# Contents



**Clase 10: Calidad en las Fases de Implementación y Pruebas**

**Objetivos de la clase:**

- Comprender la importancia de la calidad en las fases de implementación y pruebas del ciclo de vida del software.
- Identificar las mejores prácticas para la implementación de código de alta calidad.
- Conocer las diferentes técnicas y niveles de pruebas de software.
- Aplicar métricas para medir la efectividad de las pruebas y la calidad del código.

**Contenido Teórico Detallado:**

**1. Calidad en la Fase de Implementación (Codificación):**

La fase de implementación es donde el diseño se transforma en código ejecutable. La calidad del código impacta directamente la mantenibilidad, rendimiento y seguridad del software.

- **Estándares de Codificación:**
  - **Definición:** Conjunto de reglas y directrices para escribir código consistente, legible y fácil de mantener.
  - **Beneficios:** Mejora la colaboración en el equipo, reduce errores, facilita la depuración y el refactoring.
  - **Ejemplos:** Convenciones de nombres (variables, funciones, clases), indentación, comentarios, manejo de errores, límites de longitud de línea, uso de patrones de diseño.
  - **Herramientas:** Linters y analizadores estáticos (ej. ESLint para JavaScript, SonarQube) ayudan a aplicar automáticamente los estándares.
- **Revisiones de Código (Code Reviews):**
  - **Definición:** Proceso donde otros miembros del equipo examinan el código antes de su integración para identificar posibles defectos, problemas de estilo y oportunidades de mejora.
  - **Beneficios:** Detecta errores tempranamente, comparte conocimiento entre el equipo, mejora la calidad del código y promueve el cumplimiento de estándares.
  - **Proceso:** El autor del código solicita una revisión a uno o varios compañeros. Los revisores comentan sobre el código, sugieren mejoras y aprueban o rechazan la solicitud.
  - **Herramientas:** Plataformas como GitHub, GitLab y Bitbucket facilitan las revisiones de código.
- **Refactoring:**
  - **Definición:** Proceso de mejorar la estructura interna del código sin cambiar su comportamiento externo.
  - **Objetivos:** Reducir la complejidad, mejorar la legibilidad, eliminar código duplicado y facilitar futuras modificaciones.
  - **Técnicas:** Extraer métodos, renombrar variables/funciones, reemplazar algoritmos complejos, introducir patrones de diseño.
  - **Importancia:** Mantiene el código limpio y fácil de mantener a largo plazo.

**2. Calidad en la Fase de Pruebas:**

Las pruebas son esenciales para verificar que el software cumple con los requisitos y funciona correctamente.

- **Niveles de Pruebas:**
  - **Pruebas Unitarias:** Prueban unidades individuales de código (funciones, clases, módulos) de forma aislada.
    - **Objetivo:** Verificar que cada unidad funciona correctamente.
    - **Técnicas:** Test-Driven Development (TDD) donde se escriben las pruebas antes del código.
    - **Herramientas:** JUnit (Java), pytest (Python), NUnit (.NET).
  - **Pruebas de Integración:** Prueban la interacción entre diferentes unidades o módulos.

        * **Objetivo:** Verificar que las diferentes partes del sistema funcionan juntas correctamente.
        * **Técnicas:** Pruebas "top-down", "bottom-up" y "big-bang".
    - **Pruebas de Sistema:** Prueban el sistema completo para verificar que cumple con los requisitos funcionales y no funcionales.
        * **Objetivo:** Simular el uso real del sistema.
        * **Técnicas:** Pruebas de caja negra (sin conocimiento del código interno).
    - **Pruebas de Aceptación:** Prueban el sistema desde la perspectiva del usuario final.
        * **Objetivo:** Verificar que el sistema cumple con las expectativas del cliente y está listo para su lanzamiento.
        * **Técnicas:** Pruebas beta, pruebas con usuarios reales.
- **Tipos de Pruebas:**
    - **Pruebas Funcionales:** Verifican que el software funciona según los requisitos.
        * **Ejemplos:** Pruebas de casos de uso, pruebas de flujo de datos.
    - **Pruebas No Funcionales:** Verifican atributos como rendimiento, seguridad, usabilidad y confiabilidad.
        * **Ejemplos:** Pruebas de carga, pruebas de estrés, pruebas de penetración, pruebas de usabilidad.
    - **Pruebas de Regresión:** Verifican que las nuevas modificaciones no hayan introducido errores en el código existente.
        * **Importancia:** Aseguran que las correcciones y mejoras no rompan la funcionalidad existente.
        * **Automatización:** Las pruebas de regresión deben ser automatizadas para ejecutarse rápidamente después de cada cambio.
- **Métricas de Pruebas:**
    - **Cobertura de Código:** Porcentaje de código que ha sido ejecutado por las pruebas.
        * **Tipos:** Cobertura de líneas, cobertura de ramas, cobertura de condiciones.
        * **Objetivo:** Asegurar que las pruebas ejercitan todas las partes del código.
    - **Densidad de Defectos:** Número de defectos encontrados por cada 1000 líneas de código (KLOC).
        * **Objetivo:** Medir la calidad del código y la efectividad de las pruebas.
    - **Tasa de Escape de Defectos:** Porcentaje de defectos que escapan las pruebas y se encuentran en producción.
        * **Objetivo:** Medir la efectividad del proceso de pruebas.

**Ejemplos y Casos de Estudio:**

- **Caso de Estudio: Implementación y Pruebas en un Proyecto de e-commerce**
    - **Implementación:** El equipo adoptó estándares de codificación estrictos y realizó revisiones de código exhaustivas. Se utilizó un linter para asegurar el cumplimiento de los estándares.
    - **Pruebas:** Se implementaron pruebas unitarias para cada componente del sistema, pruebas de integración para verificar la interacción entre el carrito de compras, el sistema de pago y el inventario, y pruebas de sistema para simular la experiencia del usuario. También se realizaron pruebas de carga para asegurar que el sistema pudiera manejar un gran número de usuarios simultáneamente.
    - **Resultados:** La calidad del código mejoró significativamente, se redujeron los errores y se mejoró el rendimiento del sistema. La tasa de escape de defectos fue muy baja.
- **Ejemplo de Refactoring:**
    - **Código Original:**

      ```
      java public int calcularPrecioTotal(List<Producto> productos, boolean esMiembro)
      {     int total = 0;     for (Producto producto : productos) {         total +=
      producto.getPrecio();     }     if (esMiembro) {         total = (int) (total *
      0.9); // 10% de descuento     }     return total; }
      ```

    - **Código Refactorizado:**

“‘java private int calcularPrecioSinDescuento(List productos) { return productos.stream().mapToInt(Producto::get
}

private int aplicarDescuentoMiembro(int precio) { return (int) (precio * 0.9); }

public int calcularPrecioTotal(List productos, boolean esMiembro) { int total = calcularPrecioSin-Descuento(productos); if (esMiembro) { total = aplicarDescuentoMiembro(total); } return total; } “‘

- **Explicación:** Se extrajeron métodos para calcular el precio sin descuento y aplicar el descuento de miembro, haciendo el código más legible y fácil de mantener.

**Problemas Prácticos y Ejercicios con Soluciones:**

1. **Establecer Estándares de Codificación:**

   - **Problema:** Define un conjunto de estándares de codificación para un proyecto en Java. Incluye convenciones de nombres para variables, funciones y clases, reglas de indentación y comentarios, y manejo de errores.
   - **Solución:**
     - **Convenciones de Nombres:**
       - Variables: `camelCase` (ej. `nombreCliente`)
       - Funciones: `camelCase` (ej. `obtenerNombreCliente()`)
       - Clases: `PascalCase` (ej. `Cliente`)
       - Constantes: `UPPER_SNAKE_CASE` (ej. `MAX_CLIENTES`)
     - **Indentación:** 4 espacios por nivel de indentación.
     - **Comentarios:** Comentar cada clase y función con una descripción de su propósito y parámetros. Utilizar comentarios inline para explicar lógica compleja.
     - **Manejo de Errores:** Utilizar bloques `try-catch` para manejar excepciones. Registrar las excepciones con un logger.
   - **Pruebas Unitarias con JUnit:**
   - **Problema:** Escribe pruebas unitarias para la siguiente función en Java que calcula el factorial de un número:

     ```
     java public int factorial(int n) {     if (n == 0) {         return 1;     }
     else {         return n * factorial(n - 1);     } }
     ```

   - **Solución:**

     “‘java import org.junit.jupiter.api.Test; import static org.junit.jupiter.api.Assertions.*;

     class FactorialTest { @Test void factorialDeCeroDebeSerUno() { assertEquals(1, new Factorial().factorial(0)); }

     ```
     @Test
     void factorialDeUnoDebeSerUno() {
         assertEquals(1, new Factorial().factorial(1));
     }

     @Test
     void factorialDeCincoDebeSerCientoVeinte() {
         assertEquals(120, new Factorial().factorial(5));
     }

     @Test
     void factorialDeNumeroNegativoDebeLanzarExcepcion() {
         assertThrows(IllegalArgumentException.class, () -> new Factorial().factorial(-1));
     ```

```
}
```

} “‘ 3. **Cálculo de Cobertura de Código:**

- **Problema:** Dada la siguiente función y el conjunto de pruebas, calcula la cobertura de líneas:

  ```
  java public boolean esMayorDeEdad(int edad) {     if (edad >= 18) {         return true;     } else {         return false;     } }
  ```

  ```
  java @Test void esMayorDeEdad() {     assertTrue(esMayorDeEdad(20)); }
  ```

- **Solución:** La prueba solo ejecuta la rama `if (edad >= 18)`. Para obtener una cobertura completa, se necesita una prueba que ejecute la rama `else`. La cobertura actual es del 50%.

**Materiales Complementarios Recomendados:**

- **Libros:**
  - ”Clean Code: A Handbook of Agile Software Craftsmanship” by Robert C. Martin
  - ”Refactoring: Improving the Design of Existing Code” by Martin Fowler
  - ”Software Testing” by Ron Patton
- **Artículos:**
  - ”Code Review Best Practices” - Atlassian
  - ”The Importance of Unit Testing” - Martin Fowler
- **Herramientas:**
  - SonarQube (análisis estático de código)
  - JUnit, pytest, NUnit (frameworks de pruebas unitarias)
  - JaCoCo (cobertura de código para Java)

Esta clase proporciona una base sólida para comprender y aplicar las mejores prácticas de calidad en las fases de implementación y pruebas del ciclo de vida del software. Al adoptar estándares de codificación, realizar revisiones de código, refactorizar el código y aplicar diferentes niveles y tipos de pruebas, se puede mejorar significativamente la calidad del software y reducir los costos asociados con los defectos.