**Parcial 3 IS**

**Tema 6: Introducción al Diseño.**

**Introducción**

El diseño en ingeniería del software se basa en principios, conceptos y buenas prácticas orientadas al desarrollo de productos de software de calidad. Este conecta “el mundo de los humanos” con “el mundo de la tecnología”.

Este implica la búsqueda y evaluación de alternativas para converger en una solución que satisfaga los requisitos de la mejor manera posible. Aunque existen metodologías de diseño, lo esencial es comprender los principios fundamentales, ya que el diseño se ubica en el núcleo de la ingeniería del software y es independiente del modelo utilizado.

Tras analizar y modelar los requisitos, el diseño es la etapa previa a la construcción y organiza la información necesaria para un desarrollo exitoso.

**Conceptos de Diseño**

A lo largo de la historia de la ingeniería del software, varios conceptos han evolucionado y perdurado, sirviendo como base para aplicar métodos de diseño más avanzados. Estos permiten dividir el software en componentes, separar detalles de representaciones conceptuales y establecer criterios de calidad, ayudando a los desarrolladores a crear soluciones. Pressman definió los siguientes conceptos:

* **Abstracción:** Trabajar en distintos niveles de detalle para resolver un problema. A niveles bajos, más elementos específicos de la implementación, combinando terminología con técnica. Aplica a datos y procesos.
* **Arquitectura:** Define la estructura del software en unidades pequeñas y sus interrelaciones. Incluye patrones arquitectónicos para problemas recurrentes y utiliza modelos y lenguajes propios (ADL).
* **Patrones:** Soluciones prácticas y reutilizables a problemas comunes de diseño.
* **Separación de intereses (separation of concerns):** Divide un problema en partes independientes para facilitar su resolución. Reduce tiempo y esfuerzo.
* **Modularidad:** Derivada de la separación de intereses, permite que el software sea manejable intelectualmente. Previene diseños monolíticos, facilita comprensión y el mantenimiento.
* **Ocultación de la información:** Diseña módulos para que mantengan sus detalles internos privados, limitando el acceso a datos y algoritmos en cada módulo
* **Independencia funcional:** Cada módulo realiza tareas específicas y ofrece una interfaz simple, promoviendo reusabilidad y facilitando las pruebas. Se evalúa mediante cohesión y acoplamiento.
* **Refinamiento:** Proceso iterativo de diseño de arriba hacia abajo, que descompone funciones. Complementa la abstracción.
* **Refactorización:** Reorganización del código sin alterar su comportamiento, mejorando su comprensión y reduciendo costos.

La **cohesión** y el **acoplamiento** son dos conceptos fundamentales en el diseño de software. Ambos se refieren a la calidad de la estructura de los módulos de un sistema y son claves para crear software fácil de mantener, entender y escalar.

**Cohesión**

Mide si responsabilidades y funciones dentro de un módulo están bien relacionadas entre sí.

* **Alta cohesión:** Cuando todas sus funciones están estrechamente relacionadas y colaboran para cumplir una tarea específica. Facilita mantenimiento y reutilización del módulo.
* **Baja cohesión:** Cuando sus funciones son poco relacionadas o abarcan múltiples propósitos. Difícil de entender y mantener.

**Acoplamiento**

Mide el grado de interdependencia entre diferentes módulos de un sistema. Indica cuánto afecta un cambio en un módulo a otros.

* **Bajo acoplamiento:** Los módulos tienen una interacción mínima, se comunican a través de interfaces bien definidas. Reduce la posibilidad de que un cambio en un módulo afecte a otros.
* **Alto acoplamiento:** Los módulos están interconectados y dependen unos de otros. Los cambios son complicados, y pueden romper funcionalidades en otros módulos.

Lo ideal es buscar **alta cohesión** (módulos en tareas específicas) y **bajo acoplamiento** (módulos independientes con interacciones mínimas).

Por el contrario pueden aparecer varios problemas comunes en el diseño de software:

* **Alta cohesión + Alto acoplamiento:** Módulos están bien definidos pero interconectados.
* **Baja cohesión + Bajo acoplamiento:** Módulos independientes pero responsabilidades dispersas o poco relacionadas.

**Principios Básicos de Diseño**

Existen varios principios de diseño software para hacer código de calidad como:

* Don’t Repeat Yourself (**DRY**), evitar la duplicación de código.
* You ain’t gonna need it (**YAGNI**), no lo hagas hasta que no lo necesites.
* Keep it simple, stupid (**KISS**), mantenlo simple, estúpido.
* **SOLID**.

**Principios SOLID**

Son cinco reglas fundamentales para el diseño de clases en programación orientada a objetos. Promueven un código limpio, comprensible, mantenible y colaborativo. Ayudan a entender la importancia de patrones de diseño y arquitectura de software:

* El Principio de responsabilidad única (**S**ingle Responsibility Principle): Una clase debe encargarse de una sola tarea o responsabilidad en el sistema.
* El Principio Abierto-Cerrado (**O**pen-Closed Principle): Las clases deben estar abiertas a extensión pero cerradas a la modificación, poder añadir funcionalidades sin cambiar el código.
* El Principio de sustitución de Liskov (**L**iskov Substitution Principle): Las clases derivadas deben poder sustituir a sus clases base sin alterar el comportamiento esperado.
* El Principio de segregación de interfaz (**I**nterface Segregation Principle): Una clase no debe tener implementar interfaces que no usa. Es mejor dividir las interfaces en otras más específicas.
* El Principio de inversión de dependencia (**D**ependency Inversion Principle): Las clases de alto nivel no deben depender de las de bajo nivel, sino de abstracciones. Promoviendo sistemas flexibles y menos acoplados.

Su aplicación mejora la calidad del software y la colaboración entre desarrolladores.

**Niveles de Diseño**

El modelado del diseño se organiza en distintos niveles que guían el proceso de transformación de los requisitos en una solución técnica integral.

**Diseño de Datos**

Se centra en la organización, almacenamiento y acceso a los datos que utiliza el sistema. Es esencial para garantizar que los datos son consistentes, íntegros y fáciles de manejar. Elementos claves:

* Especificación de estructuras de datos: Identificación de archivos, tablas...
* Normalización de datos: Elimina redundancias y asegura consistencia.
* Selección de estructuras eficientes: Arrays, listas, árboles, grafos...

Un diseño de datos eficiente mejora el rendimiento del sistema, reduce errores y facilita futuras ampliaciones.

**Diseño Arquitectónico**

Define la estructura general del sistema en términos de módulos y sus interacciones. Se utiliza para organizar el software en componentes manejables y escalables. Elementos:

* Identificación de módulos principales: Dividir el sistema en componentes como interfaz, lógica de negocio y datos.
* Patrones arquitectónicos: MVC (Modelo-Vista-Controlador), microservicios, cliente-servidor.
* Diagramas arquitectónicos: Representaciones gráficas del diseño general.

Proporciona una visión global del sistema y asegura independientes y fácil integración de módulos.

**Diseño de Interfaz**

Cómo los módulos del sistema interactúan entre sí, con otros y con los usuarios. Contiene:

* Diseño de interfaces de usuario (UI): Interacción entre operadores y el sistema.
* Interfaces de programación (API): Especifica métodos y protocolos para que los módulos se comuniquen.
* Interacción con sistemas externos: Define los estándares y protocolos (REST, SOAP, etc.).

Un diseño de interfaz bueno garantiza la usabilidad, la interoperabilidad y la eficiencia.

**Diseño Procedimental o de las Funciones**

Define los algoritmos y procesos internos necesarios para que el software cumpla con los requisitos funcionales. Elementos clave:

* Diseño de algoritmos: Instrucciones paso a paso para realizar tareas específicas.
* Diagrama de flujo: Gráfica del flujo lógico de un procedimiento.
* Pseudocódigo: Describe los pasos de forma textual y estructurada.

Proporciona una base clara para implementar las funcionalidades de cada módulo, precisión y eficiencia.

**Patrones de Diseño**

Son soluciones generables y reutilizables a problemas comunes en el diseño de software.

Para que una solución sea considerada un patrón debe poseer unas características. Debe de haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Además, debe ser reutilizable, es aplicable a diferentes problemas de diseño en distintas circunstancias.

Por tanto, un patrón no es una solución, sino una plantilla para desarrollar una solución.

Los patrones de diseño se clasifican en tres categorías principales: creacionales, estructurales y de comportamiento.

**Patrones Creacionales**

Los patrones de diseño de creación abstraen el proceso de creación de instancias. Ayudan a hacer un sistema independiente de c´omo se crean, se componen y se representan sus objetos.

**Patrón Único o Singleton**

Garantiza que sólo exista una instancia de una clase, y proporciona un punto de acceso global a ella.

**Patrón Modelo-Vista-Controlador (MVC)**

Divide una aplicación interactiva en tres componentes: **Modelo, Vista y** **Controlador**, separando lógica de negocio y datos de la interfaz de usuario y eventos. Esta facilita la reutilización de código, la separación de conceptos y el mantenimiento del sistema.

* **Modelo:** Representa y gestiona los datos y la lógica de negocio, incluyendo privilegios de acceso. Responde a solicitudes del controlador y proporciona información requerida por las vistas.
* **Vista:** Presenta los datos del modelo en un formato interactivo. Solicita al modelo la información necesaria para mostrarla.
* **Controlador:** Actúa como intermediario entre vista y modelo. Responde a eventos del usuario, envía peticiones al modelo y actualiza la vista según sea necesario.

Organiza la funcionalidad en tres capas claras: procesamiento (modelo), entrada (controlador) y salida (vista), garantizando flexibilidad, escalabilidad y consistencia en el desarrollo y mantenimiento del software.

**Patrones Estructurales**

Los patrones de diseño estructurales se ocupan de cómo se combinan las clases y los objetos para formar estructuras más grandes.

**Patrón Compuesto (composite)**

Es un patrón estructural quepermite tratar de manera uniforme objetos individuales y composiciones de objetos. Este organiza los objetos en una estructura jerárquica en forma de árbol, los nodos pueden ser objetos simples o compuestos, facilitando la manipulación de jerarquías complejas.

**Patrones de Comportamiento**

Los patrones de diseño de comportamiento tienen que ver con algoritmos y asignación de responsabilidades a objetos. Estos describen patrones de clases, objetos y patrones de comunicación entre ellos.

**Patrón Iterador (iterator)**

Es un patrón de comportamiento que permite acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

Nos permite realizar estas operaciones trasladando la responsabilidad de acceder y recorrer el objeto lista a un objeto iterador. Esta clase definirá la interfaz de acceso a los elementos de la lista y será responsable de saber cuál es el elemento actual y cuáles se han recorrido.

**Patrón Observador (observer)**

Es un patrón de comportamiento que define una dependencia de uno a muchos entre objetos, cuando un objeto cambie de estado se notifiquen y actualicen todos los objetos que depende de él. También se llama dependiente (Dependents) y PublicadorSubscriptor (Publish-Subscribe).

**Tema 7: Introducción a las Pruebas de Software.**

**Introducción**

Los errores de software han provocado consecuencias significativas, desde menores hasta graves catástrofes. Estas resaltan la importancia de la verificación y validación en el desarrollo de sistemas.

**Técnicas de Verificación y validación (V&V)**

Durante el proceso de implementación, el programa se está desarrollando y debe ser comprobado para que satisfaga su especificación y funcionalidades esperadas por las partes de interés. V&V es el nombre dado a estos procesos de análisis y pruebas.

Estas están presentes en cada etapa del proceso de desarrollo del software, asegurando la calidad del producto durante su ciclo de vida.

**Verificación** - ¿Estamos construyendo el producto correctamente?

Tiene como objetivo garantizar que el software cumple con los requisitos establecidos en su especificación. Asegura que el producto se ajusta al diseño y que su implementación es técnicamente correcta.

**Validación** - ¿Estamos construyendo el producto correcto?

Evalúa si el software satisface las expectativas y necesidades del cliente, es decir, si cumple su propósito y es útil para los usuarios en su contexto operativo.

**Objetivos de la V & V**

* Evaluar y mejorar la calidad: Mejorar los productos durante el desarrollo y las modificaciones del software, para asegurar los estándares de calidad.
* Identificar y corregir fallos: Detectar los errores en el sistema y corregirlos para acercarse a un grado óptimo de perfección.
* Garantizar características clave del software: Asegurar consistencia, confiabilidad, utilidad, eficacia y el cumplimiento de los estándares definidos.
* Asegurar la utilidad del sistema: Confirmar que el sistema será funcional y adecuado para el entorno de trabajo.
* Establecer un nivel de confianza adecuado: El nivel de confianza necesario en el software depende de:
  + El propósito del sistema (critico o no crítico).
  + Las expectativas del usuario.
  + Las condiciones del entorno de mercado donde se utilizará.

El proceso de V & V es esencial para desarrollar software de alta calidad, minimizando riesgos y asegurando que los objetivos técnicos y de negocio sean alcanzados.

**Inspecciones del Software**

Son una técnica de validación estática usadas para revisar los diferentes artefactos generados durante el desarrollo del software para detectar errores, omisiones o anomalías.

Las inspecciones tienen las siguientes características principales:

* No requieren ejecución del sistema: Las inspecciones son una técnica estática, por ello no dependen de prototipos ejecutables.
* Aplicables a todo el ciclo de desarrollo: Pueden realizarse sobre cualquier artefacto.
* Enfoque económico y efectivo: Las inspecciones pueden detectar más del 60 % de los fallos con un coste menor al de pruebas dinámicas.
* Limitaciones:
  + No garantizan que el software sea operacional.
  + No evalúan niveles de fiabilidad ni aspectos no funcionales.
  + Útiles para detectar fallos en módulos, pero no a nivel de sistema.

Él Software Engineering Institute (SEI) considera las inspecciones como una de las prácticas esenciales en la gestión del proceso software. Las revisiones suelen realizarse en equipo, aprovechando el conocimiento y dominio del lenguaje para identificar errores.

**Pruebas del Software**

Son una técnica de validación dinámica que consiste en ejecutar el sistema o sus prototipos para comparar su comportamiento con el esperado, para así detectar fallos y evaluar su funcionalidad. Nos podemos encontrar dos tipos de pruebas:

1. Pruebas de validación
   * Verifican que el software satisface los requisitos.
   * Evalúan estadísticas de rendimiento y fiabilidad.
2. Pruebas de defectos:
   * Diseñadas para encontrar inconsistencias entre el programa y su especificación.
   * Buscan revelar errores más allá del uso esperado del sistema.

Las inspecciones y pruebas son actividades complementarias en el proceso de V&V. Aunque las pruebas suelen considerarse la principal técnica de validación, las inspecciones son esenciales para detectar errores de forma temprana y económica. Algunas metodologías, como eXtreme Programming, promueven técnicas como la programación por parejas (pair programming), que facilitan la inspección continua del código y una mejor calidad.

**Depuración del Software**

Es el proceso por el cual se localizan y corrigen errores identificados durante la verificación y validación. Sus características son:

* + Complejidad del proceso: Los errores no siempre se manifiestan donde se generaron, dificultando su localización.
  + Herramientas de depuración: Facilitan el análisis y solución de problemas.
  + Pruebas de regresión: Una vez corregido un error, se debe volver a probar el sistema para asegurarse de que:
    - La solución no ha introducido nuevos fallos.
    - Se han resuelto posibles errores relacionados.

La depuración es un ciclo iterativo que puede implicar correcciones y pruebas hasta alcanzar la estabilidad en el software.

**Pruebas del Software**

Son un conjunto de actividades en las que un sistema o componente es ejecutado bajo condiciones específicas, se registran los resultados y se hace una evaluación de los aspectos del sistema o componente.

En el contexto de la verificación de requisitos, las pruebas se utilizan para someter a examen el conjunto del sistema y verificar que satisface los requisitos y que está libre de errores.

Algunas de las definiciones que se utilizan en esta fase del ciclo de vida son según el estándar IEEE, 1990:

* + **Pruebas (test):** es una actividad en la que un sistema o uno de sus componentes se ejecuta en circunstancias especificadas, los resultados se observan y registran y se realiza una evaluación.
  + **Caso de prueba (test case):** un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo.
  + **Defecto (defect, fault, <<bug>>):** un defecto en el software, una definición de datos o un paso de procesamiento incorrectos en un programa. Se produce cuando se comete un error. Es una vista interna (lo ven los desarrolladores).
  + **Fallo (failure):** Incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas en los requisitos de rendimiento. Es un desvío respecto al comportamiento esperado, se puede producir en cualquier etapa. Es una vista externa, lo ven los usuarios. Es un resultado incorrecto.
  + **Error:** Es una acción humana que conduce a un resultado incorrecto.

**Objetivos de las Pruebas**

No se pueden probar todas las posibilidades de funcionamiento. El objetivo principal de estas es detectar defectos en el software. Por ello, descubrir un error se considera un éxito.

Existe el mito de que encontrar un defecto implica una mala práctica profesional o una razón para sentirse culpable pero:

* + Todo el mundo comete errores.
  + Descubrir un defecto contribuye a mejorar la calidad del software.

Para la realización de las pruebas se debe seguir esta filosofía o intención de trabajo:

* + Cada caso de prueba debe definir el resultado de salida esperado que se comparará con el obtenido.
  + El programador debe evitar probar sus propios programas, ya que desea demostrar que funcionan sin problemas.
  + Es normal que las situaciones que olvidó considerar al crear el programa queden de nuevo olvidadas en los casos de prueba.
  + Se debe inspeccionar el resultado de cada prueba, poder descubrir posibles defectos.
  + Al generar casos de prueba, se deben incluir datos de entrada válidos y no válidos.
  + Las pruebas deben centrarse en dos objetivos:
    - Probar si el software **no hace lo que debe hacer**.
    - Probar si el software hace lo que debe hacer, pero si **provoca efectos secundarios** **adversos**. Este suele ser ignorado.
  + No deben hacerse planes de prueba suponiendo que no hay defectos en los programas y, por ello, dedicando pocos recursos a las pruebas.
  + La experiencia indica que donde hay un defecto hay otros.
  + Las pruebas son una tarea tanto o más creativa que el desarrollo de software.
  + Las pruebas como una tarea destructiva y rutinaria.

Es interesante planificar y diseñar las pruebas de manera sistemática para detectar el máximo número de defectos con el mínimo consumo de tiempo y esfuerzo, por ello hay metodologías centradas en las pruebas.

**Tipos de pruebas**

Las pruebas de software se pueden clasificar en pruebas funcionales y no funcionales, según los aspectos del sistema que evalúan.

**Pruebas Funcionales**

Se centran en verificar que el software cumpla con los requisitos funcionales especificados, lo que el sistema “debe hacer”.

Sus características se basan en los requisitos del sistema, evaluar entradas, salidas, procesos y el comportamiento general del software, y ser generalmente pruebas de **caja negra** que no dependen de cómo se implementó la funcionalidad.

Tipos:

1. **Pruebas unitarias:** Validan componentes o funciones individuales del sistema.
2. **Pruebas de componentes:** Comprueban módulos específicos o unidades completas en conjunto.
3. **Pruebas de integración:** Verifican que los diferentes módulos interactúen correctamente.
4. **Pruebas del sistema:** Evalúan el comportamiento del sistema completo en su entorno integrado.
5. **Pruebas de aceptación (UAT):** Aseguran que el sistema satisface las expectativas de los usuarios.
6. **Pruebas de regresión:** Confirman que los cambios en el código no afectan negativamente las funcionalidades existentes.
7. **Pruebas de interfaz de usuario (UI Testing):** Evalúan la interacción del usuario con la interfaz gráfica y su comportamiento.
8. **Pruebas de versiones alfa y beta:**
   * **Alfa:** Realizadas internamente antes de liberar el producto.
   * **Beta:** Realizadas por usuarios finales en un entorno real para identificar problemas adicionales.

**Pruebas No Funcionales**

Evalúan aspectos no relacionados con funcionalidades específicas, sino con **atributos de calidad** del sistema, evalúan “como” el sistema realiza sus funciones.

Evalúan aspectos que complementan la funcionalidad del sistema, contribuyendo a mejorar la experiencia del usuario y la robustez del software. Pueden utilizar enfoques de caja negra o caja blanca, según los objetivos de las pruebas.

Tipos:

1. **Pruebas de rendimiento:**
   * **Pruebas de carga:** Evalúan el comportamiento del sistema bajo un número creciente de usuarios o transacciones.
   * **Pruebas de estrés:** Determinan los límites del sistema en condiciones extremas.
   * **Pruebas de estabilidad:** Analizan cómo funciona el sistema durante largos períodos de operación continua.
   * **Pruebas de escalabilidad:** Miden la capacidad del sistema para manejar incrementos en la carga.
2. **Pruebas de seguridad:**
   * Detectan vulnerabilidades y evalúan la capacidad del sistema para protegerse.
   * Incluyen pruebas de autenticación, cifrado y simulación de ataques (penetración).
3. **Pruebas de usabilidad:** Evalúan la facilidad de uso e intuitividad de la interfaz para el usuario.
4. **Pruebas de compatibilidad:** Verifican que el software funcione correctamente en diferentes dispositivos, sistemas operativos, navegadores…
5. **Pruebas de localización e internacionalización:** Aseguran que el software funcione correctamente en diferentes idiomas, regiones y formatos de datos…

**Desarrollo Guiado por Pruebas (Test-Driven Development, TDD)**

Es una metodología de desarrollo de software que pone las pruebas en el centro del proceso de creación. Este contrasta con metodologías más tradicionales y busca asegurar la calidad desde el inicio del desarrollo, fomentando un diseño claro y robusto.

* + En los enfoques tradicionales como la **metodología en cascada**, las pruebas se realizan en las fases finales del ciclo de vida del desarrollo, detectándose los errores tarde, lo que aumenta los costos de corrección.
  + En las **metodologías ágiles** y otros enfoques **iterativos**, las pruebas se integran en cada iteración, permitiendo una detección más temprana de errores. En la práctica, las pruebas se suelen relegar por limitaciones de tiempo o coste.
  + Por motivos como la presión para cumplir plazos o un coste, las pruebas se pueden considerar secundarias, se reducen o incluso se omiten por completo. Esto puede derivar en software con defectos o no alineado con los requisitos.

El desarrollo guiado por las pruebas es una manera de desarrollar software que intenta solucionar este problema aplicando dos reglas:

* + **Escribir las pruebas antes de implementar el código:** Estas no solo verifican el funcionamiento del sistema, sino que también actúan como especificaciones claras y concisas. Al escribirlas primero, se establece una guía del desarrollo de código.
  + **Implementar solo la funcionalidad necesaria para superar las pruebas:** El código escrito debe ser lo más simple y directo posible, diseñado para cumplir los criterios definidos en las pruebas. Esto ayuda a evitar el desarrollo innecesario o complejo.

El TDD ha traído una serie de ventajas en las pruebas de software:

* + **Diseño enfocado en requisitos:** Las pruebas actúan como un contrato, asegurando que el sistema cumple con los requisitos funcionales.
  + **Mayor calidad del código:** Al priorizar las pruebas, se reducen errores y mejora la mantenibilidad.
  + **Documentación implícita:** Las pruebas sirven como documentación del comportamiento esperado del sistema.
  + **Confianza en la evolución del sistema:** Permite realizar cambios con mayor seguridad, ya que las pruebas automatizadas detectan regresiones.

**Evolución: Del TDD al BDD**

Originalmente, el TDD se centraba en las pruebas unitarias, verificando pequeños bloques del códigos. Con el tiempo, se amplió, dando lugar a metodologías como el **Desarrollo Guiado por el Comportamiento** (Behavior-Driven Development, BDD).

El BDD se centra en describir el comportamiento del sistema en términos de escenarios comprensibles para desarrolladores e interesados en el negocio.

Sus características principales son:

* + Utiliza un lenguaje más cercano al negocio, como Gherkin, que permite describir escenarios en un formato de “Dado-Cuando-Entonces”.
  + Promueve una comunicación más clara entre equipos técnicos y no técnicos.
  + Es útil en el desarrollo de sistemas de información, donde el comportamiento esperado por usuarios y las respuestas del sistema están bien definidos.

**Flujo de trabajo del TDD**

El proceso del TDD sigue un ciclo iterativo conocido como Red-Green-Refactor:

1. **Red (Escribir una prueba que falle):** Se escribe una prueba que falla, ya que la funcionalidad aún no está implementada. Esto asegura que la prueba es válida.
2. **Green (Hacer que la prueba pase):** Se implementa el código mínimo necesario para superar la prueba.
3. **Refactor (Mejorar el código):** Con la prueba superada, se refactoriza el código para optimizarlo, manteniendo el cumplimiento de las pruebas.

Este ciclo garantiza que cada incremento de funcionalidad esté respaldado por pruebas automatizadas y código de calidad.

**Estrategia de Aplicación de las Pruebas**

Busca integrar el diseño de casos de prueba en una serie de pasos coordinados mediante la creación de niveles de prueba, cada uno con objetivos específicos. Además, permite coordinar las responsabilidades entre el personal de desarrollo, el departamento de garantía de calidad y el cliente, definiendo los roles de cada uno y cómo deben llevarse a cabo. Etapas:

* + **Pruebas a nivel de módulo (unidad):** Este nivel inicial se realiza por el equipo de desarrollo en su propio entorno, evaluando cada módulo de forma aislada.
  + **Pruebas de integración:** Una vez probados los módulos individuales, se integran siguiendo el diseño del software para verificar las interfaces y su funcionamiento conjunto.
  + **Pruebas funcionales o de validación:** Con el software ensamblado, se realizan pruebas que evalúan si el sistema cumple los requisitos funcionales, de rendimiento, seguridad...
  + **Pruebas del sistema:** Tras la validación, el software se integra con otros elementos del sistema para evaluar su funcionamiento global.
  + **Pruebas de aceptación:** Finalmente, el producto se somete a pruebas en el entorno operativo del usuario, que verifica si cumple con sus expectativas.

Esta estrategia garantiza una evaluación exhaustiva y progresiva del software desde su nivel más básico hasta su aceptación.

**Prueba Unidad**

Consiste en pruebas formales diseñadas para declarar que un módulo está terminado. Estas pruebas se diferencian de las informales que se realizan durante el desarrollo de los módulos. Uno o más módulos que cumplen las siguientes condiciones:

* + Todos los módulos pertenecen al mismo programa.
  + Al menos uno de ellos no se ha probado.
  + El conjunto de módulos es el objeto de un proceso de prueba.

Puede abarcar desde un único módulo, un grupo de ellos o incluso un programa.

Estas pruebas suelen ser realizadas por personal de desarrollo, pero es recomendable que no las haga el programador del módulo para evitar sesgos.

**Prueba de Integración**

Están orientadas a evaluar la interacción entre los diferentes componentes del software y su integración progresiva hasta formar el sistema completo que se entregará. Estas pruebas implican una secuencia ordenada, que comienza con los módulos individuales y finaliza en el sistema integrado.

El objetivo principal es verificar las interfaces entre los módulos, evaluando los flujos de datos y la comunicación entre ellos. El orden de integración afecta diversos aspectos clave, como:

* + La preparación de casos de prueba.
  + Las herramientas necesarias.
  + El orden de codificación y prueba de los módulos.
  + Los costos asociados a la depuración y a la preparación de casos de prueba.

**Tipos de Integración**

1. **Integración incremental:** Los módulos se integran progresivamente. Cada módulo nuevo se combina con los probados, incrementando el número de componentes hasta completar el sistema. Según el orden dentro de la jerarquía modular, se clasifica en:
   * **Ascendente:** Comienza con los módulos de nivel más bajo.
   * **Descendente:** Inicia con el módulo principal (raíz). Además, pueden aplicarse en **profundidad** (vertical) o en **anchura** (horizontal).
2. **Integración no incremental (Big-Bang):** Todos los módulos se prueban aisladamente y luego se integran de una vez, probando el sistema completo en una etapa. Este enfoque genera un aumento la complejidad, dificultando la depuración.

**Proceso de Integración Incremental Ascendente**

Los módulos de bajo nivel se agrupan para realizar funciones o subfunciones específicas. Es posible trabajar con los módulos de manera individual si resulta conveniente. Esto permite reducir el número de pasos para completar la integración.

Para cada grupo de módulos, se desarrolla un módulo impulsor o conductor (driver), diseñado para simular las llamadas a los módulos. Este facilita la introducción de datos de prueba a través de parámetros de entrada y recolección de resultados mediante parámetros de salida. El impulsor, se utiliza para probar el grupo correspondiente.

Tras las pruebas, impulsores se eliminan y se sustituyen por módulos de nivel superior en la jerarquía. Esto se repite, construyendo nuevos impulsores a medida que se avanza hacia los niveles superiores, hasta llegar a la raíz.

Ventajas:

* + Los defectos en partes inferiores se identifican de manera más rápida.
  + Los casos de prueba son más simples, porque los módulos inferiores suelen ser más específicos.
  + Los resultados de las pruebas son más fáciles de analizar, porque los módulos inferiores son los responsables principales de generar los datos.

Desventajas:

* + Es necesario desarrollar impulsores adicionales, lo que implica un esfuerzo de extra.
  + El sistema completo no puede evaluarse hasta que se integre el último módulo, retrasando validación global.

**Proceso de Integración Incremental Descendente**

El primer paso es probar la raíz o principal del sistema. En esta etapa los módulos subordinados se reemplazan por módulos ficticios (stubs).

Cuando la raíz ha sido probado y se confirma que está libre de defectos, se sustituye uno de los stubs por el módulo real implementado. Cuando se incorpora un módulo, se hacen las pruebas necesarias para verificar su funcionamiento.

Tras completar cada prueba, se reemplaza otro stub por su módulo real, repitiendo el proceso hasta que todos los módulos hayan sido sustituidos. Se recomienda repetir algunos casos de prueba anteriores para asegurar que hoy nuevos errores.

* + **En Profundidad:** completando ramas del árbol (A, B, E, C, F, G, D).
  + **En Anchura:** completando niveles de jerarquía (A, B, C, D, E, F, G).

Ventajas:

* + Permite detectar defectos en los niveles superiores del sistema desde las primeras etapas de integración
  + Facilita la gestión de los casos de prueba al incluir funciones de entrada y salida claras.
  + Proporciona una visión preliminar de la estructura del programa, lo que ayuda a identificar posibles problemas estructurales.

Desventajas:

* + Los módulos ficticios (stubs) pueden ser complejos y costosos.
  + Es complicado gestionar casos de prueba antes de integrarlo completamente.
  + A veces, no es posible crear casos de prueba adecuados por falta de información detallada sobre los módulos inferiores.
  + Los resultados pueden ser difíciles de interpretar y su funcionalidad principal depende de los módulos inferiores, que aún no están integrados.
  + Puede retrasar la finalización de las pruebas al parecer que el programa funciona correctamente, ocultando defectos en niveles inferiores.

**Proceso de Integración No Incremental**

Para módulos individualmente en un enfoque no incremental, se requieren los siguientes elementos:

* + Un **módulo impulsor**, encargado de enviar los datos de prueba al módulo que se evalúa y de mostrar los resultados.
  + Uno o más **módulos ficticios** (stubs) que simulan las funciones de los módulos llamados por el módulo en prueba.

Tras realizar las pruebas individuales, todos se ensamblan simultáneamente y se someten a pruebas en conjunto.

**Prueba del Sistema**

Consiste en evaluar un sistema integrado para verificar aspectos clave:

* + Cumplimiento de requisitos funcionales: Asegura que el producto de software cumpla con todas las especificaciones en un sistema completo.
  + Funcionamiento y rendimiento: Evalúa las interfaces para garantizar su correcto desempeño.
  + Documentación del usuario: Verifica que la documentación proporcionada sea adecuada y funcional para los usuarios.
  + Condiciones extremas: Examina cómo el sistema opera bajo condiciones límite y de sobrecarga para garantizar su robustez.

El diseño de los casos de prueba para la evaluación del sistema se basa en los siguientes enfoques:

* + Requisitos del sistema: Utiliza técnicas de caja negra para evaluar las especificaciones funcionales.
  + Rendimiento y capacidad: Incluye pruebas específicas para evaluar el rendimiento del sistema (conocidas como pruebas de estrés o stress testing).
  + Diseño de alto nivel: Emplea técnicas de caja blanca para analizar los flujos de datos de alto nivel, tomando como referencia los diagramas de flujo de datos.

Este proceso asegura que el sistema cumple las expectativas funcionales y operativas en un entorno realista.

**Prueba de Aceptación**

Es un proceso planificado y formalmente organizado que tiene como objetivo verificar si el sistema cumple con los requisitos de aceptación. Características principales:

* + La participación activa del usuario en el proceso de evaluación.
  + Su enfoque se centra en la verificación de los requisitos de usuario.
  + Se considera como la fase final del proceso, diseñada para generar confianza en que el producto es adecuado para su uso real.

Para asegurar el éxito de la prueba de aceptación, se sugiere:

* + Establecer criterios de aceptación claros y aprobados por el usuario.
  + Validar el sistema, la documentación y los procedimientos de uso asociados, lo que puede incluir auditorías.
  + Realizar las pruebas en el entorno real en el que se usara el sistema, involucrando al personal que lo operará.

En cuanto a las modalidades de prueba:

* + Si el sistema es un encargo específico, las pruebas se realizan bajo control de la organización desarrolladora en el entorno de trabajo real, asistiendo al usuario en su ejecución (pruebas Alfa).
  + Para productos de interés general, se entregan versiones preliminares a usuarios de confianza que las prueban sin control directo. Estos informan sobre su experiencia y retroalimentan sobre la aplicación (pruebas Beta).

**Técnicas de Diseño de Casos de Prueba**

Se ve condicionado por la imposibilidad de realizar una prueba exhaustiva del software. El objetivo es lograr una confianza aceptable en que se detectarán los defectos existentes, sin que implique un consumo excesivo de recursos.

Al diseñar los casos de prueba se busca seleccionar algunos que se consideren representativos de otros posibles. Permitiendo obtener una cobertura adecuada del software sin probar todas las combinaciones. Si al ejecutar estos casos no se detectan defectos, podemos tener un nivel de confianza en que el programa no presenta fallos importantes.

Sin embargo, la dificultad principal es saber elegir los casos de prueba adecuados para cada situación porque no todos los escenarios pueden cubrirse de manera exhaustiva.

Existen varias aproximaciones para diseñar casos de prueba:

* + El **enfoque basado en requisitos**: los casos se diseñan para probar los requisitos del sistema. Se usan las pruebas del sistema, ya que los requisitos se implementan por varios componentes. Para cada requisito se establece un caso de prueba para comprobar que se satisface.
  + El **enfoque funcional** o de **caja negra**: se identifican particiones de entrada y salida y se diseñan pruebas para que el sistema ejecute todas las entradas y genere sus salidas. Las particiones son grupos de datos con características comunes.
  + El **enfoque estructural** o de **caja blanca**: se utiliza el conocimiento de la estructura interna para diseñar pruebas que ejecutan las partes del programa. Al probar un programa, cada sentencia debe ejecutarse al menos una vez.
  + El **enfoque aleatorio** consiste en utilizar modelos que representen las posibles entradas al programa para crear a partir de ellos los casos de prueba.

**Pruebas Basadas en Requisitos**

Un principio general de la ingeniería de requisitos es que los requisitos deberían poder probarse. Se considera cada requisito y se deriva un conjunto de pruebas para validarlos (prueba de validación).

**Pruebas Funcionales o de Caja Negra**

Se enfocan en evaluar las funciones del sistema, considerando entradas y salidas. Dado que no es viable probar todas las combinaciones posibles, es esencial seleccionar casos de prueba efectivos y estratégicos.

Para elegir un buen caso se pueden seguir los siguientes criterios:

* + Optimización: Diseñar casos que reduzcan la cantidad necesaria, abarcando mayor cantidad de combinaciones de entrada en una prueba.
  + Cobertura Amplia: Representar un rango extenso de escenarios similares para detectar defectos en las entradas.

**Prueba de Partición o Clases de Equivalencia**

Se basan en crear casos que cubran grupos de entradas representativas, optimizando el esfuerzo de prueba.

* + Cada caso debe abarcar el máximo número de entradas.
  + El dominio de valores de entrada se divide en un número finito de clases, para probar un valor representativo de una clase permitiendo suponer que el resultado será el mismo que al probar cualquier otro valor en esa clase.

El método utilizado en estas pruebas para diseñar casos de prueba consiste en identificar las clases de equivalencia y crear los casos de prueba correspondientes.

**Pasos para identificar clases de equivalencia**

1. Identificar las condiciones de entrada: Reconocer las restricciones de formato o contenido de los datos de entrada.
2. Definir las clases de equivalencia: Estas pueden ser:
   * Datos válidos: Representan entradas aceptadas.
   * Datos no válidos o erróneos: Entradas que el sistema debe rechazar.
3. Asignar identificadores únicos a cada clase de equivalencia.
4. Crear casos de prueba:
   * Incluir todas las clases válidas posibles en una ejecución.
   * Diseñar un caso de prueba por cada clase no válida para aislar y detectar defectos específicos.

**Reglas de identificación**

Existen varias reglas para identificar el tipo de las clases de equivalencia:

1. **Rango de valores:** Si los datos de entrada tienen un rango especificado, se crean:
   * Una clase válida
   * Dos clases no válidas (una por cada extremo del rango).
2. **Valores limitados o consecutivos:** Si se especifica un número fijo o finito de valores consecutivos, se crean:
   * Una clase válida
   * Dos clases no válidas.
3. **Condiciones booleanas:** Si existe una condición tipo “debe ser” o booleana, se identifican:
   * Una clase válida.
   * Una clase no válida.
4. **Conjunto de valores admitidos:** Si los datos son un conjunto de valores específicos tratados de forma diferente, se crean:
   * Una clase válida por cada valor del conjunto.
   * Una clase no válida para cualquier valor fuera del conjunto. 5.
5. **Sospechas de tratamiento desigual:** Si ciertos elementos de una clase podrían ser tratados de manera diferente al resto, divídalos en subclases.

Este enfoque permite diseñar pruebas más efectivas, asegurando una cobertura amplia sin incrementar excesivamente la cantidad de casos de prueba.

**Pruebas Estructurales o de Caja Blanca**

Se enfocan en la estructura interna del programa, analizando sus caminos de ejecución para garantizar su cobertura.

Los casos de prueba se diseñan considerando caminos importantes, para detectar defectos de manera efectiva. Este enfoque es crucial, ya que el número de secuencias posibles puede ser inmenso.

Para optimizar el proceso y cubrir los caminos relevantes, se diseñan pruebas que cumplan con los siguientes criterios:

* + Garantizar que todos los caminos independientes en cada módulo sean ejecutados al menos una vez.
  + Asegurar la ejecución de todas las decisiones lógicas, considerando tanto las caras verdaderas y las falsas.
  + Probar todos los bucles, enfocándose en sus límites.
  + Verificar las estructuras de datos internas para confirmar su validez.

Estas pruebas suelen apoyarse en **Diagramas de Flujo de Control** como base de diseño.

Estos diagramas se realizan siguiendo los siguientes pasos:

1. **Identificación de condiciones:** Marcar las condiciones presentes en las decisiones del código, como IF-THEN, CASE-OF, WHILE, y UNTIL.
2. **Agrupación de sentencias:** Organizar las sentencias entre condiciones, siguiendo las estructuras básicas del programa.
3. **Numeración de elementos:** Asignar identificadores únicos a las condiciones y grupos de sentencias.
4. **Reordenación de decisiones multicondicionales:** En decisiones con múltiples condiciones, reorganizarlas en un orden decreciente de restricción para simplificar el diseño de casos de prueba.

**Prueba del Camino Básico**

La cobertura de caminos es el proceso de ejecutar una secuencia de sentencias desde inicio hasta final del programa, constituyendo uno de los criterios fundamentales en las pruebas de caja blanca. Pese a ello probar todos los caminos de un programa puede resultar impracticable.

Para abordar esto, se utilizan caminos básicos, que recorren cada bucle como máximo una vez. McCabe definió la **Complejidad Ciclomática V(G)** como una métrica para medir los caminos independientes.

La complejidad ciclomática indica el número máximo de pruebas necesarias para que cada sentencia del programa sea ejecutada al menos una vez.

Existen una serie de conceptos clave:

* + **Camino independiente:** Es un recorrido que introduce al menos una sentencia o condición nueva. En un grafo de flujo de control, un camino independiente pasa por al menos una arista que no haya sido recorrida.
  + **Nodo decisión:** Contiene una condición y tiene dos o más salidas (aristas).

**Cálculo de la Complejidad Ciclomática**

Se puede calcular de 3 formas diferentes:

* + V (G) = a-n+2 (a el número de arcos del grafo y n el número de nodos).
  + V (G) = r (r el número de regiones cerradas del grafo).
  + V (G) = c + 1 (c el número de nodos decisión).

El criterio de prueba de McCabe establece que deben generarse tantos casos de prueba como caminos independientes, calculados mediante V(G).

* + V(G) representa el número mínimo de casos de prueba necesarios.
  + Si V (G) > 10, aumenta la probabilidad de defectos, sugiriendo la necesidad de dividir el módulo en partes más pequeñas para mejorar su mantenibilidad.

**Procedimiento para Derivar Casos de Prueba**

1. **Construir el grafo de flujo de control**: Representar gráficamente las sentencias y condiciones del programa.
2. **Calcular la Complejidad Ciclomática V(G):** Determinar el número de caminos independientes.
3. **Identificar los caminos independientes:** Localizar las rutas únicas que deben ser probadas.
4. **Diseñar casos de prueba:** Preparar pruebas específicas para ejecutar cada camino básico identificado.

**Matriz de conexiones**

Se usa para automatizar la determinación del conjunto básico de caminos. Esta se construye según el número de nodos del grafo y sus entradas representan las conexiones entre los nodos (aristas).

Para calcular la complejidad ciclomática con la matriz:

1. Añadir una columna a la matriz con la suma de cada fila, restando 1 de cada total.
2. La complejidad ciclomática se obtiene sumando los valores de esta columna y añadiendo 1 al resultado.

El cálculo de esta matriz puede tener usos adicionales:

* + Probabilidad de ejecución de una arista: Permite calcular la probabilidad de que una arista sea recorrida durante la ejecución del programa.
  + Recursos requeridos: Se puede utilizar para estimar los recursos, como tiempo y memoria, necesarios para la ejecución de cada enlace o arista.

Cuando se ha calculado la complejidad ciclomática, se diseñan los casos de prueba, que deben cubrir cada uno de los caminos del conjunto básico.

**Tema 8: Despliegue, Evolución y Mantenimiento del Software.**

**Introducción**

Cuando el software se construye y pasa las pruebas, se ha terminado el trabajo. Es el punto de partida para una serie de actividades cruciales que garantizarán la utilidad y la sostenibilidad del software en el tiempo. Estas actividades, que forman parte del ciclo de vida del software, se enfocan en:

1. **La puesta a disposición para los clientes o usuarios**

El software debe ser preparado para su uso real. Esto implica:

* + La instalación o despliegue en un entorno adecuado.
  + La configuración inicial para satisfacer las necesidades.
  + La capacitación de los usuarios finales para su correcta utilización.

Este asegura que el software esté disponible y sea utilizable de manera efectiva.

1. **La incorporación de nuevas funcionalidades y la mejora de las existentes**

Cuando los usuarios interactúan con el software, surgen necesidades. Estas pueden incluir:

* + Nuevas características para aumentar el alcance o el valor del sistema.
  + Mejoras en las funcionalidades existentes para optimizarlas o adaptarlas a mejores prácticas.

Estas son esenciales para mantener la relevancia del software.

1. **La corrección de errores no detectados durante las pruebas**

Es posible que algunos defectos no se detecten. Cuando el software este en uso, estos errores pueden emerger debido a:

* + Casos de uso imprevistos.
  + Datos específicos del entorno real.
  + Limitaciones en las pruebas realizadas previamente.

La corrección de estos errores debe ser continua y crítica para garantizar la confiabilidad del sistema.

1. **La mejora en el modo de funcionamiento del sistema en nuevos entornos**

El entorno operativo de un software puede cambiar con el tiempo. El software debe ser ajustado para:

* + Asegurar la compatibilidad con nuevas condiciones.
  + Aprovechar nuevas capacidades tecnológicas que mejoren el rendimiento.

1. **La readaptación de la concepción del sistema ante cambios en el negocio**

Las empresas evolucionan constantemente, y el software debe de adaptarse a:

* + Nuevos procesos de negocio o cambios en los existentes.
  + Regulaciones o normativas actualizadas.
  + Cambios en las expectativas del mercado o los usuarios.

Esta readaptación asegura que el software siga cumpliendo con los objetivos de la organización.

Estas actividades forman parte del ciclo de vida del software. No son eventos aislados, sino que constituyen fases esenciales dentro del ciclo de vida del software.

Para abordarlas de manera efectiva, se requiere un enfoque estructurado que incluye:

* + **Metodologías tradicionales o ágiles que faciliten la labor de los ingenieros** y permiten abordar proyectos según necesidades, con iteraciones rápidas, procesos estructurados o integración continua agilizando entregas y mejorando la colaboración.
  + **Herramientas de soporte para la gestión y automatización de tareas** del sistema que optimizan el mantenimiento, reducen esfuerzos manuales y aseguran un desempeño eficiente del software.

**Proceso de Despliegue**

Consiste en poner el software o sus actualizaciones a disposición de los usuarios para su uso en un entorno real. Este abarca una serie de operaciones clave, como la preparación y lanzamiento de versiones, configuración e instalación del sistema, y monitoreo del rendimiento tras la puesta en marcha. Actualmente combinan procesos manuales y automáticos, facilitando la distribución eficiente de actualizaciones y parches.

Las operaciones más comunes en el despliegue del software son:

* + **Preparación y lanzamiento de versiones:** Compilación de los cambios aprobados en una nueva versión lista para ser distribuida.
  + **Configuración e instalación:** Adaptación del software al entorno específico en el que será usado, asegurando compatibilidad y funcionamiento.
  + **Monitoreo del rendimiento:** Supervisión continua para identificar posibles problemas y garantizar la estabilidad del sistema.

El despliegue suele seguir tres pasos fundamentales:

1. **Recibir peticiones:** Identificar cambios o mejoras requeridas.
2. **Elegir e implementar peticiones:** Evaluar, aceptar e integrar las peticiones al sistema, considerando su impacto y viabilidad.
3. **Hacer los cambios disponibles:** Liberar una nueva versión del software que incluya los cambios implementados.

Puede variar según las características del sistema y su entorno operativo. Cada organización debe definir un flujo de trabajo claro, estableciendo actividades, prácticas y herramientas adecuadas.

Cuando se incorpora una nueva funcionalidad, el software pasa por tres estados:

1. **Preparación:** Se recopila el código y recursos necesarios, como configuraciones y librerías, para su integración.
2. **Prueba:** El código preparado es evaluado en un entorno específico de pruebas (staging) para evitar la introducción de errores, utilizando casos.
3. **Despliegue:** El código es integrado en el entorno de producción, quedando disponible para los usuarios junto con las funcionalidades previas.

Tener un proceso estructurado y riguroso tiene múltiples beneficios. Primero facilita la incorporación segura de nuevas funcionalidades que responden a las necesidades del negocio, garantizando un impacto positivo en la evolución del sistema. Este también mejora las operaciones de la organización al optimizar procesos y eficiencia.

Además, un proceso controlado permite monitorear cada etapa, recolectar datos y utilizarlos para mejorar el despliegue.

Cuando el despliegue está automatizado, se aceleran las pruebas, y los cambios se hacen transparentes y accesibles para los usuarios, reduciendo la incertidumbre y aumentando la confianza en el sistema.

**Despliegue Continuo (CD) y su Integración con CI**

Muchas organizaciones usan el enfoque de **despliegue continuo** (CD), un sistema totalmente automatizado para desplegar nuevas funcionalidades con alta frecuencia. Este se combina con la **integración continua** (CI), donde los cambios en el código se integran en la rama principal. El modelo CI/CD asegura una entrega fluida, rápida y confiable de nuevas versiones del software.

Para maximizar la efectividad, se deben implementar las siguientes prácticas:

* + **Mantener un repositorio único de código:** Facilita colaboración y evita duplicación de esfuerzos.
  + **Automatizar la construcción (build):** Asegura que las versiones del software se generen de manera consistente y confiable.
  + **Pruebas automáticas:** Permite detectar errores de tempranos y mantener la calidad del código.
  + **Commits diarios a la rama principal:** Evita conflictos complejos y mantiene un estado funcional del sistema.
  + **Integración correcta de commits:** Cada cambio debe probarse y verificarse para garantizar su compatibilidad con la versión.
  + **Bugs corregidos con casos de prueba asociados:** Los errores corregidos deben ir acompañados de pruebas que aseguren que no vuelva a ocurrir.
  + **Construcciones rápidas:** Mantener tiempos cortos de construcción agiliza los ciclos de prueba y despliegue.
  + **Pruebas en un entorno específico (staging):** Un entorno intermedio permite validar los cambios antes de liberarlos en producción.
  + **Acceso rápido y fácil a nuevas funcionalidades:** Se deben poder acceder fácilmente a las nuevas características para evaluar su impacto.
  + **Transparencia en errores de construcción:** Si la construcción falla, todos los involucrados se puede identificar rápidamente el problema y tomar medidas.

**Evolución del Software**

Es un proceso inevitable y necesario para que los sistemas sigan siendo útiles y relevantes en un entorno cambiante. Los cambios y la evolución forman parte de la ingeniería del software, y reflejan sus necesidad de adaptarse a nuevas condiciones.

Los cambios en el software pueden ser debidos a:

* + **Nuevas situaciones en el negocio:** Cambios en procesos, modelos de negocio o estrategias que exigen ajustes.
  + **Expectativas de los clientes:** Nuevos requisitos o funcionalidades solicitadas para satisfacer demandas.
  + **Adaptación tecnológica:** Necesidad de operar en nuevos entornos de hardware o software.
  + **Mejoras de rendimiento y aspectos no funcionales:** Incrementar las características técnicas del sistema.

La frecuencia de lanzamiento de versiones ha aumentado en las últimas décadas debido a enfoques como CI/CD. Esto es relevante a nivel empresarial, donde los cambios en un componente pueden tener un impacto en otros sistemas interconectados.

El proceso de evolución del software puede variar según el tipo de sistema. Generalmente, comienza con la identificación y propuesta de cambios y culmina con la generación de una nueva versión del sistema.

**El Proceso de Evolución del Software**

Implica analizar cómo un cambio afecta al sistema y su entorno operativo. Este puede llevarse a cabo por la empresa de desarrolló el software, el cliente, o por terceros. Su formalidad puede variar según el tipo de sistema y su criticidad. Etapas:

1. **Identificación y propuesta de cambios:** Cualquier interesado puede proponer cambios al sistema. Estas deben ser evaluadas para determinar su viabilidad.
2. **Análisis de impacto:** Se debe identificar qué partes del sistema seran afectadas por los cambios, evaluando tanto coste asociado como el impacto en el funcionamiento del software y sus dependencias.
3. **Planificación de la nueva versión:** Incluir los cambios , como la corrección de errores, adaptaciones tecnológicas o nuevas funcionalidades, y establecer un cronograma para implementarlo y pruebas.
4. **Implementación y validación:** Realizar cambios en el código y validar una nueva versión para asegurar que cumpla con los requisitos.
5. **Entrega del sistema:** Se entrega la nueva versión del sistema a los usuarios o al entorno de producción.

En situaciones críticas, el proceso de evolución puede simplificarse para completar los cambios con mayor rapidez.

**Mantenimiento del Software**

Abarca todos los procesos necesarios para realizar cambios después de la entrega (delivery). Es especialmente relevante en sistemas desarrollados a medida para clientes, donde los equipos encargados del desarrollo y del mantenimiento suelen ser diferentes.

Los cambios en el software pueden variar desde simples correcciones hasta modificaciones más profundas como rediseños o la incorporación de nuevos requisitos. Entre estas actividades de mantenimiento se incluyen:

* + **Reparación de fallos:** Corregir errores y vulnerabilidades que afectan al funcionamiento del sistema.
  + **Adaptación a nuevas plataformas o entornos de operación:** Ajustar el software para que sea compatible con nuevas tecnologías o sistemas operativos.
  + **Incorporación de nuevas funcionalidades o soporte para nuevos requisitos:** Agregar nuevas características para satisfacer necesidades emergentes del cliente o mejorar el rendimiento.

**Tipos de Mantenimiento**

* + **Mantenimiento Correctivo:** Se enfoca en corregir errores del sistema después del despliegue (bugs o vulnerabilidades) que afectan su funcionamiento. Es esencial para asegurar que el sistema continúe operando correctamente.
  + **Mantenimiento Adaptativo:** Consiste en modificar el software para que ser compatible con nuevos entornos, plataformas o tecnologías. Permite que el software siga siendo útil y funcional en un entorno cambiante.
  + **Mantenimiento Perfectivo:** Se basa en mejorar el rendimiento del sistema o agregar nuevas funcionalidades en respuesta a cambios en los requisitos del negocio o a demanda de características. Esto incluye la optimización del código, mejora de la interfaz de usuario o la incorporación de funciones.
  + **Mantenimiento Preventivo:** Busca anticipar problemas y evitar errores futuros. Esto implica refactorizar el código, mejora de la documentación y la revisión de la arquitectura del sistema para mantenerlo eficiente y sostenible.

**Costes del Mantenimiento**

La reparación de fallos no suele ser la tarea de mantenimiento más costosa, la adición de nuevas funcionalidades si necesita más inversión. Estos cambios requieren un análisis detallado y a veces, un rediseño del sistema. El equipo de mantenimiento debe tener buen entendimiento para implementar cambios sin comprometer su estabilidad. A veces, el equipo de desarrollo no estuvo involucrado en la creación de un código fácilmente mantenible, esto puede hacer que las modificaciones sean más complicadas y costosas.

**Desafíos del Mantenimiento del Software**

A veces se percibe como tareas menos relevantes, lo que puede llevar a que las actividades sean delegadas a personal con menos experiencia. Trabajar con tecnologías o lenguajes obsoletos puede complicar el proceso. Si el sistema crece y se agregan funcionalidades, su estructura se vuelve compleja y la implementación de nuevos cambios es más difícil. La documentación tiende a volverse inconsistente o desactualizada, dificultando la comprensión y modificación del sistema a largo plazo.

El mantenimiento del software, presenta retos que requieren un enfoque planificado y bien gestionado para garantizar que el sistema funcione de manera eficiente y satisfaga las necesidades del negocio.

**Predicción del Mantenimiento**

En la gestión de software, anticiparse a problemas es clave para evitar sorpresas costosas e inesperadas. Por ello se deben predecir los posibles cambios que requerirá un sistema, identificar sus partes más difíciles de mantener y estimar los costes de mantenimiento en un largo periodo determinado. Estas predicciones están interrelacionadas:

* + La decisión de aceptar o rechazar un cambio depende, de la facilidad con que los componentes del sistema afectados puedan ser mantenidos.
  + La implementación de cambios degrada la estructura del sistema, reduciendo su mantenibilidad a largo plazo.
  + Los costes de mantenimiento están relacionados con el número de cambios, mientras que los costes de implementación dependen de la mantenibilidad de los componentes.

**Proceso de Reingeniería**

Es un proceso aplicado a sistemas heredados (legacy systems), que llevan mucho tiempo en funcionamiento y cuya actualización puede ser compleja. Este enfoque es útil cuando realizar cambios sobre el sistema no es viable o cuando reemplazarlo por uno nuevo es poco práctico o arriesgado.

Beneficios de un proceso de reingeniería andes de crear uno nuevo:

* + **Reducción del riesgo:** En sistemas críticos para el negocio, un retraso en la implementación de un nuevo sistema puede resultar en pérdidas significativas. Modernizar un sistema existente minimiza estas incertidumbres.
  + **Reducción de costes:** Al reutilizar y mejorar componentes del sistema, se evitan gastos asociados al desarrollo completo de un sistema nuevo.

El propósito de la reingeniería es obtener una versión mejorada y reestructurada de un sistema existente, optimizando su funcionamiento, adaptabilidad y sostenibilidad. Este no siempre requiere aplicar todas las actividades disponibles; la selección depende de las necesidades del sistema y del negocio.

Las actividades del proceso de reingeniería son:

1. **Traducción de código:** Consiste en migrar el código fuente a una versión más moderna del lenguaje de programación o incluso a un lenguaje diferente, mejor adaptado a las necesidades.
2. **Ingeniería inversa:** Analizar el código para comprender su estructura, diseño y funcionalidades, facilitando su modificación y mejora, especialmente cuando la documentación original está desactualizada.
3. **Mejora de la estructura del programa:** Reorganizar el código mediante técnicas como la refactorización, para hacerlo más claro, eficiente y mantenible.
4. **Modularización del programa:** Reestructurar el sistema a nivel arquitectónico, dividiendo el software en módulos más independientes y cohesivos, para mejorar flexibilidad y escalabilidad.
5. **Reingeniería de datos:** Redefinir tablas, esquemas y estructuras de las bases de datos, para optimizar su desempeño y garantizar compatibilidad con tecnologías actuales.