

Framework moderno de testing para .NET con excelente soporte para TDD.

💡 Consejo de Estudio

Para dominar técnicas avanzadas: 1) Practica con diferentes frameworks de mocking, 2) Aprende a identificar cuándo usar mocks vs stubs vs fakes, 3) Estudia patrones de diseño que facilitan el testing, 4) Experimenta con pruebas de integración además de unitarias, 5) Lee código de proyectos open source que usen TDD. Las técnicas avanzadas vienen con experiencia práctica.
