Contents

“‘

Clase 9: Calidad en el ciclo de vida del software - Requisitos y Diseño

Objetivos de la Clase:

- Comprender la importancia de la calidad en las fases de requisitos y diseño del ciclo de vida del software.
- Identificar técnicas y prácticas para asegurar la calidad de los requisitos.
- Analizar cómo las decisiones de diseño impactan la calidad del software.
- Aplicar principios de diseño orientado a la calidad.

Contenido Teórico Detallado:

La calidad del software no es algo que se ”agrega” al final del ciclo de desarrollo. Debe ser una consideración primordial desde las primeras etapas, comenzando con la definición de requisitos y continuando a través del diseño. Ignorar la calidad en estas fases tempranas lleva inevitablemente a problemas costosos y difíciles de corregir más adelante.

1. Calidad en la Fase de Requisitos:

Los requisitos definen lo que el software debe hacer y cómo debe comportarse. Requisitos ambiguos, incompletos, inconsistentes o incorrectos son la principal fuente de defectos.

- **Importancia de la Calidad de los Requisitos:**
  - *Reducción de Costos:* Detectar y corregir errores en la fase de requisitos es significativamente más barato que hacerlo en las fases de codificación o pruebas.
  - *Reducción de Riesgos:* Requisitos claros y precisos minimizan el riesgo de desarrollar software que no cumpla con las necesidades del usuario.
  - *Mejora de la Comunicación:* Un buen conjunto de requisitos sirve como base para la comunicación entre los interesados (clientes, desarrolladores, testers, etc.).
- **Técnicas para Asegurar la Calidad de los Requisitos:**
  - *Recopilación exhaustiva:* Involucrar a todos los interesados en el proceso de recopilación de requisitos, utilizando técnicas como entrevistas, talleres, encuestas y análisis de documentos.
  - *Elaboración de casos de uso/historias de usuario:* Describir el comportamiento del sistema desde la perspectiva del usuario, facilitando la comprensión y la validación de los requisitos.
  - *Prototipado:* Crear prototipos (de baja o alta fidelidad) para visualizar el sistema y validar los requisitos con los usuarios.
  - *Inspecciones y Revisiones de Requisitos:* Revisar formalmente los documentos de requisitos por un equipo de expertos para identificar errores, omisiones y ambigüedades. Utilizar listas de verificación (checklists) para guiar la revisión.
  - *Trazabilidad de Requisitos:* Establecer enlaces entre los requisitos, el diseño, el código, las pruebas y la documentación para asegurar que todos los requisitos estén implementados y probados. Herramientas de gestión de requisitos ayudan a mantener la trazabilidad.
- **Atributos de un Buen Requisito:**
  - *Claro:* Fácil de entender, sin ambigüedades.
  - *Completo:* Incluye toda la información necesaria.
  - *Consistente:* No contradice otros requisitos.
  - *Verificable:* Se puede probar o demostrar que se cumple.
  - *Trazable:* Se puede rastrear a lo largo del ciclo de vida.
  - *Priorizado:* Se asigna una prioridad para ayudar a la toma de decisiones.

2. Calidad en la Fase de Diseño:

El diseño define la arquitectura, los componentes y las interfaces del software. Las decisiones tomadas en esta fase tienen un impacto profundo en la calidad del software, incluyendo su rendimiento, mantenibilidad,

seguridad y escalabilidad.

- **Importancia de la Calidad del Diseño:**
  - *Mantenibilidad:* Un buen diseño facilita la modificación y extensión del software.
  - *Rendimiento:* Un diseño eficiente optimiza el uso de los recursos del sistema (memoria, CPU, red).
  - *Escalabilidad:* Un diseño escalable permite que el software maneje un número creciente de usuarios y datos.
  - *Seguridad:* Un diseño seguro protege el software contra vulnerabilidades y ataques.
- **Principios de Diseño Orientado a la Calidad:**
  - *Modularidad:* Dividir el sistema en módulos independientes y cohesivos, que realizan funciones específicas.
  - *Abstracción:* Ocultar los detalles de implementación de los módulos, exponiendo solo las interfaces necesarias.
  - *Encapsulamiento:* Proteger los datos internos de los módulos del acceso externo.
  - *Separación de Responsabilidades:* Asignar a cada módulo una única responsabilidad bien definida. (Principio de Responsabilidad Única - SRP).
  - *Acoplamiento Débil:* Minimizar las dependencias entre los módulos.
  - *Alta Cohesión:* Asegurar que los elementos dentro de un módulo estén relacionados entre sí.
  - *Reutilización:* Diseñar módulos que puedan ser reutilizados en diferentes partes del sistema o en otros proyectos.
  - *Patrones de Diseño:* Utilizar patrones de diseño probados y documentados para resolver problemas comunes de diseño.
- **Técnicas para Asegurar la Calidad del Diseño:**
  - *Revisiones de Diseño:* Revisar formalmente los documentos de diseño por un equipo de expertos para identificar problemas y proponer mejoras.
  - *Modelado UML:* Utilizar UML (Unified Modeling Language) para representar visualmente el diseño del sistema.
  - *Análisis de Arquitectura:* Evaluar la arquitectura del sistema para identificar riesgos y asegurar que cumple con los requisitos de calidad.
  - *Prototipado:* Crear prototipos para validar las decisiones de diseño y obtener retroalimentación temprana.
  - *Simulaciones:* Realizar simulaciones para evaluar el rendimiento y la escalabilidad del sistema.
  - *Uso de Herramientas CASE (Computer-Aided Software Engineering):* Herramientas que ayudan a automatizar tareas de diseño, como la generación de código a partir de modelos UML.

Ejemplos y Casos de Estudio:

- **Caso de Estudio: El desastre del Ariane 5:** Este ejemplo, ya mencionado en clases anteriores, puede ser revisitado desde la perspectiva de los requisitos. Un requisito del Ariane 4 (el cohete anterior) se reutilizó en el Ariane 5 sin una adecuada reevaluación del contexto. La falta de validación de este requisito en el nuevo sistema causó un desbordamiento aritmético y la destrucción del cohete. Este caso ilustra la importancia de una correcta gestión y validación de requisitos.
- **Ejemplo: Diseño de una API REST:** Considera el diseño de una API REST para un servicio de gestión de libros. Un mal diseño (por ejemplo, URLs no intuitivas, falta de versionado, manejo inconsistente de errores) puede llevar a una API difícil de usar y mantener. Un buen diseño (siguiendo principios RESTful, usando HTTP verbs correctamente, proporcionando documentación clara) resultará en una API más usable y robusta.

Problemas Prácticos y Ejercicios con Soluciones:

1. **Ejercicio: Análisis de Requisitos:** Dado el siguiente requisito: "El sistema debe ser rápido.", identifique por qué este requisito es deficiente y proponga una versión mejorada.
   - **Solución:** El requisito "El sistema debe ser rápido" es ambiguo y no verificable. Una versión mejorada podría ser: "El sistema debe responder a las consultas de búsqueda en menos de 2 segundos en el 95% de los casos, con una carga concurrente de 100 usuarios." Esto es claro, medible y verificable.

2. **Ejercicio: Principios de Diseño:** Considere un sistema para gestionar una biblioteca. ¿Cómo aplicaría los principios de modularidad y cohesión alta para diseñar este sistema?
    - **Solución:** Se podría dividir el sistema en módulos como: "Gestión de Libros", "Gestión de Usuarios", "Gestión de Préstamos" y "Gestión de Multas". Cada módulo se encarga de una tarea específica. Dentro del módulo "Gestión de Libros", todas las clases y funciones deberían estar relacionadas con la gestión de libros (añadir, eliminar, buscar, etc.). Esto asegura una alta cohesión.
3. **Ejercicio: Revisión de Diseño:** Dibuje un diagrama de clases simplificado para un sistema de comercio electrónico que incluya las clases Cliente, Producto, Carrito, y Pago. Identifique posibles problemas de diseño y proponga mejoras. Por ejemplo, ¿dónde colocaría la lógica para calcular el total del carrito de compras?
    - **Solución:** Un posible problema es colocar la lógica de cálculo del total del carrito directamente en la clase Carrito. Esto podría violar el principio de responsabilidad única (SRP). Una mejor solución sería crear una clase separada, como "CalculadorDeCarrito", que se encargue de calcular el total. Esto aumenta la modularidad y facilita la reutilización de la lógica de cálculo.

Materiales Complementarios Recomendados:

- **Libros:**
  - "Software Requirements" de Karl Wiegers y Joy Beatty: Un clásico sobre la ingeniería de requisitos.
  - "Design Patterns: Elements of Reusable Object-Oriented Software" de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (la "banda de los cuatro" - GoF): La biblia de los patrones de diseño.
- **Artículos:**
  - Buscar artículos sobre "calidad en el ciclo de vida del software" en IEEE Xplore o ACM Digital Library.
- **Sitios Web:**
  - SWEBOK (Software Engineering Body of Knowledge): https://www.computer.org/education/bodies-of-knowledge/software-engineering - Un recurso completo sobre ingeniería de software, incluyendo la gestión de requisitos y el diseño. "'