INGENIERIA DE SOFTWARE 2 – RESUMEN TEORIA 2

DISEÑO DE SOFTWARE

- Representación significativa de ingeniería de algo que se va a construir.
- Es el proceso creativo de transformación del problema en una solución.
- Es el núcleo técnico de la ingeniería de software.
    - Una vez que se analizan y especifican los requisitos del software, el diseño es la primera actividad técnica a realizar.
- Es independiente del modelo de proceso que se utilice.
- El diseño se centra en cuatro áreas importantes:
    - Datos, Arquitecturas, Interfaces y Componentes.

- Diseño de datos: transforma el modelo del dominio, obtenido del análisis, en estructura de datos, objetos de datos, relaciones, etc.
- Diseño arquitectónico: define la relación entre los elementos estructurales más importantes del software, los estilos arquitectónicos, patrones de diseño, etc., para lograr los requisitos del sistema. La información para realizar el diseño puede derivarse de la especificación, del modelo de análisis y de la interacción de los subsistemas definidos.
- Diseño a nivel componentes: Transforma los elementos estructurales de la arquitectura de software en una descripción procedimental de los componentes del software. La información obtenida de los modelos basados en clases, modelos de flujos, de comportamiento (DTE) sirven como base.
- Diseño de interface: Describe la forma de comunicación dentro del mismo sistema, con otros sistemas, y con las personas. Una interface implica flujo de información (datos o control) y comportamiento.

- El diseño es la etapa en la que se fomentará la calidad.
- Proporciona las representaciones del software susceptibles de evaluar respecto de la calidad.
- Sin diseño se corre el riesgo de construir un sistema inestable, el cual fallará cuando se realicen cambios pequeños, será difícil de probar.

- Características para la evaluación de un diseño:
  El diseño deberá implementar todos los requisitos explícitos del modelo de análisis, y deberá ajustarse a todos los requisitos implícitos que desea el cliente.
  Deberá ser una guía legible y comprensible para aquellos que generan código y para aquellos que comprueban y consecuentemente, dan soporte al software.
  Deberá proporcionar una imagen completa del software, enfrentándose a los dominios de comportamiento funcionales y de datos desde una perspectiva de implementación.

- Criterios técnicos para un buen diseño:
    1. Deberá presentar una estructura arquitectónica que: Se haya creado mediante patrones de diseño reconocibles. Que esté formado por componentes que exhiban características de buen diseño. Se implemente en forma evolutiva.
    2. Deberá ser modular, el software deberá dividirse lógicamente en elementos que realicen funciones y sub-funciones específicas.
    3. Deberá contener distintas representaciones de datos, arquitectura, interfaces y componentes (módulos).
    4. Deberá conducir a estructuras de datos adecuadas para los objetos que se van a implementar y que procedan de patrones de datos reconocibles.
    5. Deberá conducir a componentes que presenten características funcionales independientes.
    6. Deberá conducir a interfaces que reduzcan la complejidad de las conexiones entre los módulos y con el entorno externo.
    7. Deberá derivarse mediante un método repetitivo y controlado por la información obtenida durante el análisis de los requisitos del software.
    8. Debe representarse por medio de una notación que comunique de manera eficaz su significado.

DISEÑO

- El diseño es tanto un proceso como un modelo.
- El proceso de diseño es una secuencia de pasos que hacen posible que el diseñador describa todos los aspectos del software que se va a construir.

- Principios del diseño:
  1. En el proceso de diseño se deben tener en cuenta enfoques alternativos.
  2. Deberá poderse rastrear hasta el modelo de análisis.
  3. No deberá inventar nada que ya esté inventado.
  4. Deberá minimizar la distancia intelectual entre el software y el problema.
  5. Deberá presentar uniformidad e integración.
  6. Deberá estructurarse para admitir cambios.
  7. Deberá estructurarse para degradarse poco a poco, incluso cuando se enfrenta con datos, sucesos o condiciones de operaciones aberrantes.
  8. El diseño no es escribir código y escribir código no es diseñar.
  9. Deberá evaluarse en función de la calidad mientras se va creando, no después de terminado.
  10. Deberá revisarse para minimizar los errores conceptuales.

- Conceptos: Los conceptos de diseño de software fundamentales proporcionan el marco de trabajo necesario para conseguir que lo haga correctamente. Cada concepto proporciona al diseñador una base para aplicar los métodos de diseño. Los conceptos van a ayudar al diseñador a responder esas preguntas.
  - Abstracción: La noción de abstracción permite concentrarse en un problema a un nivel de generalización sin tener en cuenta los detalles irrelevantes de bajo nivel.
    Nivel de abstracción: A medida que profundizamos en la solución del problema se reduce el nivel de abstracción. Desde los requerimientos (abstractos) hasta llegar al código fuente.
    Tipos de abstracción:
    - Procedimental: Secuencia "nombrada" de instrucciones que tienen una funcionalidad específica. Ej.: Módulos (procedimientos, funciones, unidades, etc.).
    - De datos: Colección "nombrada" de datos que definen un objeto real Ej.: un registro que representa una persona con sus datos, el objeto persona en POO.
  - Arquitectura: Es la estructura general del software y las formas en que la estructura proporciona una integridad conceptual para un sistema.
  - Patrones: "Un patrón es una semilla de conocimiento, la cual tiene un nombre y transporta la esencia de una solución probada a un problema recurrente dentro de cierto contexto". Dicho de otro modo, describe una estructura de diseño que resuelve un problema de diseño particular dentro de un contexto específico. La finalidad de cada patrón de diseño es proporcionar una descripción que le permita al diseñador determinar si es aplicable al trabajo actual, si se puede reutilizar, si puede servir como guía para desarrollar un patrón similar pero diferente en cuanto a la funcionalidad o estructura.
  - Modularidad: El software se divide en componentes nombrados y abordados por separado, llamados frecuentemente módulos, que se integran para satisfacer los requisitos del problema. Códigos Monolíticos (un único módulo) y Modularización excesiva (a nivel de instrucciones)
  - Ocultamiento de información: La información que está dentro un módulo es inaccesible a otros que no la necesiten. Se consigue una modularidad efectiva definiendo un conjunto de módulos independientes que se comunican entre sí intercambiando sólo la información necesaria para su funcionalidad.
  - Independencia funcional: Modularidad + Abstracción + Ocultamiento de Información. Es deseable que cada módulo trate una subfunción de requisitos y tenga una interfaz sencilla para que sea más fácil de desarrollar, mantener, probar y reusar. Se mide mediante la cohesión y el acoplamiento entre los módulos. Se busca una alta cohesión y bajo acoplamiento
    - Cohesión (Coherente): Se define como la medida de fuerza o relación funcional existente entre las sentencias o grupos de sentencias de un mismo módulo. Un módulo es altamente cohesivo cuando lleva a cabo solo una tarea dentro del procedimiento y requiere poca

interacción con el resto de los procedimientos. Un módulo es poco cohesivo cuando realiza tareas muy diferentes o sin relación entre ellas

- Tipos de Cohesión:
  - Funcional → Cuando las sentencias o grupos de sentencias de un mismo módulo están relacionadas en el desarrollo de una única función (la cohesión más alta).
  - Coincidental (Casual) → Cuando las sentencias llevan a cabo un conjunto de tareas que no están relacionadas o tienen poca relación (la cohesión más baja).
  - Lógica → Cuando las sentencias se relacionan lógicamente.
  - Temporal → Cuando las sentencias se deben ejecutar en el mismo intervalo de tiempo.
  - Procedimental → Cuando la sentencia tiene que ejecutarse en un orden especifico.
  - Comunicacional → Cuando los elementos de procesamiento se centran en los datos de entrada y salida.
- Acoplamiento: Es la medida de interconexión entre los módulos. Punto donde se realiza la entrada o referencia y los datos que pasan a través de la interfaz. Una conectividad sencilla entre módulos da como resultado una conectividad más fácil.
- Niveles de Acoplamiento:
  - Bajo: Acoplamiento de datos. Acoplamiento de marca.
  - Moderado: Acoplamiento de control.
  - Alto: Acoplamiento común. Acoplamiento externo. Acoplamiento de contenido.

- Refinamiento: Se refina de manera sucesiva los niveles de detalle procedimentales. El refinamiento es un proceso de elaboración. Se comienza con una descripción de información de alto nivel de abstracción, sobre una funcionalidad puntual, sin conocer las características del funcionamiento, se va trabajando sobre la funcionalidad original proporcionando en cada iteración un mayor nivel de detalle hasta obtener todos los detalles necesarios para conocer su funcionamiento. La abstracción y el refinamiento son conceptos complementarios. La abstracción permite especificar procedimientos y datos sin considerar detalles de grado menor. El refinamiento ayuda a revelar los detalles de grado menor mientras se realiza el diseño.
- Refabricación o rediseño (Refactoring): Técnica de reorganización (sugerida por las metodologías ágiles) que simplifica el diseño de un componente sin cambiar su función o comportamiento. Cuando se refabrica el diseño existente, se examina en busca de redundancias, elementos inútiles, algoritmos innecesarios, estructuras de datos inapropiadas, etc. Ej: Una primera iteración del diseño podría producir un componente con poca cohesión. El diseñador puede decidir que el componente debe refabricarse en componentes distintos para elevar la cohesión.

DISEÑO ARQUITECTONICO

- Define la relación entre los elementos estructurales, para lograr los requisitos del sistema. Es el proceso de identificar los subsistemas dentro del sistema y establecer el marco de control y comunicación entre ellos. Los grandes sistemas se dividen en subsistemas que proporcionan algún conjunto de servicios relacionados.
- La arquitectura afecta directamente a los requerimientos no funcionales:
  - Los más CRÍTICOS:
    - Rendimiento: deben agrupar las operaciones críticas en un grupo reducido de sub-sistemas (componentes de grano grueso, baja comunicación).
    - Protección: la arquitectura deberá diseñarse para que las operaciones relacionadas con la protección se localicen en un único sub-sistema (o grupo pequeño), para reducir los costos y problemas de validación de la protección.
    - Seguridad: se debe utilizar una arquitectura en capas, protegiendo los recursos más críticos en las capas más internas.
    - Disponibilidad: la arquitectura se deberá diseñar con componentes redundantes para que sea posible el reemplazo sin detener el sistema, arquitectura muy tolerante a fallos.
    - Mantenibilidad: la arquitectura del sistema debe diseñarse con componentes autocontenidos de grano fino que puedan modificarse con facilidad.

1- Organización del sistema: representa la estrategia básica usada para estructurar el sistema. Los subsistemas de un sistema deben intercambiar información de forma efectiva (Todos los datos compartidos, se almacenan en una base de datos central. Cada subsistema mantiene su información y los intercambia entre los subsistemas).
    - Estilos organizacionales (PATRONES arquitectónicos):
        - Repositorios: La mayoría de los sistemas que usan grandes cantidades de datos se organizan alrededor de una base de datos compartida (repositorio). Los datos son generados por un subsistema y utilizados por otros subsistemas. Ejemplo: Sistemas de gestión, Sistemas CAD, Herramientas Case, etc.

            Ventajas: Forma eficiente de compartir grandes cantidades de datos, no hay necesidad de transmitir datos de un subsistema a otro. Los subsistemas que producen datos no deben saber cómo se utilizan. Las actividades de backup, protección, control de acceso están centralizadas. El modelo compartido es visible a través del esquema del repositorio. Las nuevas herramientas se integran de forma directa, ya que son compatibles con el modelo de datos.

            Desventajas: Los subsistemas deben estar acordes a los modelos de datos del repositorio. Esto en algunos casos puede afectar el rendimiento. La evolución puede ser difícil a medida que se genera un gran volumen de información de acuerdo con el modelo de datos establecido. La migración de estos modelos puede ser muy difícil, en algunos casos imposible. Diferentes subsistemas pueden tener distintos requerimientos de protección o políticas de seguridad y el modelo de repositorio impone las mismas para todos. Es difícil distribuir el repositorio en varias máquinas, existen repositorios centralizados lógicamente, pero pueden ocasionar problemas de redundancia e inconsistencias.
        - Cliente – Servidor: Es un modelo donde el sistema se organiza como un conjunto de servicios y servidores asociados, más unos clientes que utilizan los servicios.

            Componentes: Un conjunto de servidores que ofrecen servicios, otros sistemas. Un conjunto de clientes que llaman a los servicios. Una red que permite a los clientes acceder a los servicios. Caso particular cuando los servicios y el cliente corren en la misma máquina. Los clientes conocen el nombre del servidor y el servicio que brinda, pero el servidor no necesita conocer al cliente.
        - Arquitectura en capas: El sistema se organiza en capas, donde cada una de ellas presenta un conjunto de servicios a sus capas adyacentes.

            Ventajas: Soporta el desarrollo incremental. Es portable y resistente a cambios. Una capa puede ser reemplazada siempre que se mantenga la interfaz, y si varía la interfaz se genera una capa para adaptarlas. Permite generar sistemas multiplataforma, ya que solamente las capas más internas son dependientes de la plataforma (se genera una capa interna para cada plataforma).

            Desventajas: Difícil de estructurar. Las capas internas proporcionas servicios que son requeridos por todos los niveles. Los servicios requeridos por el usuario pueden estar brindados por las capas internas teniendo que atravesar varias capas adyacentes. Si hay muchas capas, un servicio solicitado de la capa superior puede tener que ser interpretado varias veces en diferentes capas.

2- Descomposición modular: Una vez organizado el sistema, a los subsistemas los podemos dividir en módulos, se puede aplicar los mismos criterios que vimos en la organización, pero la descomposición modular es más pequeña y permite utilizar otros estilos alternativos.
    - Estrategias de descomposición modular:
        - Descomposición orientada a flujo de funciones: Conjunto de módulos funcionales (ingresan datos y los transforman en salida). En un Modelo orientado a flujo de funciones, los datos fluyen de una función a otra y se transforman a medida que pasan por una secuencia de funciones hasta llegar a los datos de salida. Las transformaciones se pueden ejecutar en secuencial o en paralelo.

        - Descomposición orientada a objetos: Conjunto de objetos que se comunican. Un modelo arquitectónico orientado a objetos estructura al sistema en un conjunto de objetos débilmente acoplados y con interfaces bien definidas.
    - Definiciones:
        - Subsistema: Es un sistema en sí mismo cuyo funcionamiento no depende de los servicios proporcionados por otros. Los subsistemas se componen de módulos con interfaces definidas que se utilizan para comunicarse con otro subsistema.
        - Módulo: Es un componente de un subsistema que proporciona uno o más servicios a otros módulos. A su vez utiliza servicios proporcionados por otros módulos. Por lo general no se los considera un sistema independiente.

3- Modelos de control: En un sistema, los subsistemas están controlados para que sus servicios se entreguen en el lugar correcto en el momento preciso.
    - Los modelos de control a nivel arquitectónico:
        - Control Centralizado: Un subsistema tiene la responsabilidad de iniciar y detener otro subsistema. Un subsistema se diseña como controlador y tiene la responsabilidad de gestionar la ejecución de otros subsistemas, la ejecución puede ser secuencial o en paralelo. Modelo de llamada y retorno (Modelo de subrutinas descendentes. Aplicable a modelos secuenciales) Modelo de gestor (Un gestor controla el inicio y parada coordinado con el resto de los procesos. Aplicable a modelos concurrentes)
        - Control Basado en Eventos: Cada subsistema responde a eventos externos al subsistema. Se rigen por eventos generados externamente al proceso. Eventos (Señal binaria. Un valor dentro de un rango. Una entrada de un comando. Una selección del menú). Modelos de sistemas dirigidos por eventos (Modelos de transmisión (Broadcast): es un evento que se trasmite a todos los subsistemas, cualquier subsistema programado para manejar ese evento lo atenderá. Modelo dirigido por interrupciones: se utilizan en sistemas de tiempo real donde las interrupciones externas son detectadas por un manejador de interrupciones y se envía a algún componente para su procesamiento).

4- Arquitectura de los Sistemas Distribuidos: es un sistema en el que el procesamiento de información se distribuye sobre varias computadoras.
    - Tipos genéricos de sistemas distribuidos: Cliente-Servidor. Componentes distribuidos.
    - Características de los sistemas distribuidos:
        - Compartir recursos: un sistema distribuido permite compartir recursos
        - Apertura: son sistemas abiertos y se diseñan con protocolos estándar para simplificar la combinación de los recursos-
        - Concurrencia: varios procesos pueden operar al mismo tiempo sobre diferentes computadoras.
        - Escalabilidad: La capacidad puede incrementarse añadiendo nuevos recursos para cubrir nuevas demandas.
        - Tolerancia a fallos: la disponibilidad de varias computadoras y el potencial para reproducir información hace que los sistemas distribuidos sean más tolerantes a fallos de funcionamiento de hardware y software.
    - Desventajas:
        - Complejidad: Son más complejos que los centralizados, además del procesamiento hay que tener en cuenta los problemas de la comunicación y sincronización entre los equipos.
        - Seguridad: Se accede al sistema desde varias computadoras generando tráfico en la red que puede ser intervenido.
        - Manejabilidad: Las computadoras del sistema pueden ser de diferentes tipos y diferentes S.O. lo que genera más dificultades para gestionar y mantener el sistema
        - Impredecibilidad: La respuesta depende de la carga del sistema y del estado de la red, lo que hace que el tiempo de respuesta varié entre una petición y otra.

- Arquitectura Multiprocesador: El sistema de software está formado por varios procesos que pueden o no ejecutarse en procesadores diferentes. La asignación de los procesos a los procesadores puede ser predeterminada o mediante un dispatcher. Es común en sistemas grandes de tiempo real que recolectan información, toman decisiones y envían señales para modificar el entorno.
- Arquitectura Cliente-Servidor: Una aplicación se modela como un conjunto de servicios proporcionado por los servidores y un conjunto de clientes que usan estos servicios. Los clientes y servidores son procesos diferentes. Los servidores pueden atender varios clientes. Un servidor puede brindar varios servicios. Los clientes no se conocen entre sí.
    - Clasifican en niveles:
        - Dos Niveles: Cliente ligero (el procesamiento y gestión de datos se lleva a cabo en el servidor). Cliente pesado (el cliente implementa la lógica de la aplicación y el servidor solo gestiona los datos).
        - Multinivel: La presentación, el procesamiento y la gestión de los datos son procesos lógicamente separados y se pueden ejecutar en procesadores diferentes.
- Arquitectura de Componentes Distribuidos: Diseña al sistema como un conjunto de componentes u objetos que brindan una interfaz de un conjunto de servicios que ellos suministran. Otros componentes u objetos solicitan estos servicios. No hay distinción tajante entre clientes y servidores. Los componentes pueden distribuirse en varias máquinas a través de la red utilizando un middleware como intermediario de peticiones.
- Computación Distribuida inter-organizacional: Una organización tiene varios servidores y reparte su carga computacional entre ellos. Extender este concepto a varias organizaciones.
    - Arquitecturas Peer-to-Peer (P2P): Sistemas descentralizados en los que el cálculo puede llevarse a cabo en cualquier nodo de la red. Se diseñan para aprovechar la ventaja de la potencia computacional y el almacenamiento a través de una red. Pueden utilizar una arquitectura descentralizada (donde cada nodo rutea los paquetes a sus vecinos hasta encontrar el destino) Semi-centralizada (donde un servidor ayuda a conectarse a los nodos o coordinar resultados). Ejemplos: Torrents, Skype, ICQ, SETI@Home.
    - Arquitectura de sistemas orientados a servicios:
        - Servicio: Representación de un recurso computacional o de información que puede ser utilizado por otros programas. Un servicio es independiente de la aplicación que lo utiliza. Un servicio puede ser utilizado por varias organizaciones. Una aplicación puede construirse enlazando servicios. Las arquitecturas de las aplicaciones de servicios web son arquitecturas débilmente acopladas.
        - Funcionamiento: Un proveedor de servicios oferta servicios definiendo su interfaz y su funcionalidad. Para que el servicio sea externo, el proveedor publica el servicio en un “servicio de registro” con información del mismo. Un solicitante enlaza este servicio a su aplicación, es decir que el solicitante incluye el código para invocarlo y procesa el resultado del mismo.
        - Los estándares fundamentales que permiten la comunicación entre servicios:
            - SOAP (simple Object Access Protocol) Define una organización para intercambio de datos estructurados entre servicios web.
            - WSDL (Web Service Description Lenguage) Define como puede representarse las interfaces web.
            - UDDI (Universal Desscription Discovery and Integration) Estándar de búsqueda que define como puede organizarse la información de descripción de servicios.

CODIFICACION

- Una vez establecido el diseño, se deben escribir los programas que implementen dicho diseño.
- Pautas generales:
    - Localización de entrada y salida: es deseable localizarlas en componentes separados del resto del código ya que generalmente son más difíciles de probar.

    - Inclusión de pseudocódigo: Es útil avanzar el diseño, realizando un pseudocódigo para adaptar el diseño al lenguaje elegido.
    - Revisión y reescritura, no a los remiendos: Es recomendable realizar un borrador, revisarlo y reescribirlo tantas veces como sea necesario.
    - Reutilización: Hay dos clases de reutilización:
        - Productiva: se crean componentes destinados a ser reutilizados por otra aplicación.
        - Consumidora: Se usan componentes originalmente desarrollados para otros proyectos.
- Documentación: conjunto de descripciones escritas que explican al lector qué hace el programa y cómo lo hace.
    - Se divide en:
        - Documentación interna: Es concisa, escrita en un nivel apropiado para un programador. Contiene información dirigida a quienes leerán el código fuente. Incluye información de algoritmos, estructuras de control, flujos de control.
        - Documentación externa: Se prepara para ser leída por quienes, tal vez, nunca verán el código real. Por ejemplo, los diseñadores, cuando evalúan modificaciones o mejoras.

PRUEBAS

- Enfoque estratégico de pruebas:
    - Una estrategia de pruebas del software proporciona una guía que describe los pasos a seguir, cuándo se planean y llevan a cabo, cuánto esfuerzo, tiempo y recurso se requerirán. Proporciona: Planificación de las pruebas. Diseño de los casos de prueba. Ejecución de las pruebas. Recolección y evaluación de los datos resultantes.
    - La prueba es un conjunto de actividades que se planean con anticipación y se realizan de manera sistemática.
    - Conjunto de pasos en el que se incluyen técnicas y métodos específicos del diseño de casos de prueba.
    - Una estrategia de pruebas debe incluir pruebas de bajo nivel y de alto nivel.
    - Las actividades de las estrategias de pruebas son parte de la Verificación y Validación incluidas en el aseguramiento de la calidad del software.
- Concepto de verificación y validación: La verificación es el conjunto de actividades que asegura que el software implemente correctamente una función específica y validación es un conjunto diferente de actividades que aseguran que el software construido corresponde con los requisitos del cliente.
    - Verificación: ¿Estamos construyendo el producto correctamente? Comprobar que el software está de acuerdo con su especificación, donde se debe comprobar que satisface tanto los requerimientos funcionales como los no funcionales.
    - Validación: ¿Estamos construyendo el producto correcto? Es un proceso más general, cuyo objetivo es asegurar que el software satisface las expectativas del cliente.
- Tipos de pruebas software convencionales:
    - De unidad: Verifican que el componente funciona correctamente a partir del ingreso de distintos casos de prueba. Se prueba la interfaz del módulo para asegurar que la información fluye de forma adecuada. Se examinan las estructuras de datos locales. Se prueban las condiciones límite para asegurar que el módulo funciona correctamente. Se ejercitan todos los caminos independientes.
        - Los errores más comunes: Cálculos incorrectos (aplicación incorrecta de predecesores aritméticos. Operaciones mezcladas. Inicialización incorrecta. Falta de precisión. Representación simbólica incorrecta). Comparaciones erróneas. Flujos de control inapropiados.
        - Procedimiento: Como un componente no es un programa independiente, se debe desarrollar para cada prueba de unidad un software que controle y/o resguarde. Un controlador es un «programa principal» que acepta los datos del caso de prueba, pasa estos datos al módulo (a ser probado) y muestra los resultados.
          Un resguardo sirve para reemplazar a módulos subordinados al componente que hay que probar. Los controladores y resguardos son una sobrecarga de trabajo. Si los controladores y

resguardos son sencillos, el trabajo adicional es relativamente pequeño. La prueba de unidad se simplifica cuando se diseña un módulo con un alto grado de cohesión.

- De integración: Verifican que los componentes trabajan correctamente en forma conjunta. Técnica sistemática para construir la arquitectura del software mientras al mismo tiempo se aplican las pruebas. Se toman los componentes que han pasado las pruebas de unidad y se los combina según el diseño establecido. El programa se construye y se prueba en pequeños segmentos en los que los errores son más fáciles de aislar y de corregir
  - Integración descendente: Los módulos se integran al descender por la jerarquía de control, iniciando por el programa principal. Se puede realizar:
    - En profundidad: Primero-en-profundidad integra todos los módulos de un camino de control principal de la estructura.
    - En anchura: Primero-en-anchura incorpora todos los módulos directamente subordinados a cada nivel.
  - Integración ascendente: Se empieza la prueba con los módulos atómicos (es decir, módulos de los niveles más bajos de la estructura del programa). Dado que los módulos se integran de abajo hacia arriba, el proceso requerido de los módulos subordinados siempre está disponible y se elimina la necesidad de resguardos, pero no así, los conductores.
  - Pruebas de regresión: Cada vez que se añade un nuevo módulo como parte de una Prueba de integración, el software cambia. Se establecen nuevos caminos, pueden ocurrir nuevas E/S y se invoca una nueva lógica de control. En el contexto de una estrategia de Prueba de integración, la Prueba de regresión es volver a ejecutar un subconjunto de pruebas que se han llevado a cabo anteriormente para asegurarse de que los cambios no han propagado efectos colaterales no deseados. Esta prueba se puede hacer manualmente, volviendo a realizar un subconjunto de todos los casos de prueba o utilizando herramientas automáticas. El conjunto de pruebas de regresión contiene tres clases diferentes de casos de prueba:
    - una muestra representativa de pruebas que ejercite todas las funciones del software.
    - pruebas adicionales que se centren en las funciones del software que son probablemente afectadas por el cambio.
    - pruebas que se centren en los componentes del software que han cambiado.
  - Criticidad: Se deben identificar los módulos críticos, que pueden ser los que:
    1. Abordan muchos requerimientos de software
    2. Tienen alto nivel de control
    3. Es complejo o proclive a error
    4. Tiene requerimientos de rendimientos definidos.

    Deben probarse lo antes posible. Las pruebas de regresión deben hacer foco en ellos.
- De validación: Proporcionan una seguridad final de que el software satisface todos los requisitos funcionales y no funcionales. La validación del software se consigue mediante una serie de pruebas que demuestren la conformidad con los requisitos. Una vez que se procede con cada caso de prueba de validación, puede darse una de las dos condiciones:
  - Las características de funcionamiento o de rendimiento están de acuerdo con las especificaciones y son aceptables;
  - o se descubre una desviación de las especificaciones y se crea una lista de deficiencias.

Comienzan cuando finalizan las pruebas de integración. Revisión de la configuración (asegurar que todos los elementos de la configuración del software se hayan desarrollado apropiadamente, estén catalogados y contengan detalle suficiente para reforzar la fase de soporte).

- Pruebas de aceptación (ALFA y BETA): Las realiza el usuario final en lugar del responsable del desarrollo del sistema, una prueba de aceptación puede ir desde algo informal, hasta la ejecución sistemática de una serie de pruebas bien planificadas.
  - ALFA: Se llevan a cabo, por un cliente, en el lugar de desarrollo. Se usa el software de forma natural con el desarrollador como observador del usuario y registrando los errores y problemas de uso. Las pruebas alfa se hacen en un entorno controlado. Se

realizan después de que todos los procedimientos de prueba básicos, como las pruebas unitarias y pruebas de integración se han completado, y se produce después de las pruebas del sistema. Esta no es la versión final de software y cierta funcionalidad puede ser añadido al software incluso después de la prueba alfa.

- BETA: Se llevan a cabo por los usuarios finales del software en los lugares de trabajo de los clientes. El desarrollador no está presente normalmente. Así, la prueba beta es una aplicación en vivo del software en un entorno que no puede ser controlado por el desarrollador. El cliente registra todos los problemas que encuentra durante la prueba beta e informa a intervalos regulares al desarrollador. Las pruebas beta es la última fase de las fases de prueba y se hace utilizando técnicas de caja negra. A veces la versión beta también es liberada en el mercado, y en base a las modificaciones que se hacen comentarios de los usuarios o si no hay cambios en el software se libera.

- Del sistema: Verifica que cada elemento encaja de forma adecuada y que se alcanza la funcionalidad y el rendimiento del sistema total. Está constituida por una serie de pruebas diferentes. Aunque cada prueba tiene un propósito diferente, todas trabajan para verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas.
  - Pruebas de recuperación: Se controla la recuperación de fallas y el modo de reanudación del procesamiento en un tiempo determinado. Generalmente se fuerza el fallo para comprobarlo.
  - Pruebas de seguridad: Se comprueban los mecanismos de protección integrados.
  - Pruebas de resistencia (Stress): Se diseñan para enfrentar a los programas a situaciones anormales.
  - Prueba de rendimiento: Se prueba el sistema en tiempo de ejecución. A veces va emparejada con la Prueba de resistencia.

- Depuración: La depuración de programas, es el proceso de identificar y corregir errores en programas informáticos. La depuración no es una prueba, pero siempre ocurre como consecuencia de la prueba efectiva. Es decir, se descubre un error, la depuración elimina dicho error.
  El proceso de depuración siempre tiene uno de los dos resultados: Se encuentra la causa, se corrige y se elimina; o No se encuentra la causa. La persona que realiza la depuración debe sospechar la causa, diseñar un caso de prueba que ayude a confirmar sus sospechas y el trabajo vuelve hacia atrás a la corrección del error de una forma iterativa.
  - Características de los errores:
    1.Síntoma lejano (geográficamente) de la causa.
    2.Síntoma desaparece temporalmente al corregir otro error.
    3.Síntoma producido por error.
    4.Síntoma causado por error humano.
    5.Síntoma causado por problemas de tiempo.
    6.Condiciones de entrada difíciles de reproducir.
    7.Síntoma intermitente (especialmente en desarrollos hardware-software).
    8.El síntoma se debe a causas distribuidas entre varias tareas que se ejecutan en diferentes procesadores.
  - Enfoques de la depuración: Diseñar programas de prueba adicionales que repitan la falla original y ayuden a descubrir la fuente de la falla en el programa. Rastrear el programa manualmente y simular la ejecución. Usar las herramientas interactivas. Una vez corregido el error debe reevaluarse el sistema: volver a hacer las inspecciones y repetir las pruebas (pruebas de regresión).
  - Pruebas de entornos especializados: A medida que el software se hace más complejo, crece también la necesidad de enfoques de pruebas especializados.
    - Pruebas de interfaces gráficas
    - Pruebas de arquitecturas cliente-servidor: Pruebas de funcionalidad de la aplicación. Prueba de servidor. Prueba de base de datos. Pruebas de transacciones. Pruebas de comunicación de red.

- Pruebas de la documentación y ayuda: Es importante para la aceptación del programa. Revisar la guía del usuario o funciones de ayuda en línea. Prueba de documentación es en dos fases: Revisar e inspeccionar (examinar la claridad editorial del documento). Prueba en vivo (usar la documentación junto con el programa real).
- Pruebas de sistema en tiempo real: El diseño de los casos de prueba, además de los convencionales deben incluir manejo de eventos (interrupciones), temporización de los datos, el paralelismo entre las tareas, etc. Pruebas de tareas. Pruebas de comportamiento. Pruebas inter-tareas. Pruebas de sistema.

PRUEBAS - TIPOS DE DEFECTO

- Algorítmicos Ej.: No inicializar variables
- De sintaxis Ej.: Confundir un 0 por una O
- De precisión Ej.: Fórmulas no implementadas correctamente
- De documentación Ej.: Documentación no acorde con lo que hace el software
- De sobrecarga Ej.: El sistema funciona bien con 100 usuarios, pero no con 110.
- De capacidad Ej.: El sistema funciona bien con ventas <1.000.000
- De coordinación o sincronización Ej.: Comunicación entre procesos con fallas
- De rendimiento Ej.: Tiempo de respuesta inadecuado.
- De recuperación Ej.: No volver a un estado normal luego de una falla
- De relación hardware-software Ej.: Incompatibilidad entre componentes
- De estándares Ej.: No cumplir con la definición de estándares y procedimientos

- Clasificación ortogonal de defectos: Primeramente, se debe identificar si es un:
  - Defecto por omisión (resulta cuando algún aspecto clave del código falta). Ej: variable no inicializada.
  - Defecto de cometido (resulta cuando algún aspecto es incorrecto). Ej: variable inicializada con un valor erróneo.
- Objetivo y beneficios: Diseñar pruebas que saquen a la luz diferentes clases de errores, haciéndolo en la menor cantidad de tiempo y esfuerzo. Descubrir errores antes que el software salga del ambiente de desarrollo. Detectar un error no descubierto hasta entonces. Bajar los costos de corrección de errores en la etapa de mantenimiento. La prueba tiene éxito cuando descubre errores.
- Principios de la prueba: A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente. Las pruebas deberían planificarse mucho antes de que empiecen. Es aplicable el principio de Pareto. El mismo dice que "el 80% de los errores de un software es generado por un 20% del código de dicho software, mientras que el otro 80% genera tan sólo un 20% de los errores". Las pruebas deberían empezar por «lo pequeño» y progresar hacia «lo grande». Es importante asegurarse que se han aplicado (probado) todas las condiciones a nivel de componente. Para ser más eficaces, las pruebas deberían ser realizadas por un equipo independiente.
- Prueba de caja blanca (abierta o cristal): se basa en el minucioso examen de los detalles procedimentales. Se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles.
  Deriva casos de prueba de la estructura de control, para verificar detalles procedimentales. Mediante los métodos de prueba de caja blanca, el ingeniero del software puede obtener casos de prueba que garanticen que se ejercita por lo menos una vez todos los caminos independientes de cada módulo. Ejerciten todas las decisiones lógicas. Ejecuten todos los bucles en sus límites. Ejerciten las estructuras internas de datos para asegurar su validez.
  - Prueba del camino básico: Es una técnica propuesta por Tom McCabe. Permite al diseñador de casos de pruebas obtener una medida de la complejidad lógica y usarla como guía para la definición de caminos de ejecución. Los casos de prueba obtenidos garantizan que se ejecuta al menos una vez cada sentencia del programa. La complejidad ciclomática es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Define el número de caminos independientes del conjunto básico de un programa y nos da un límite superior para el número de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia al menos una

vez. Un camino independiente es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una nueva condición.
Complejidad ciclomática:
1. V(g)=Cantidad de regiones del grafo ó
2. V(g)= A - N + 2 ó
3.V(g)= P + 1
La complejidad ciclomática debe medirse con las tres fórmulas de manera de verificar su exactitud.
Pasos:
1. Dibujar el grafo de flujo correspondiente.
2. Determinar la complejidad ciclomática.
3. Determinar un conjunto básico de caminos independientes.
4. Preparar los casos de prueba que forzarán la ejecución de cada camino del conjunto.
5. Ejecutar cada caso de prueba y comparar los resultados obtenidos con los esperados.

- Prueba de caja negra (cerrada): se refiere a las pruebas que se llevan a cabo sobre la interfaz del software. También denominada prueba de comportamiento, se centran en los requisitos funcionales del software. Intenta descubrir diferentes tipos de errores que los métodos de caja blanca. Busca errores como:
    - Funciones incorrectas o ausentes
    - Errores de interfaz
    - Errores en estructuras de datos o en accesos a bases de datos externas
    - Errores de rendimiento
    - Errores de inicialización y de terminación
    - Concentra la prueba en el dominio de información: por ejemplo, las pruebas de partición equivalente.
    - Prueba de partición equivalente: El diseño de los casos de prueba se basa en una evaluación de las clases de equivalencia para una condición de entrada. Una clase de equivalencia representa un conjunto de estados válidos o no válidos para condiciones de entrada. Una condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición lógica. Las clases de equivalencia se pueden definir de acuerdo con las siguientes directrices:
        - Si una condición de entrada especifica un rango, se define una clase de equivalencia válida y dos no válidas.
        - Si una condición de entrada requiere un valor específico, se define una clase de equivalencia válida y dos no válidas.
        - Si una condición de entrada especifica un elemento de un conjunto, se define una clase de equivalencia válida y una no válida.
        - Si una condición de entrada es lógica, se define una clase de equivalencia válida y una no válida.
    - Análisis de valores limites (AVL): Los errores tienden a darse más en los límites del campo de entrada que en el centro. Complementa a la partición equivalente. En lugar de seleccionar cualquier elemento de una clase de equivalencia, el AVL selecciona los casos de prueba en los extremos de la clase. En lugar de centrarse solamente en las condiciones de entrada, el AVL obtiene casos de prueba también para el campo de salida.

MANTENIMIENTO

- Atención del sistema a lo largo de su evolución después que el sistema se ha entregado. A esta fase se la llama “Evolución del Sistema”. En ocasiones debe realizarse mantenimiento a sistemas “heredados”. Es necesario evaluar cuándo es conveniente cerrar el ciclo de vida de ese sistema y reemplazarlo por otro. La decisión se toma en función del costo del ciclo de vida del viejo proyecto y la estimación del nuevo proyecto En ocasiones la complejidad del sistema crece por los cambios.
- Características: Su consecuencia es la disminución de otros desarrollos. Las modificaciones pueden provocar disminución de la calidad total del producto. Las tareas de mantenimiento generalmente provocan reiniciar las fases de análisis, diseño e implementación. Mantenimiento estructurado vs. no estructurado. Involucra

entre un 40% a 70% del costo total de desarrollo. Los errores provocan insatisfacción del cliente. Pueden existir efectos secundarios sobre código, datos, documentación.

- Actividades: Debe utilizarse un mecanismo para realizar los cambios que permita: identificarlos, controlarlos, implementarlos e informarlos. El proceso de cambio se facilita si en el desarrollo están presentes los atributos de claridad, modularidad, documentación interna del código fuente y de apoyo.
- Ciclo de mantenimiento:
    - Análisis: comprender el alcance y el efecto de la modificación.
    - Diseño: rediseñar para incorporar los cambios.
    - Implementación: recodificar y actualizar la documentación interna del código.
    - Prueba: revalidar el software.
    - Actualizar la documentación de apoyo.
    - Distribuir e instalar las nuevas versiones.
- Facilidades para ayudar al mantenimiento:
    - Análisis: Señalar principios generales, armar planes temporales, especificar controles de calidad, identificar posibles mejoras, estimar recursos para mantenimiento.
    - Diseño arquitectónico: Claro, modular, modificable, con notaciones estandarizadas.
    - Diseño detallado: Notaciones para algoritmos y estructuras de datos, especificación de interfaces, manejo de excepciones, efectos colaterales.
    - Implementación: Indentación, comentarios de prólogo e internos, codificación simple y clara.
    - Verificación: Lotes de prueba y resultados.
- Tipos de mantenimiento:
    - Mantenimiento correctivo: Diagnóstico y corrección de errores.
    - Mantenimiento adaptativo: Modificación del software para interaccionar correctamente con el entorno.
    - Mantenimiento perfectivo: Mejoras al sistema.
    - Mantenimiento preventivo: Se efectúa antes que haya una petición, para facilitar el futuro mantenimiento. Se aprovecha el conocimiento sobre el producto.
- Métricas: Tiempo de reconocimiento del problema. Tiempo de búsqueda de herramientas para mantenimiento. Tiempo de análisis del problema. Tiempo de especificación de cambios. Tiempo activo de modificación. Tiempo de prueba local. Tiempo de prueba global. Tiempo de revisión de mantenimiento. Tiempo total de recuperación.
- Evaluación: Promedio de fallas por ejecución. Total de personas-hora por cada categoría de mantenimiento. Promedio de cambios por programa, lenguaje y tipo de mantenimiento. Promedio personas-hora por sentencias añadidas y eliminadas. % de peticiones de mantenimiento por tipos.
- Rejuvenecimiento del software: Es un desafío del mantenimiento, intentando aumentar la calidad global de un sistema existente. Contempla retrospectivamente los subproductos de un sistema para intentar derivar la información adicional o reformarlo de un modo comprensible.
    - Tipos de Rejuvenecimiento:
        - Re-documentación: Representa un análisis del código para producir la documentación del sistema.
        - Re-estructuración: Se reestructura el software para hacerlo más fácil de entender.
        - Ingeniería Inversa: Parte del código fuente y recupera el diseño y en ocasiones la especificación, para aquellos sistemas en los que no hay documentación.
        - Re-ingeniería: Extensión de la ingeniería Inversa Produce un nuevo código fuente correctamente estructurado, mejorando la calidad sin cambiar la funcionalidad del sistema.

AUDITORIA INFORMATICA

- Auditoría: Es un examen crítico que se realiza con el objeto de evaluar la eficiencia y la eficacia de una sección o de un organismo y determinar cursos alternativos de acción para mejorar la organización y lograr los objetivos propuestos. No es una actividad meramente mecánica que implique la aplicación de ciertos procedimientos cuyos resultados son de carácter indudable. Requiere de un juicio profesional, sólido y maduro, para juzgar los procedimientos que deben seguirse y evaluar los resultados obtenidos. Puede ser interna, externa o una combinación de ambas.

Por lo tanto, es la revisión y evaluación de: los controles, sistemas y procedimientos de la informática; los equipos de cómputo, su utilización, eficiencia y seguridad; la organización que participa en el procesamiento de la información, a fin de que por medio del señalamiento de cursos alternativos se logre una utilización más eficiente, confiable y segura de la información que servirá para una adecuada toma de decisiones.
Permite definir estrategias para prevenir delitos o problemas legales, definir seguridades de accesos en el sistema, documentar los cambios de configuración, verificar la aplicación de normas de calidad.
Es una actividad preventiva, el auditor sugiere.
Los procedimientos de auditoría en informática varían de acuerdo con la filosofía y técnica de cada organización y departamento de auditoría en particular.
La auditoría en informática debe evaluar todo (informática, organización del centro de cómputo, computadoras, comunicación y programas).

- Definiciones:
  "Es una función que ha sido desarrollada para asegurar la salvaguarda de los activos de los sistemas de computadoras, mantener la integridad de los datos y lograr los objetivos de la organización en forma eficaz y eficiente". Ron Weber.
  "Es la verificación de los controles en las siguientes tres áreas de la organización (informática): Aplicaciones, Desarrollo de sistemas, Instalación del centro de cómputos". William Mair.
- Objetivos: Salvaguardar los activos. Se refiere a la protección del hardware, software y recursos humanos. Integridad de datos. Los datos deben mantener consistencia y no duplicarse. Efectividad de sistemas. Los sistemas deben cumplir con los objetivos de la organización. Eficiencia de los sistemas. Que se cumplan los objetivos con los menores recursos. Seguridad y confidencialidad.
- Influencia: Factores que pueden influir en la organización a través del control y la auditoría en informática:
  - Controlar el uso de la computadora.
  - Los altos costos que producen los errores en una organización.
  - Abuso en las computadoras.
  - Posibilidad de pérdida de capacidades de procesamiento de datos.
  - Posibilidad de decisiones incorrectas.
  - Valor del hardware, software y personal.
  - Necesidad de mantener la privacidad individual.
  - Posibilidad de pérdida de información o mal uso de la misma.
  - Toma de decisiones incorrectas.
  - Necesidad de mantener la privacidad de la organización.
- Campo de acción:
  1- Evaluación administrativa del área de informática: Los objetivos del área informática (departamento, dirección o gerencia informática). Metas, planes, políticas y procedimientos de procesos electrónicos estándar. Organización del área y su estructura orgánica. Funciones y niveles de autoridad y responsabilidad del área de procesos electrónicos. Integración de los recursos materiales y técnicos. Costos y controles presupuestarios. Controles administrativos del área de procesos electrónicos.
  2- Evaluación de los sistemas y procedimientos, y de la eficiencia que se tiene en el uso de la información: Evaluación del análisis de los sistemas y sus diferentes etapas. Evaluación del diseño lógico. Evaluación del desarrollo físico del sistema. Facilidades para la elaboración de los sistemas. Control de proyectos. Control de sistemas y programación. Instructivos y documentación. Formas de implantación. Seguridad física y lógica de los sistemas. Confidencialidad de los sistemas. Controles de mantenimiento y formas de respaldo de los sistemas. Utilización de los sistemas.
  3- Evaluación del proceso de datos, de los sistemas y de los equipos de cómputo (software, hardware, redes, bases de datos, comunicaciones): Controles de los datos fuente y manejo de las cifras de control. Control de operación. Control de salida. Control de asignación de trabajo. Control de medios de almacenamiento masivo. Control de otros elementos de cómputo. Control de medios de comunicación. Orden en el centro de cómputo.
  4- Seguridad y confidencialidad: Seguridad física y lógica. Confidencialidad. Respaldos. Seguridad del personal. Seguros. Seguridad en la utilización de equipos. Plan de contingencia y procedimiento de respaldo en caso de desastre. Restauración de equipo y de sistemas.
  5- Aspectos legales de los sistemas y de la información.