INGENIERÍA DE SOFTWARE 2

Recopilación extensa de todo el contenido de la materia

CAMILA CHIVELLI

2023

CONTENIDO

ESTIMACIONES

- Estimación de recursos
- Estimación de tiempo
- Estimación de costos
- Fijación de Precio y la Relación Precio-Costo

TÉCNICAS DE ESTIMACIÓN

- Juicio Experto
- Planning Poker

MODELOS EMPÍRICOS DE ESTIMACIÓN

- COCOMO II
- Composición de COCOMO II

PLANIFICACIÓN TEMPORAL

CALENDARIZACIÓN

- Componentes
- Parámetros de una tarea
- Retrasos

MODELO DE PLANIFICACIÓN

- GANT
- Red de Tareas
- PERT
- CMP
- Método del Camino Crítico

PLANIFICACIÓN ORGANIZATIVA

- Participantes
- Líderes
- Equipo de Software
- Comunicación grupal
- Organigramas de equipos

ANÁLISIS DE RIESGOS

- Componentes
- Aspectos
- Factores de riesgo
- Tipos de riesgo
- Administración
- Estrategia de gestión
- Proceso de gestión

CONCEPTOS DE DISEÑO

- Importancia
- Abstracción
- Arquitectura
- Patrones
- Modularidad
- Ocultamiento de información
- Independencia funcional
- Refinamiento
- Refabricación

DISEÑO DE SOFTWARE

- Importancia
- Áreas
- Criterios de evaluación
- Criterios técnicos
- Principios del modelado de diseño
- Evolución

DISEÑO ARQUITECTÓNICO

- Requisitos no funcionales

ASPECTOS DEL DISEÑO ARQUITECTÓNICO

ORGANIZACIÓN DEL SISTEMA

- Patrón de Repositorio
- Patrón Cliente-Servidor
- Patrón de Arquitectura en Capas

DESCOMPOSICIÓN MODULAR

- Orientada a Flujo de Funciones
- Orientada a Objetos

MODELOS DE CONTROL

- Control Centralizado
- Sistemas Dirigidos por Eventos

ARQUITECTURA DE LOS SISTEMAS DIRIGIDOS

- Ventajas y desventajas
- Multiprocesador
- Cliente-Servidor
- Objetos Distribuidos
- Inter-Organizacional

IMPLEMENTACIÓN

- Codificación
- Pautas generales
- Documentación

ASPECTOS DEL DISEÑO DE INTERFAZ

Las 3 reglas doradas de Theo Mandel

USABILIDAD

- Concepto
- Factores que determinan la usabilidad
- Principios de Usabilidad de Jacob Nielsen

Estilos de Interface

Soporte

DISEÑO UX/UI

DISEÑO DE INTERFAZ DE USUARIO

- Objetivos
- Conceptos iniciales
- Los 6 principios para el diseño UI
- Etapas

DISEÑO DE EXPERIENCIA DE USUARIO

- Tipos de diseño
- Etapas
- Investigación de usuarios
- Perfil del usuario
- Contexto y ambiente de trabajo

PRESENTACIÓN DE LA INFORMACIÓN EN PANTALLA

- Enfoque UI
- Enfoque UX

Iceberg de UX/UI

Diferencias entre UX y UI

PRUEBAS DEL SOFTWARE

- Aspectos
- Principios
- Equipo independiente de pruebas
- Origen de errores
- Tipos de errores
- Clasificación ortogonal de defectos
- Clasificación de defectos de Pfeegler

TIPOS DE PRUEBAS DEL SOFTWARE

CAJA BLANCA

- Prueba del Camino Básico
- Complejidad Ciclomática
- Componentes
- Cálculo

CAJA NEGRA

- Prueba de Partición Equivalente
- Análisis de Valores Límite (AVL)

ESTRATEGIAS DE PRUEBAS

- Enfoque estratégico de pruebas
- Concepto de Verificación y Validación

PRUEBAS DE UNIDAD

- Procedimiento
- Prueba de Unidad Orientadas a Objetos

PRUEBAS DE INTEGRACIÓN

- Pruebas de regresión
- Criticidad

ESTRATEGIAS

- Descendente
- Ascendente
- Sándwich

- Pruebas Integración Orientadas a Objetos

PRUEBAS DEL SISTEMA

- Tipos

PRUEBAS DE VALIDACIÓN

- Tipos

OTRAS PRUEBAS

- Prueba de Entornos Especializados
- Prueba de la Documentación y Funciones de Ayuda
- Pruebas de Sistemas de Tiempo Real
- Prueba de Arquitectura Cliente-Servidor
- Pruebas de Interfaces Gráficas

DEPURACIÓN

- Orígenes de los errores
- Enfoques de la depuración

CONTROL DEL SOFTWARE

GESTIÓN DE LA CONFIGURACIÓN DEL SOFTWARE - GCS

- Utilidad
- Importancia
- Línea Base
- Etapas

MANTENIMIENTO

- Barrera de mantenimiento
- Desventajas
- Tipos
- Actividades en torno a un cambio
- Ciclo de mantenimiento
- Rejuvenecimiento del Software
- Tipos de Rejuvenecimiento

AUDITORÍA

- Definiciones
- Objetivos
- Influencia
- Campo de acción

SOFTWARE REQUIREMENTS SPECIFICATIONS

Las Especificaciones de Requisitos de Software (SRS) son documentos detallados que describen de manera exhaustiva los requisitos y características de un sistema de software. Estas especificaciones actúan como un contrato entre los clientes, usuarios y desarrolladores, estableciendo claramente qué funcionalidades debe tener el sistema y cómo debe comportarse.

Naturaleza

El SRS es una especificación para un producto de software particular. Es redactado por uno o más representantes del equipo de desarrollo y uno o más representantes del cliente, o incluso por ambos.

Ambiente

El software puede contener toda la funcionalidad del proyecto o formar parte de un sistema más amplio. Es crucial describir el entorno en el cual el proyecto tendrá impacto y cómo se integrará con la funcionalidad necesaria. En el último caso, se debe detallar las interfaces entre el sistema y el software desarrollado, así como los requerimientos de funcionalidad externa.

ALCANCE

El alcance se refiere a los límites y fronteras de un proyecto de desarrollo de software. Describirlo implica definir qué incluirá y qué no incluirá el sistema, así como establecer cuál es el ámbito del proyecto y qué funcionalidades y características se abordarán.

El alcance del SRS determina qué requerimientos y características específicas del software se describirán en detalle en el documento y cuáles podrían abordarse en documentos o fases posteriores. Este documento realiza una especificación de requerimientos escrita con buenas prácticas para describir los contenidos y cualidades de los requerimientos.

CARACTERÍSTICAS DEL SRS

Un SRS efectivo no solo se trata de enumerar una lista de requisitos, sino también de garantizar que esos requisitos sean claros, coherentes y comprensibles tanto para el equipo de desarrollo como para el resto de los stakeholders involucrados en el proyecto. Para lograr esto, el SRS debe exhibir una serie de características fundamentales que aseguran su calidad y utilidad a lo largo de todo el ciclo de vida del desarrollo de software.

Correcto	Es correcto si y solo si cada requisito declarado está presente en el software.
No ambiguo	Es inequívoco si y solo si cada requisito declarado tiene una única interpretación.
Completo	Está completo si y solo si cubre todos los aspectos necesarios para que el sistema funcione correctamente y cumpla con los objetivos establecidos
Consistente	La consistencia se refiere a la consistencia interior. Si un SRS no concuerda con algún documento de nivel superior, como una especificación de requerimientos del sistema, entonces carece de consistencia. Por ejemplo, con respecto a la pila del producto, todos los requisitos de la pila del producto deben estar presentes en el SRS y viceversa.
Priorizado	Se prioriza por la importancia de sus requerimientos particulares.
Comprobable	Es comprobable si y solo si cada requisito declarado es verificable. Un requisito es verificable si existe un proceso mediante el cual una persona o una máquina puede verificar que el producto de software cumple con el requisito. Por lo general, cualquier requisito ambiguo no es verificable.
Modificable	Es modificable si y solo si su estructura y estilo permiten realizar cambios a los requerimientos de manera sencilla, completa y coherente, preservando al mismo tiempo la estructura, el estilo y la consistencia.
Trazable	La trazabilidad se refiere a la capacidad de seguir y comprender el origen de cada requerimiento, así como su capacidad de ser rastreado hacia adelante y hacia atrás en todo el proceso de desarrollo del software. En otras palabras, desde un requerimiento específico registrado en el SRS, debería ser posible seguir su camino hasta llegar a su diseño asociado y a su implementación en el código, y también realizar el proceso inverso.

CONSIDERACIONES PARA UN BUEN SRS

Preparación conjunta

El SRS debe ser elaborado en colaboración con todas las partes interesadas para lograr un acuerdo sólido y equitativo entre las partes involucradas.

Evolución

El SRS debe evolucionar de manera paralela al desarrollo del software, registrando todos los cambios que se realicen, identificando a los responsables de dichos cambios y asegurando la aceptación de las modificaciones por parte de las partes involucradas.

Prototipos

La utilización de prototipos es una práctica frecuente para definir y refinar los requerimientos de manera más precisa, permitiendo a los stakeholders visualizar y experimentar cómo se verá y funcionará la implementación de los requerimientos en el software.

Diseño incorporado

El SRS puede incluir los atributos o funciones externas que el sistema debe tener para interactuar con otros subsistemas o componentes. Esto puede involucrar detalles de diseño que son relevantes para la comprensión del funcionamiento del sistema.

Requerimientos incorporados

Los detalles particulares de algunos requerimientos específicos son anexados como documentos externos. Estos documentos complementarios proporcionan detalles más profundos sobre ciertos aspectos de los requerimientos, como es el caso de los casos de uso, planes de proyecto, planes de aseguramiento de calidad, entre otros.

FICHA DE CONTENIDO DE UN SRS

FICHA DEL DOCUMENTO CONTENIDO

1. INTRODUCCIÓN
    - 1.1 Propósito
    - 1.2 Alcance
    - 1.3 Referencias
2. DESCRIPCIÓN GENERAL
    - 2.1 Perspectiva del producto
    - 2.2 Funcionalidad del producto
    - 2.3 Características de los usuarios
    - 2.4 Evolución previsible del sistema
3. REQUISITOS NO FUNCIONALES
    - A. Requisitos de rendimiento
    - B. Seguridad
    - C. Portabilidad
4. MANTENIMIENTO
5. APÉNDICES

CONTENIDO DEL DOCUMENTO

1. INTRODUCCIÓN

Se definen las bases del proyecto y el dominio en donde se verá reflejado el producto.

1.1 Propósito

Se define el propósito del documento y se especifica a quién va dirigido.

1.2 Alcance

Se nombra al sistema futuro, explicando su funcionalidad y lo que no abordará. También se describen los beneficios, objetivos y metas del sistema.

1.3 Referencias

Se presenta una lista completa de todas las referencias de documentos utilizados para escribir el SRS. Identifica cada documento con título, número de reporte, fechas, publicación y fuentes de referencia.

2. DESCRIPCIÓN GENERAL

Se detallan factores generales que influyen en el producto.

2.1 Perspectiva del producto

Se establece si el producto es independiente o forma parte de un sistema más grande. En este último caso, se relacionan los requerimientos con la funcionalidad del sistema mayor y se identifican interfaces.

2.2 Funciones del sistema

Se proporciona un resumen de las funciones del futuro sistema, presentándolas de manera organizada. Pueden incluirse gráficos que muestren las relaciones entre funciones, sin entrar en el diseño.

2.3 Características del Usuario

Se describen las características generales de los usuarios previstos, como nivel educativo, experiencia y especialización técnica.

2.4 Evoluciones previsibles del sistema

Se identifican requerimientos que serán implementados en futuras versiones del sistema.

3. REQUERIMIENTOS NO FUNCIONALES

Se presentan los requisitos no funcionales en detalle para permitir a los diseñadores construir el sistema y a los auditores verificar que se cumpla con lo descripto.

3.1 Requerimientos de rendimiento

Se definen requisitos relacionados con la carga soportada por el sistema, estableciendo mediciones mensurables y específicas.

3.2 Seguridad

Especificación de los elementos que protegerán el software de accesos no autorizados, usos inapropiados y actos maliciosos, además de prevenir modificaciones o destrucciones tanto maliciosas como accidentales.

3.3 Portabilidad

Se especifican atributos que permitirán al software trasladarse a diferentes plataformas o entornos.

4. MANTENIMIENTO

Se identifican el tipo y las responsabilidades del mantenimiento necesario, estableciendo cuándo y quién llevará a cabo estas tareas.

5. APÉNDICE

Pueden contener información relevante para la SRS pero que no forma parte directa de ella, como casos de uso o historias de usuario.

GESTIÓN DE PROYECTOS

La Gestión de Proyectos se refiere al proceso de planificar, organizar, dirigir y controlar los recursos y actividades necesarios para lograr los objetivos de un proyecto de manera exitosa. Esto implica una serie de prácticas y metodologías para asegurar que el proyecto se complete dentro de los límites de tiempo, costo y alcance establecidos, mientras se mantienen altos estándares de calidad.

Se basa en 4 aspectos

- **Planificación**
- **Dirección**
- **Organización**
- **Control**

PROYECTO

Un proyecto es un esfuerzo temporal y progresivo con un inicio y fin definidos, que se lleva a cabo para crear un producto, servicio o resultado único. Además, puede considerarse como la subdivisión de un problema inicial en tareas más pequeñas y sencillas, lo que permite manejar la complejidad del software que se desea desarrollar.

- **Temporal**
  Un proyecto tiene un punto de inicio claro, marcado por la fase inicial de planificación y organización, y un fin definido, que se alcanza cuando los objetivos del proyecto han sido logrados satisfactoriamente o cuando se reconoce que no se pueden alcanzar.

- **Resultado**
  Estos resultados pueden ser productos, servicios, resultados únicos o logros específicos que se crean para satisfacer necesidades particulares.

- **Elaboración gradual**
  La metodología de desarrollo gradual es común en los proyectos. Esto implica dividir el proyecto en fases o etapas, desarrollando y mejorando incrementos sucesivos del producto a lo largo del tiempo.

LAS 4 P DE LA GESTIÓN DE PROYECTOS DE SOFTWARE

Personal (RRHH)

El factor humano es fundamental en la gestión de proyectos de software ya que son quienes realizan el esfuerzo. Identificar a los interesados, conocer sus capacidades y expectativas, y gestionar eficazmente su participación es esencial para asegurar el éxito del proyecto.

Producto

El producto de software es intangible, lo que puede dificultar la evaluación del progreso. Planificar el proyecto con claridad es crucial para establecer objetivos y alcances precisos del producto. Esto implica considerar soluciones alternativas, identificar restricciones, determinar técnicas y enfoques administrativos adecuados, entre otros aspectos.

Proceso

El proceso de software brinda la estructura desde la cual se puede establecer un plan detallado para el desarrollo del software. Definir un proceso efectivo es esencial para garantizar una gestión coherente y controlada del proyecto, así como para establecer estándares y procedimientos que guíen el trabajo del equipo.

Proyecto

Los proyectos deben ser planificados y controlados para manejar su complejidad. Esto implica comprender los factores que contribuyen a su éxito y los riesgos asociados. La gestión adecuada del proyecto abarca la definición de objetivos claros, la asignación de recursos, la monitorización del progreso y la toma de medidas para mantener el proyecto en línea con sus objetivos.

MANIFESTACIÓN DE UNA MALA GESTIÓN

Incumplimiento de plazos

Cuando un proyecto no se completa dentro de los plazos establecidos, puede tener un impacto significativo en la confianza de los clientes, la moral del equipo y la viabilidad general del proyecto. La planificación deficiente, la falta de seguimiento y los cambios no controlados en el alcance son factores que pueden contribuir a esta situación.

Incremento de los costos

Si los costos del proyecto exceden el presupuesto planificado, puede causar dificultades financieras y afectar la rentabilidad. Problemas en la estimación de costos, un control inadecuado de los gastos y una mala gestión de los recursos pueden llevar a un aumento en los costos totales del proyecto.

Entrega de productos de mala calidad

Entregar productos o servicios de baja calidad puede dañar la reputación de la organización y afectar la satisfacción del cliente. Una planificación y control inadecuados pueden resultar en la entrega de productos que no cumplen con los estándares de calidad esperados. La falta de pruebas y problemas de comunicación también pueden contribuir a este problema.

En conjunto, estas manifestaciones de mala gestión pueden generar un impacto económico negativo. Los costos adicionales, el tiempo perdido y los problemas de calidad pueden resultar en pérdida de ingresos y oportunidades, y en última instancia, en un prejuicio económico para la organización.

ELEMENTOS CLAVE DE LA GESTIÓN DE PROYECTOS

La gestión de proyectos abarca todo el proceso de desarrollo, desde la concepción hasta la finalización, y requiere la integración efectiva de un conjunto de elementos para asegurar el éxito en términos de tiempo, costo, calidad y satisfacción del cliente. Algunos de estos elementos son:

- Métricas
- Estimaciones
- Planificación temporal
- Planificación organizativa
- Análisis de riesgos
- Seguimiento y control

MÉTRICAS

- **Entender**
- **Controlar**
- **Evaluar**
- **Mejorar**

Las métricas son una tecnología fundamental para el desarrollo y el mantenimiento del software. Su objetivo principal es comprender y controlar los procesos durante el desarrollo y mantenimiento, además de mejorar los diversos aspectos del proyecto y evaluar su calidad. Las métricas brindan una visión cuantitativa que permite medir y analizar diferentes aspectos del software y su proceso de creación, lo que contribuye a una toma de decisiones más informada y a la optimización continua del proyecto. Son herramientas para que profesionales e investigadores tomen mejores decisiones. Sirven para medir procesos, proyectos y productos de software, entre otros, y garantizan la calidad.

MEDIDA	**MEDICIÓN**
Es una indicación cuantitativa de alguna propiedad o atributo, como tiempo, recursos, líneas de código, etc. Se utiliza para cuantificar la magnitud de un aspecto particular de un sistema, componente o proceso.	Es el proceso de determinar una medida a través de cálculos y análisis. Implica el acto de evaluar y registrar valores cuantitativos para obtener información sobre un atributo específico.
MÉTRICA	**INDICADOR**
Es un conjunto de reglas, fórmulas o criterios que se aplican a una medida para obtener información específica sobre el objeto que se está evaluando. Las métricas son herramientas que te obtener información más significativa a partir de las medidas.	Un indicador es un elemento de información que se forma a partir una combinación de métricas y establece una pauta de cuál es la medida correcta para un atributo. Esto permite al gestor del proyecto ajustar el producto, proceso o proyecto en función de la información brindada por el indicador

La medida es un objeto intangible, la medición es una acción. El resultado de una medición, la medida, da un valor cuantitativo concreto tomado en un punto particular. La métrica, en cambio, da un valor cuantitativo en base al análisis y recopilación de varias medidas de un atributo específico. De igual modo, el indicador también se obtiene en base a una recopilación de resultados de métrica.

MÉTRICAS DEL PROYECTO

Las métricas del proyecto suelen tener propósitos tácticos porque se centran en medir el progreso y el rendimiento inmediato de un proyecto específico. Estas métricas son utilizadas para tomar decisiones operativas y ajustar el curso del proyecto en función de la información actual. Su uso recae en distintos sectores:

Estado	Evaluaciones del progreso del proyecto en curso.
Uso del tiempo	Ajustes en el calendario para evitar demoras.
Tareas	Ajustes del flujo de trabajo y tareas.
Recursos humanos	Determinación de la cantidad de personas requeridas.
Equipo	Evaluaciones de habilidad del equipo.
Riesgos	Rastreo y cuantificación de riesgos.
Errores	Identificación de errores a lo largo del proyecto.
Áreas problemáticas	Identificación de áreas problemáticas.

MÉTRICAS DEL PROCESO

Las métricas del proceso son medidas cuantitativas utilizadas para evaluar y analizar diversos aspectos del proceso de desarrollo de software. Estas métricas brindan información objetiva sobre la eficiencia, efectividad y calidad del proceso, lo que permite identificar áreas de mejora, tomar decisiones informadas y optimizar la gestión del desarrollo.

Un proceso se orienta hacia objetivos a largo plazo, lo cual hace imperativo tener una estrategia que oriente las acciones y decisiones tanto en el ámbito de la organización como en proyectos específicos. Estos propósitos estratégicos son fundamentales para definir prioridades, alinear recursos y tomar decisiones que contribuyan al logro de metas y resultados de amplio alcance. Los dos propósitos estratégicos clave en este contexto son:

- **Recopilación:** A lo largo de varios proyectos en un período de tiempo se recopilan medidas y métricas diversas.
- **Intención:** En base a las medidas, métricas y experiencia se generan indicadores para orientar al equipo.

MÉTRICAS DEL PROCESO | USOS

Las métricas del proceso se utilizan para evaluar y mejorar la eficiencia, calidad y efectividad de los procedimientos y flujos de trabajo en una organización. Ayudan a identificar áreas de mejora y a tomar decisiones estratégicas para optimizar los procesos a largo plazo, lo que conduce a una mayor eficiencia operativa, reducción de costos y mejora continua de la calidad del producto o servicio.

Conocimiento	Fomentar el sentido común, sensibilidad y consciencia organizacional en el equipo, brindándoles información sobre el proceso.
Retroalimentación	Mantener una comunicación constante con el equipo acerca de todos los aspectos involucrados en el proceso, ofreciendo una visión integral y continua de cada etapa.
Evaluaciones	Utilizar métricas para evaluar el rendimiento del equipo sin recurrir a críticas destructivas, permitiendo identificar el desempeño de los individuos y sus cualidades.
Claridad	Establecer metas y métricas con precisión y transparencia.
Ampliación	Evitar la exclusión de métricas y, en cambio, considerar la ampliación de las métricas utilizadas.

MÉTRICAS DEL PROCESO | MEDICIÓN

La única forma de mejorar un proceso es medir atributos internos de este, desarrollar un conjunto de métricas en base a estos atributos y luego usarlos para lograr indicadores que llevaran a la mejora del proceso. Por ejemplo, errores detectados antes de la liberación del software, cantidad de productos de trabajo entregados, esfuerzo humano invertido, cumplimiento del calendario establecido, tiempo y esfuerzo invertido en actividades propias del proceso, entre otros.

MÉTRICAS DEL PRODUCTO

Las métricas del producto son medidas cuantitativas utilizadas para evaluar y cuantificar diversos aspectos del software o producto en sí mismo. Estas métricas proporcionan información objetiva sobre características y cualidades específicas del producto, como su funcionalidad, rendimiento, calidad y cumplimiento de requisitos no funcionales. Se utilizan para analizar y mejorar la calidad del producto, identificar posibles deficiencias y tomar decisiones informadas durante su desarrollo y mantenimiento. A estas métricas se las divide en métricas dinámicas y métricas estáticas.

ESTÁTICAS	DINÁMICAS
Realizadas en función de las representaciones del sistema.	Realizadas durante la ejecución del programa.
Evalúan la complejidad, comprensibilidad y mantenibilidad del software.	Evalúan la eficiencia y fiabilidad del software.
Relacionadas de manera indirecta con los atributos de calidad del software.	Relacionadas con las características de calidad del software.

MÉTRICAS DEL PRODUCTO | ESTÁTICAS

Una métrica estática del producto se refiere a una medida cuantitativa que se obtiene a partir del análisis de las representaciones estáticas del software, como el código fuente, la arquitectura, los diagramas y la documentación. Estas métricas se utilizan para evaluar aspectos relacionados con la estructura, el diseño y la complejidad del software, y proporcionan información sobre la calidad, mantenibilidad y comprensibilidad del producto.

ALGUNAS MÉTRICAS ESTÁTICAS	
Fan-in/Fan-out	Fan-in es una medida del número de funciones que llaman a una función X. Fan-out es el número de funciones que son llamadas por una función X. Cuando una función tiene un alto Fan-in, significa que está fuertemente acoplada al resto del diseño. Un alto Fan-out sugiere que la función podría tener una alta complejidad por la coordinación en cada llamada.
Longitud del código	Medida del tamaño del programa. Entre más extenso sea el código, es más complejo y susceptible a errores.
Complejidad ciclomática	Medida de la complejidad del control de un programa. Está relacionada con la comprensión del programa.
Longitud de los identificadores	Medida de la longitud promedio de los diferentes identificadores en un programa. Entre más grande sea su longitud, serán más fáciles de distinguir y por lo tanto el programa será más comprensible.
Profundidad del anidamiento de las condiciones	Medida de profundidad de anidamiento de las instrucciones condicionales <if> en un programa. Un gran anidamiento dificulta la comprensión y aumenta la probabilidad de errores.
Índice de Fog	Medida de la longitud promedio de las palabras y las frases en los documentos. Cuanto más grande sea el índice Fog, el documento será más difícil de comprender.

MÉTRICAS DEL PRODUCTO | ESTÁTICAS | POST MORTEN

Las métricas Post mortem se refieren al análisis y evaluación de métricas después de que un proyecto o proceso haya sido completado. Estas métricas se utilizan para analizar retrospectivamente el desempeño, el éxito y los resultados de un proyecto una vez que ha finalizado. La idea es obtener patrones, tendencias e información valiosa sobre lo que funcionó bien, lo que no funcionó y las lecciones aprendidas para futuros proyectos.

Utilidad

- **Línea base:** Establece una línea base para métricas futuras. Al finalizar un proyecto, se crea un registro que puede servir como punto de partida para proyectos posteriores que requieran mediciones similares.
- **Mantenimiento:** Facilita el mantenimiento al conocer la complejidad lógica, el tamaño y el flujo de información. Esto, entre otras cosas, permite identificar módulos críticos y áreas de enfoque.
- **Reingeniería:** Contribuye a los procesos de reingeniería al proporcionar información valiosa sobre el rendimiento y la estructura del proyecto finalizado.

MÉTRICAS DEL PRODUCTO | ESTÁTICAS | LDC

En base al tamaño de un producto es posible obtener diversas métricas para su evaluación. La métrica LDC (Líneas de Código) se refiere a la cantidad total de líneas de código fuente escritas para desarrollar un software o sistema. Es la métrica más comúnmente utilizada para evaluar el tamaño y la complejidad de un proyecto de programación, abarcando líneas de código ejecutable, comentarios y líneas en blanco. Los resultados finales de esta métrica se obtienen al concluir el proyecto, proporcionando una visión integral y precisa del mismo.

Esta métrica es objeto de debates ya que se la considera Post Mortem. Si se poseen registros de proyectos anteriores finalizados, que sean similares al proyecto actual, las métricas LDC pueden aprovechar estos para crear indicadores o tablas de datos que se orienten hacia el tamaño, lo que contribuye a su evaluación y mejora.

Los resultados derivados de esta métrica permiten analizar diversos aspectos:

- **Tiempo:** La relación del producto con el tiempo de desarrollo.
- **Personas:** La relación del producto con los recursos humanos involucrados, la cantidad de personal y la asignación de tareas.

MÉTRICAS DEL PRODUCTO | ESTÁTICAS | LDC | DERIVADAS

En base a la métrica LDC se pueden obtener otras métricas relacionadas al tamaño como la productividad, la calidad, la documentación y el costo. Esto permiten evaluar diferentes aspectos del proceso de desarrollo y su resultado final.

KLDC - miles de líneas de código

- **Productividad:** KLDC/persona-mes
- **Calidad:** errores / KLDC
- **Documentación:** n° de páginas/KLDC
- **Costo:** costo del proyecto / KLDC

- **Productividad:** Esta métrica mide la cantidad de código producido por cada unidad de esfuerzo humano durante un mes. Un valor alto de productividad indica que se está generando más código con menos esfuerzo.
- **Calidad:** Esta métrica proporciona información sobre la calidad del código en términos de errores por cantidad de líneas de código escritas. Una menor cantidad de errores por KLDC sugiere una mayor calidad y menor propensión a defectos en el código.
- **Documentación:** Indica cuánta documentación se ha generado en relación con la cantidad de código desarrollado. Una alta proporción de documentación por KLDC puede indicar un enfoque detallado en la documentación.
- **Costo:** Esta métrica permite entender el costo asociado a cada unidad de código desarrollada. Un costo alto por KLDC podría indicar ineficiencias en el proceso de desarrollo o el uso de recursos.

Se pueden comparar los resultados con los de otros proyectos ya concluidos. De ese modo, es posible reconocer cuál tiene mayor calidad, en base a sus errores/KLDC, cuál tiene mayor costo por línea, en base al costo del proyecto/KLDC, y cual tiene menor productividad por persona, en base a KLDC/personas-mes.

MÉTRICAS DEL PRODUCTO | ESTÁTICAS | LDC | PROPUESTA FENTON/PFLEEGER

Debido a que el manejo de líneas en blanco y comentarios suelen interferir en los cálculos reales del tamaño de un código, Fenton y Pfleeger proponen que, al realizar medidas relacionadas con el tamaño del código, se excluyan los comentarios presentes en el código fuente. Para mejorar la precisión en la evaluación de la longitud real del programa, se propone distinguir entre las líneas de código que son comentarios y las que no.

En esta técnica se cuantifica la cantidad de líneas que consisten en comentarios dentro del código fuente (CLDC) y luego, por otro lado, la cantidad de líneas de código fuente que no son comentarios (NCLDC). Se suman ambas y como resultado se obtiene la longitud real del código (LDC).

Fórmula:

LDC = NCLDC + CLDC
DC = CLDC/LDC

Bajo este enfoque es posible generar otras métricas adicionales basadas en la relación entre las líneas de código y los comentarios. Por ejemplo, CLDC/LDC mide la densidad de comentarios en relación con las líneas de código totales.

MÉTRICAS DEL PRODUCTO | ESTÁTICAS | PUNTO FUNCIÓN

La Métrica de Punto Función desarrollada por Allan Albrecht en 1978, mide la cantidad de funcionalidad de un sistema descripto en una especificación. Los puntos de función se utilizan para evaluar cuántas unidades de funcionalidad ofrece un sistema en términos de las operaciones que realiza.

Esta métrica se centra en evaluar la cantidad y la complejidad de las operaciones que el software debe realizar desde la perspectiva del usuario. No se basa en la cantidad de líneas de código o en la complejidad algorítmica interna del software. Dado que los puntos de función se basan en los requisitos y la funcionalidad prevista, se pueden calcular durante las etapas de planificación y diseño. Esto brinda una visión temprana del tamaño y la complejidad del software, lo que puede ser valioso para la gestión del proyecto y la toma de decisiones. Por estas razones, es considerada una métrica temprana.

MÉTRICAS DEL PRODUCTO | ESTÁTICAS | PUNTO FUNCIÓN | TÉCNICA

La técnica utilizada consiste en poner en una tabla a las cinco unidades de funcionalidad (entradas, salidas, consultas, almacenamientos internos e interfaces externas) y asignarles una cantidad de puntos a cada. Estos puntos se asignan según la complejidad de los datos que manejan y de los procesos que se realizan sobre ellos.

La tabla posee un factor de ponderación que está divido en tres categorías: simple, mediano o complejo. Según la unidad, estas categorías toman valores diferentes y están definidos por la organización o el equipo de desarrollo. Para cada unidad funcional, se deben tomar los puntos cuantificados y multiplicarlos por el valor de su categoría en la tabla. El nivel del factor de ponderación de cada unidad también lo decide cada organización. Finalmente se deben sumar todos los puntos ponderados.

UNIDAD FUNCIONAL			FACTOR DE PONDERACIÓN				
			Simple	Mediano	Complejo		
Entradas	P	*	3	4	6	=	t
Salidas	P	*	4	5	7	=	t
Consultas	P	*	3	4	6	=	t
Almacenamientos internos	P	*	7	10	15	=	t
Interfaces externas	P	*	5	7	10	=	t
TOTAL DE PUNTOS						=	TP

- **Fi:** Valor de ajuste
- **i:** Pregunta

0<= Fi <=5 i = [1 … 14]

Fórmula:

PF = TOTAL * [0.65 + 0.01 * SUM(Fi)]

Para calcular el Punto de Función, al total obtenido se lo multiplica por el cálculo de distintos valores de ajustes. Para ello se evalúan 14 preguntas (arbitrarias) que reflejen la complejidad del sistema evaluado, y a cada una se le asigna un valor de ajuste de entre 0 a 5 según su impacto. Estos valores y la cantidad de preguntas son de carácter general, al igual que los valores 0.65 y 0.01.

VALORES DE AJUSTE	No Influencia	Incidental	Moderado	Medio	Significativo	Esencial
F	0	1	2	3	4	5

La combinación de 0.65 y 0.01 se utiliza para equilibrar y ajustar la puntuación de puntos de función en función de la complejidad percibida del sistema, según los valores de las 14 preguntas. Estos valores pueden variar en diferentes métodos de cálculo de Puntos de Función, pero la idea general es proporcionar una medida más precisa de la funcionalidad del software teniendo en cuenta su complejidad y características específicas.

El resultado final es la cantidad total de puntos de función, que es una medida del tamaño funcional del sistema. Esta medida representa la cantidad de trabajo funcional que el sistema realiza desde la perspectiva del usuario.

MÉTRICAS DEL PRODUCTO | ESTÁTICAS | PUNTO FUNCIÓN | DERIVADAS

A partir de la métrica de punto función, se pueden derivar otras métricas que proporcionan información valiosa sobre diferentes aspectos del desarrollo y la calidad del software.

- **Productividad:** PF/persona-mes
- **Calidad:** errores / PF
- **Costo:** costo del proyecto / PF

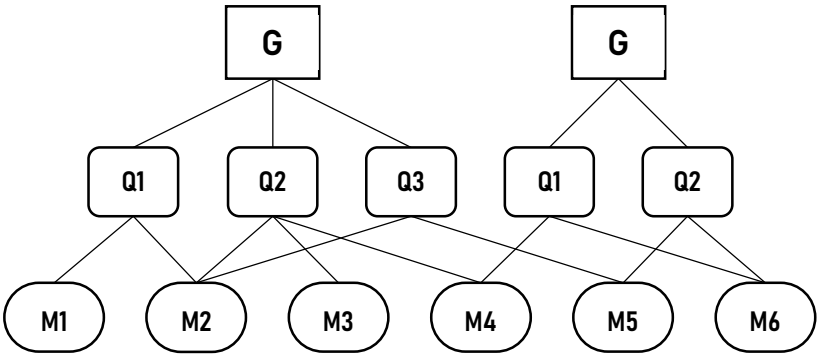
- **Productividad:** Es una medida de eficiencia laboral y proporciona una idea de cuánta funcionalidad se está produciendo por cada persona a lo largo de un mes.
- **Calidad:** Es una medida que indica la proporción de errores por unidad de funcionalidad, lo que puede ser un indicador importante de la calidad del código y la atención a la prueba y corrección de errores.
- **Costo:** Es una medida que indica cuánto ese está gastando para desarrollar cada unidad funcional del software. Puede ser útil para el seguimiento del presupuesto y la eficiencia financiera.

MÉTRICAS DEL PRODUCTO | ESTÁTICAS | GQM

El enfoque GQM (Goal, Question, Metric), desarrollado por Victor Basili, es una metodología ampliamente utilizada para diseñar métricas propias que miden metas específicas dentro de un proyecto, en cualquier parte del proceso de desarrollo, y en base a una serie de preguntas relacionadas a objetivos concretos. Estas métricas proporcionan información puntual sobre el rendimiento y el cumplimiento de objetivos, permitiendo mejorar la calidad del producto.

MÉTRICAS DEL PRODUCTO | ESTÁTICAS | GQM | ESTRUCTURA

1. **Nivel Conceptual - Objetivo (Goal):**
   Se establece un objetivo claro y concreto que se busca lograr en el proyecto. Este objetivo debe ser relevante para la mejora general y alineado con los objetivos del proyecto.
2. **Nivel Operativo - Pregunta (Question):**
   A partir del objetivo, se formulan preguntas específicas y medibles que guían hacia la información necesaria para evaluar el progreso. Estas preguntas desglosan el objetivo en aspectos tangibles. Las respuestas a estas preguntas se expresan en términos cuantificables, como sí-no, verdadero-falso o 0...8, lo que proporciona una evaluación clara y objetiva.
3. **Nivel Cuantitativo - Métrica (Metric):**
   Basándose en las preguntas formuladas, se crean métricas que recolectan datos objetivos y cuantificables. Además, se incorporan indicadores específicos que permiten determinar si los valores cuantitativos obtenidos cumplen con las expectativas o no.



Objetivo	Detalle del objetivo.
Pregunta 1	Pregunta específica.
Métrica 1	Respuesta valor binario.
Métrica 2	Respuesta valor binario.
Pregunta 2	Pregunta específica.
Métrica 2	Respuesta valor binario.
...	...

Indicador	Descripción	Fórmula	Cumplimiento
Indicador 1		M2 & M3 & M4 & M5	V o F
Indicador 2		M1 & M4 & M5	V o F
...	...	...	...

- **Flexibilidad**
  El enfoque GQM puede aplicarse en cualquier etapa del proceso de desarrollo y en diferentes tipos de proyectos.
- **Personalización**
  Cada proyecto define sus propios objetivos y, en consecuencia, genera métricas específicas que se alinean con esas metas.
- **Enfoque en la mejora**
  GQM no solo mide el rendimiento, sino que también tiene como objetivo impulsar mejoras continuas al proporcionar información medible sobre el progreso hacia los objetivos.

MÉTRICAS DEL PRODUCTO | ESTÁTICAS | ORIENTADAS A OBJETOS

Las métricas orientadas a objetos son un conjunto de medidas cuantitativas que se aplican específicamente a sistemas de software desarrollados utilizando la programación orientada a objetos (POO). Estas métricas se centran en evaluar las características, relaciones y estructuras de los objetos, clases y componentes en el software, con el objetivo de medir la calidad, la complejidad y otros aspectos relevantes del diseño y la implementación.

ALGUNAS MÉTRICAS ESTÁTICAS ORIENDAS A OBJETOS	
Métodos ponderados por clase (weighted methods per class, WMC)	Número de métodos en cada clase, ponderado por la complejidad de cada uno. Cuánto más sea el valor para esta métrica, más compleja será la clase.
Profundidad de árbol de herencia (depth of inheriance tree, DIT)	Número de niveles discretos en el árbol de herencia atributos y operaciones (métodos) de las superclases. A mayor profundidad, más complejidad.
Número de hijos (number of children, NOC)	Número de subclases inmediatas en una clase. Mide la amplitud de una jerarquía de clase. A mayor amplitud, mayor reutilización. Podría significar también mayor esfuerzo para validar las clases debido a las dependencias.
Acoplamiento entre clases de objetos (coupling between object classes, CBO)	Medida de acoplamiento entre clases. Un alto CBO indica clases estrechamente dependientes, cambiar una clase puede afectar a otras clases del programa.
Respuesta por clase (response for a class, RFC)	Número de métodos que potencialmente podrían ejecutarse en respuesta a un mensaje recibido por un objeto de dicha clase. A mayor RFC, más compleja será la clase y más proclive a errores.
Falta de cohesión en métodos (lack of cohesion in methods, LCOM)	Se calcula al considerar pares de métodos en una clase. LCOM es la diferencia entre el número de pares de métodos sin compartir atributos y el número de pares de métodos con atributos compartidos.

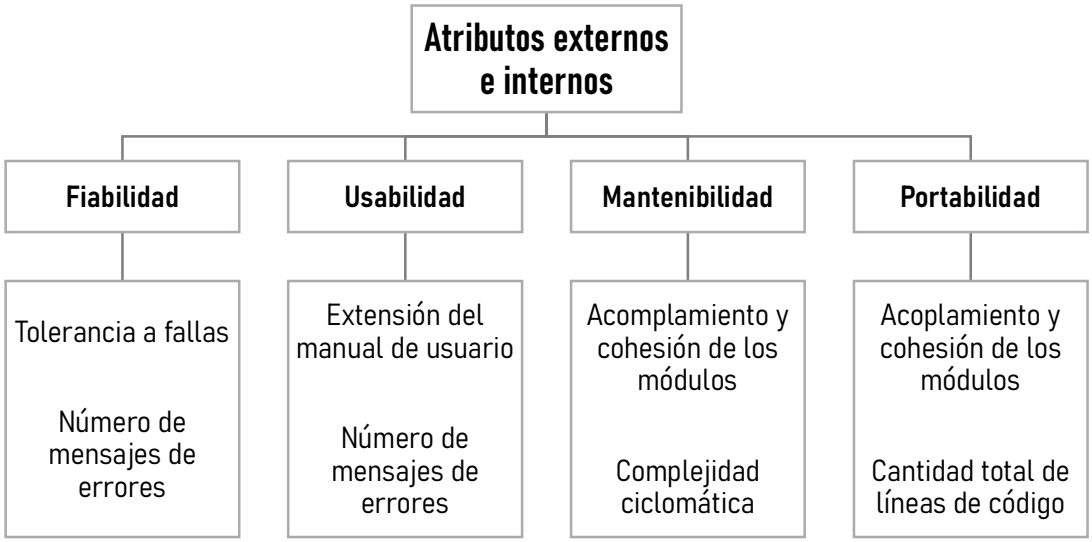
MÉTRICAS DEL PRODUCTO | DINÁMICAS

Las métricas dinámicas del producto se relacionan con las características de calidad del software y son indicadores de rendimiento y comportamiento que se generan y evalúan en tiempo real. Son esenciales para comprender el rendimiento en tiempo real y permiten tomar decisiones informadas sobre mejoras y ajustes inmediatos.

MÉTRICAS DEL PRODUCTO | DINÁMICAS | ATRIBUTOS DE CALIDAD

Los atributos que miden la calidad producto se pueden diferenciar entre atributos de calidad externos (mediante métricas dinámicas) y atributos internos (mediante métricas estáticas). Se buscan que los atributos internos contribuyan a cumplir los atributos externos. Los atributos de calidad externos están relacionados con los requerimientos no funcionales del producto y otros atributos como la fiabilidad, la portabilidad, la mantenibilidad, entre otros.

Por su parte, los atributos internos están dentro de las métricas estáticas y son características o propiedades que no son directamente observables por los usuarios finales, sino que están relacionadas con la estructura, el diseño, el código o los componentes internos del producto. Ejemplos de atributos internos incluyen el modularidad del código, la claridad de la estructura, el tamaño del programa en líneas de código y el número de mensajes de error.



Para medir la calidad en base a un atributo externo, antes se debe vincular a ese atributo con los atributos internos con los que está relacionado. Por ejemplo, para medir la eficiencia, se compararía a este atributo con la métrica del producto que mide el tiempo promedio de respuesta de una funcionalidad. Una vez que se recopilan los datos, se realizan los cálculos necesarios y se analizan con los indicadores resultantes.

ESTIMACIONES

¿Qué son?

Las estimaciones son técnicas que permiten asignar un valor aproximado a una cantidad, como tiempo, recursos, costos o cualquier otro aspecto. Se utilizan para proporcionar una idea general y rápida de lo que se espera.

¿Cómo usarlas?

Las estimaciones se emplean cuando no es posible aplicar métricas debido a la falta de datos o a la naturaleza de la tarea. Son útiles en situaciones donde se necesita una idea rápida y aproximada de una cantidad.

¿Qué podemos estimar?

Las estimaciones pueden aplicarse a diversos aspectos, como recursos, proyectos, costos, tiempo y más. Pueden ayudar a planificar y tomar decisiones en una etapa temprana.

¿Qué tener en cuenta?

Es importante entender que las estimaciones no son una ciencia exacta. Pueden ser precisas o no, y factores imprevistos pueden alterarlas. Las estimaciones también requieren experiencia para realizarlas de manera efectiva.

¿Qué factores influyen?

Varios factores influyen en las estimaciones, incluida la complejidad del proyecto, el tamaño del producto o tarea, la estructuración del proyecto y otros aspectos relacionados. Evaluar cuidadosamente estos factores es crucial para generar estimaciones más realistas.

¿En qué se diferencian con las métricas?

A diferencia de las métricas, que son medidas seguras y precisas, las estimaciones son valores aproximados y más subjetivos. Las métricas se basan en datos concretos, mientras que las estimaciones son más flexibles y se utilizan en situaciones en las que la información precisa es difícil de obtener.

ESTIMACIÓN DE RECURSOS

Recursos Humanos (Personal)

- **Cantidad:** Determinar cuántas personas serán necesarias para llevar a cabo el proyecto. Esto puede incluir desarrolladores, diseñadores, gerentes, etc.
- **Habilidades:** Identificar las habilidades y competencias específicas requeridas para cada rol en el proyecto. Esto asegura que el equipo tenga las capacidades necesarias.
- **Duración:** Estimar el tiempo que cada miembro del equipo necesitará para completar sus tareas. Esto contribuye a planificar el cronograma general.
- **Ubicación:** Definir dónde se encuentra físicamente el equipo de trabajo o si habrá algún componente de trabajo remoto.

Entorno

- **Herramientas de Software:** Identificar y estimar el uso de herramientas de software necesarias para el desarrollo y la gestión del proyecto.
- **Hardware:** Prever el hardware requerido, como servidores, dispositivos móviles, computadoras, etc.
- **Recursos de Red:** Evaluar la infraestructura de red necesaria para la colaboración y el acceso a recursos compartidos.

Software Reutilizable

- **Componentes COTS:** Evaluar adquisición de componentes de software que proveen una funcionalidad específica y están disponibles en el mercado para ser adquiridos e integrados dentro del sistemas de software.
- **Componentes Nuevos:** Identificar cualquier nuevo componente de software que necesite ser desarrollado específicamente para el proyecto.
- **Componentes de Experiencia Completa:** Incluir componentes que ya se hayan utilizado o desarrollado en proyectos anteriores y que se puedan aprovechar nuevamente.
- **Componentes de Experiencia Parcial:** Considerar componentes que se hayan desarrollado parcialmente y requieran ajustes y pruebas adicionales.

ESTIMACIÓN DE TIEMPO

Las estimaciones de tiempo son un componente fundamental en la gestión de proyectos, ya que permiten prever la duración de las tareas y el proyecto en su conjunto. Estas estimaciones proporcionan una idea de cuánto tiempo se necesitará para completar cada fase, tarea o entrega. Normalmente se estima el tiempo con la técnica de planificación temporal.

P (People)
E (Environment)
R (Resources)
P (Priority)
S (Scope)
D (Dependencies)
G (Goals)

Al utilizar un enfoque estructurado como PERPSDG, se pueden realizar estimaciones más sólidas y desarrollar un cronograma que sea realista y adaptable a las circunstancias cambiantes del proyecto. Tomando esto en cuenta, realizar las estimaciones de tiempo junto con la planificación temporal, resulta más realista y preciso. Al considerar cada elemento de PERPSDG, se pueden prever obstáculos potenciales y tomar decisiones informadas para garantizar que el proyecto se complete dentro del plazo establecido.

ESTIMACIÓN DE COSTOS

Costos de Esfuerzos

Estos costos están relacionados con los recursos humanos, específicamente con los salarios y remuneraciones de los miembros del equipo de desarrollo y otros profesionales involucrados en el proyecto.

Hardware y Software

En este parámetro se incluyen los costos asociados con la infraestructura tecnológica. Esto abarca el hardware necesario, como servidores, dispositivos móviles y equipos de cómputo, así como el software utilizado para el desarrollo, las herramientas de gestión del proyecto y otras soluciones tecnológicas. También se incluyen los costos de internet, plataformas, licencias y mantenimiento de software.

Viajes

Estos costos involucran los gastos relacionados con la movilidad del equipo, como los viajes necesarios para reuniones, colaboraciones o para estar presente en diferentes ubicaciones.

ESTIMACIÓN DE COSTOS | FIJACIÓN DE PRECIO Y RELACIÓN PRECIO-COSTO

La fijación de precios y la relación entre precio y costo son elementos críticos en cualquier proyecto o negocio. El costo se refiere a los gastos y recursos necesarios para completar un proyecto, mientras que el precio es la cantidad que se cobra al cliente por el producto o servicio entregado. Aunque el costo establece una base financiera, el precio puede variar y estar cercano o alejado del costo real.

Oportunidades de Mercado

Evaluar la demanda y la competencia en el mercado para determinar un precio que sea atractivo para los clientes y competitivo.

Incertidumbre en Estimaciones de Costos

Reconocer que las estimaciones de costos pueden ser inexactas y ajustar el precio para mitigar riesgos financieros.

Términos Contractuales

Considerar los acuerdos contractuales y compromisos asumidos con los clientes que puedan influir en la fijación de precios.

Volatilidad de Requerimientos

Anticipar cambios en los requisitos del proyecto y cómo estos pueden afectar los costos y, por lo tanto, el precio.

Salud Financiera

Evaluar la situación financiera de la empresa y determinar si el precio cubre los costos y permite la rentabilidad.

Otros Factores:

- **Intereses de la Empresa:** Evaluar cómo el precio afecta los objetivos y estrategias de la empresa.
- **Riesgos:** Ajustar el precio para acomodar riesgos potenciales y proteger la viabilidad financiera.
- **Tipo de Contrato:** El tipo de contrato (fijo, por hora, por resultado) puede afectar cómo se establece el precio.

La fijación de precios implica un equilibrio entre los costos, la rentabilidad y la competitividad en el mercado. Las decisiones de fijación de precios deben considerar tanto los intereses de la empresa como las necesidades y expectativas de los clientes. Todo esto puede hacer que el precio suba o baje.

TÉCNICAS DE ESTIMACIÓN

Las técnicas de estimación son herramientas valiosas en la planificación de proyectos, ya que permiten prever con mayor precisión aspectos como tiempo, costos y recursos.

Juicio Experto

En esta técnica, se consulta a varios expertos en el campo relevante para obtener sus opiniones y estimaciones. Los expertos estiman, comparan y discuten entre sí para llegar a una solución conjunta.

Técnica Delphi

La técnica Delphi involucra a un grupo de expertos que estiman de manera anónima. En cada ronda, se recopilan las estimaciones individuales y se proporcionan a todos los participantes sin revelar la identidad de los demás. Luego, los expertos vuelven a estimar, teniendo en cuenta las opiniones de los demás. Este proceso se repite en sucesivas rondas hasta que se alcanza un consenso. La anonimidad permite evitar la influencia de la opinión de otros participantes.

División de Trabajo Jerárquica

Esta técnica implica la consulta a personas en diferentes niveles jerárquicos de la organización. Comienza consultando a quienes están en niveles más bajos y avanza hacia arriba en la jerarquía. Cada nivel proporciona su estimación basada en su comprensión del proyecto y de cómo se verá afectado por sus roles.

TÉCNICAS DE ESTIMACIÓN | JUICIO EXPERTO

El juicio experto es una técnica valiosa que se utiliza para evaluar insumos, detalles técnicos y aspectos de gestión en proyectos. Proporciona una perspectiva externa beneficiosa, especialmente en proyectos de alto impacto. Aunque su subjetividad puede influir, establecer pautas claras ayuda a mejorar la objetividad. Además de su aplicación principal, el juicio de expertos también es útil en situaciones clave:

- **Validación:** En proyectos de alto impacto cuando hay mucho en juego y es necesaria la validación de un experto para avanzar.
- **Pre-Planificación:** Antes de la planificación completa del proyecto.
- **Precio-Costo:** Para la gestión de costos y determinar precios de productos.
- **Riesgos:** En la gestión de riesgos.

TÉCNICAS DE ESTIMACIÓN | PLANNING POKER

El Planning Poker es una técnica colaborativa ampliamente utilizada en la estimación de esfuerzo para las historias de usuario durante un Sprint de desarrollo. Su nombre proviene del uso de cartas numeradas usualmente con valores de la serie de Fibonacci.

Pasos:

1. **Reunión del Equipo**
   Todos los miembros del equipo se reúnen alrededor de una mesa. Cada uno posee un conjunto de cartas numeradas basadas en la serie de Fibonacci.

2. **Lectura de Historias**
   El Product Owner (dueño del producto) presenta una historia de usuario al equipo. En este momento, se aclaran cualquier duda o pregunta.

3. **Selección de Cartas**
   Cada miembro del equipo elige una carta con el número que considera representa el esfuerzo requerido para completar la historia.

4. **Revelación y Comparación**
   Una vez que todos han seleccionado una carta, las cartas se revelan simultáneamente. Esta simultaneidad evita que las opiniones sean influenciadas por otros.

5. **Comparación y Justificación**
   Se comparan las cartas seleccionadas por los miembros y se brindan justificaciones para las elecciones realizadas. Si hay consenso, se asigna el valor de la carta a la historia. En caso contrario, se repite el proceso hasta llegar a un acuerdo.

Este proceso se repite con cada historia de usuario en el backlog del producto.

MODELOS EMPÍRICOS DE ESTIMACIÓN

Los modelos empíricos de estimación se aplican en proyectos de gran envergadura debido a la complejidad que estos presentan. Estos modelos proponen fórmulas desarrolladas a partir de datos recopilados para prever los costos o el esfuerzo necesario en un proyecto.

MODELOS EMPÍRICOS DE ESTIMACIÓN | COCOMO II

El modelo COCOMO II (Constructive Cost Model) es una técnica de estimación utilizada en la gestión de proyectos de desarrollo de software. Este modelo tiene como objetivo estimar el esfuerzo, el costo y el tiempo necesarios para desarrollar un sistema de software, basándose en factores específicos del proyecto y del producto. COCOMO II se basa en la recopilación y análisis de datos históricos de proyectos de software previos para establecer relaciones empíricas entre diversos factores y las medidas de esfuerzo y costo.

Este modelo vincula fórmulas principalmente sobre el tamaño del sistema, factores de producto, factores del proyecto y factores del entorno, para estimar el esfuerzo en horas/persona, el costo y la duración del proyecto.

COCOMO II evolucionó a partir de COCOMO I o COCOMO 81, que se centraba en gran medida en las líneas de código y el tamaño del sistema, así como en factores del proyecto. Con la evolución del tiempo y la adopción de metodologías ágiles, COCOMO II reconoce la dificultad de estimar las líneas de código tempranamente y considera enfoques diversos para el desarrollo, como la creación de prototipos, el desarrollo basado en componentes y el desarrollo en espiral.

Se **basa** en datos históricos de un gran número de proyectos pasados y **utiliza** fórmulas **para** relacionar el tamaño del sistema y los factores del producto, proyecto y equipo **con** el esfuerzo para realiza el sistema.

MODELOS EMPÍRICOS DE ESTIMACIÓN | COCOMO II | COMPOSICIÓN

Modelo de composición de aplicación

- **PM:** esfuerzo estimado en persona/mes
- **NAP:** total de puntos de aplicación
- **PROD:** productividad medida en puntos de objeto

Fórmula:

PM = (NAP x (1 - %reutilización / 100)) / PROD

Este modelo se basa en el número de puntos de aplicación y se utiliza para sistemas desarrollados con lenguajes dinámicos y programación de base de datos. Se aplica a sistemas creados a partir de componentes de reutilización y proyectos de creación de prototipos. Estima el esfuerzo en función de puntos de aplicación (o puntos objeto), que es una medida ponderada de varios factores del sistema, como pantallas, informes y módulos.

Modelo de diseño temprano

Basado en el número de puntos de función, se utiliza para estimar el esfuerzo inicial en función de los requerimientos del sistema y el diseño. Es útil en las primeras etapas del proyecto, antes de tener un diseño arquitectónico detallado. Proporciona una estimación rápida y aproximada del esfuerzo necesario.

- **PMauto:** esfuerzo estimado en persona/mes
- **ASLOC:** nº de líneas de códigos en componentes a reutilizar
- **PROD:** porcentaje de código adaptado que se genera automáticamente
- **ATROD:** productividad de los ingenieros que integran el código

Fórmula:

PMauto = (ASLOC x AT/100)/ATPROD

Modelo de reutilización

- **PM:** esfuerzo estimado en persona/mes
- **A:** 2.94 (según Boehm)
- **Tamaño:** KLDC totales
- **B:** esfuerzo requerido conforme aumenta el tamaño del proyecto (de 1.1 a 1.24)
- **M:** n atributos de proyecto y proceso que aumentan o disminuyen la estimación.

Fórmula:

PM = A x Tamaño^B x M

Este modelo se basa en el número de líneas de código de reutilización o generadas automáticamente. Se utiliza para estimar el esfuerzo necesario para integrar y adaptar estos componentes en el proyecto.

Modelo de Post-Arquitectura

Basado en el número de líneas de código fuente, se emplea cuando existe un diseño arquitectónico inicial. Se usa para estimar el esfuerzo de desarrollo basado en especificaciones de diseño del sistema. El cálculo del tamaño (KLDC) se hace en base a tres componentes:

- Líneas nuevas de código.
- Líneas equivalentes de código fuente (ESLOC) basadas en reutilización.
- Líneas modificadas debido a cambios en los requerimientos.

- **PM:** esfuerzo estimado en persona/mes
- **A:** 2.94 (según Boehm)
- **KLDC:** miles de líneas de código
- **B:** esfuerzo requerido conforme aumenta el tamaño del proyecto (de 1.1 a 1.24)
- **m:** 7 atributos de proyecto y proceso que aumentan o disminuyen la estimación.

Fórmula:

PM = A x KLDC^B x m

PLANIFICACIÓN TEMPORAL

La planificación temporal es un aspecto esencial en la gestión de proyectos y se refiere a la distribución efectiva del esfuerzo estimado y los recursos a lo largo del período de tiempo previsto para llevar a cabo un proyecto. Esta actividad busca establecer una secuencia lógica de tareas y asignar los recursos adecuados a cada una, con el objetivo de cumplir con los plazos y metas del proyecto de manera eficiente.

La exactitud en estas estimaciones es esencial, ya que una precisa planificación temporal resulta crucial para evitar la insatisfacción de los clientes, la generación de costos adicionales y la disminución del impacto en el mercado.

CALENDARIZACIÓN

La calendarización implica la distribución del esfuerzo estimado a lo largo del período planificado para un proyecto, asignando dicho esfuerzo a tareas específicas del desarrollo de un software. La designación de la fecha final de un proyecto puede realizarse en dos escenarios diferentes:

- **Fecha final establecida por el cliente**
  En este caso, el equipo está obligado a distribuir el esfuerzo dentro del plazo establecido por el cliente. Los recursos deben ajustarse para cumplir con esa fecha.
- **Fecha final fijada por los desarrolladores**
  En este escenario, el esfuerzo se distribuye de manera que se logre una utilización óptima de los recursos. Se determina una fecha de finalización después de un análisis exhaustivo.

Desafortunadamente, la primera opción es la más comúnmente encontrada.

CALENDARIZACIÓN | COMPONENTES

Tarea

Se refiere a una secuencia de acciones a realizar en un plazo determinado. Por ejemplo, el desarrollo de un caso de uso, la realización de una entrevista con un cliente, la validación de una función, entre otros.

Tarea crítica

Es aquella cuyo retraso genera un retraso en todo el proyecto. Por ejemplo, las tareas que están conectadas con otras y cuyo retraso podría detener la producción si no se completan a tiempo.

Hito

Un hito es un punto de referencia en el proyecto que marca la finalización de una fase importante o la consecución de un objetivo clave. Representa un logro significativo y tangible en el proyecto. Por ejemplo, la finalización de un módulo con pruebas exitosas y su documentación, o cualquier otro logro que sea objetivo, fácil de evaluar y tenga un impacto notable en el proyecto.

CALENDARIZACIÓN | PARÁMETROS DE UNA TAREA

Precursor

Se refiere a un evento o conjunto de eventos que deben tener lugar antes de que la actividad pueda dar inicio. Estos eventos previos establecen las condiciones necesarias para que la tarea pueda comenzar.

Duración

Es la cantidad de tiempo requerida para llevar a cabo la actividad en su totalidad. La duración de una tarea es un factor clave en la planificación temporal del proyecto.

Fecha de entrega

Es la fecha límite para la cual la actividad debe estar completamente finalizada. Esta fecha es crucial para mantener el proyecto en el camino correcto y asegurar la entrega oportuna. El incumplimiento con el tiempo asignado para cada tarea puede ser un factor de riesgo.

Resultado

Se refiere al logro o componente específico que se espera tener al finalizar la tarea. Puede tratarse de un hito o de un elemento concreto que esté listo para su uso o evaluación.

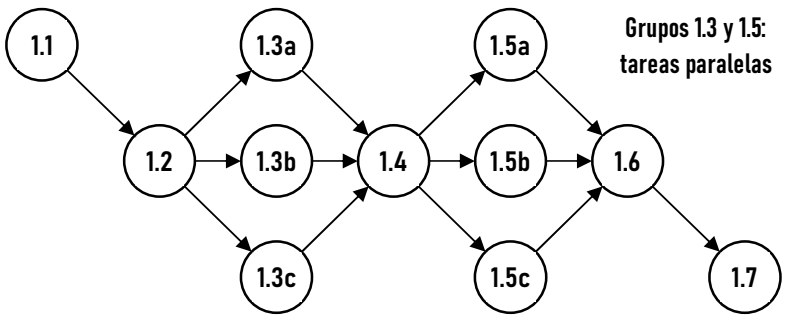
CALENDARIZACIÓN | CONJUNTO DE TAREAS

El conjunto de tareas varía según el tipo de proyecto y el nivel de rigor requerido. Los factores que influyen en el conjunto de tareas a elegir son:

- **Tamaño del proyecto:**
  Proyectos más grandes pueden requerir una subdivisión más detallada de tareas para una gestión efectiva.
- **Número de usuarios potenciales:**
  La cantidad de usuarios o partes interesadas puede influir en la complejidad de las tareas y en la comunicación requerida.
- **Criticidad del proyecto:**
  La importancia y urgencia del proyecto influyen en cómo se organizan las tareas.
- **Estabilidad de los requerimientos:**
  Si los requisitos cambian con frecuencia, es necesario adaptar la red de tareas de manera más ágil.
- **Facilidad de comunicación con el cliente/usuario:**
  La interacción con el cliente o usuarios puede afectar la planificación, especialmente en proyectos donde se requiere retroalimentación constante.
- **Madurez de la tecnología aplicable:**
  La familiaridad y madurez de las tecnologías a utilizar pueden influir en la organización de tareas.
- **Restricciones:**
  Limitaciones de recursos, presupuesto y tiempo también tienen un impacto en cómo se configura la red de tareas.

CALENDARIZACIÓN | RED DE TAREAS

La red de tareas es una representación gráfica que ilustra el flujo de las diferentes tareas desde el inicio hasta la conclusión de un proyecto. Esta representación gráfica permite apreciar la secuencia de las tareas y su interdependencia.



La red de tareas es especialmente útil para identificar la secuencia lógica en la que deben realizarse las actividades y comprender cómo las tareas se relacionan entre sí. En algunos casos, grupos de tareas pueden ejecutarse de manera paralela, lo que agiliza el proceso.

CALENDARIZACIÓN | ACCIONES ANTE RETRASOS

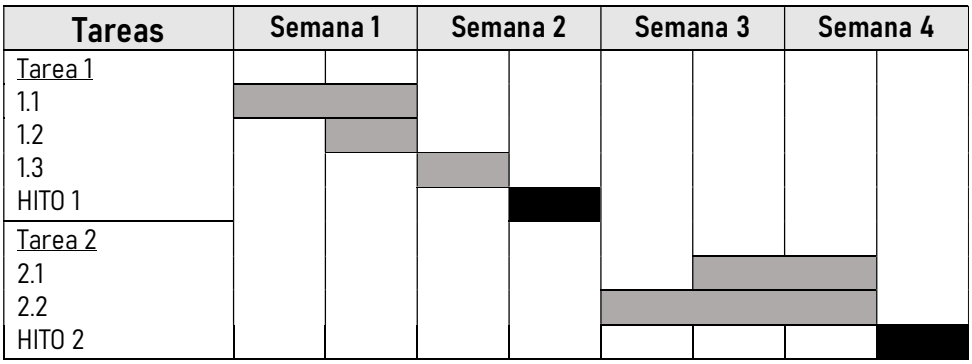
1. **Revisar el impacto sobre la fecha de entrega**
   Algunas tareas pueden ser críticas y tener un impacto significativo en el cronograma general del proyecto, mientras que otras pueden tener cierta flexibilidad.
2. **Reasignar recursos**
   Si las tareas están retrasadas, se puede considerar la posibilidad de reasignar recursos para acelerar el trabajo. Sin embargo, aumentar el número de personas involucradas no siempre resulta en un aumento directo de la productividad. De hecho, en algunos casos, puede llevar a confusiones y dificultades de coordinación.
3. **Reordenar tareas**
   En algunos casos, es posible reorganizar la secuencia de las tareas para minimizar el impacto de los retrasos. Sin embargo, esta acción debe ser cuidadosamente evaluada para asegurarse de que no cause problemas adicionales o comprometa la calidad del trabajo.
4. **Modificar entrega**
   En situaciones en las que los retrasos son inevitables y afectan significativamente el cronograma, se podría considerar la posibilidad de ajustar la fecha de entrega o la entrega parcial de resultados. Esto podría implicar negociaciones con los stakeholders del proyecto para establecer expectativas realistas en cuanto a las fechas y alcance.

MÉTODOS DE PLANIFICACIÓN TEMPORAL

Un Método de Planificación Temporal plantea una estrategia organizada y sistemática para programar y ordenar las tareas y actividades de un proyecto a lo largo del tiempo, asegurando una secuencia lógica, asignación de recursos adecuada y cumplimiento de plazos.

MÉTODOS DE PLANIFICACIÓN TEMPORAL | GANTT

El método GANTT es una herramienta que ofrece una representación visual de las tareas o actividades en un proyecto. Se estructura en dos columnas: una lista de tareas y otra con la duración, secuencialidad, dependencia y posibilidad de paralelismo entre ellas. También permite señalar aquellas tareas que culminan con un hito o logro específico. Sin embargo, es importante mencionar que este método está en desuso en la actualidad.



MÉTODOS DE PLANIFICACIÓN TEMPORAL | PERT

El PERT (Program Evaluation & Review Technique) es una técnica desarrollada en 1958-1959 para proyectos del programa de defensa del gobierno de Estados Unidos. Diseñado para proyectos extensos, se enfoca en proyectos que involucran investigación, desarrollo y pruebas, generalmente de gran envergadura.

Este método propone la creación de una red de tareas, considerando tiempos más probables, optimistas y pesimistas para completar cada tarea. Utiliza conceptos como fechas tempranas, fechas tardías y camino crítico para la gestión de proyectos.

El PERT se caracteriza por ser probabilístico, lo que significa que no se determina con exactitud el tiempo que tomará cada tarea. En cambio, se manejan intervalos de tiempo basados en los diferentes escenarios posibles. Esto es especialmente útil en proyectos complejos donde las incertidumbres y variabilidades son altas.

MÉTODOS DE PLANIFICACIÓN TEMPORAL | CMP

El CPM (Critical Path Method) es una técnica desarrollada entre 1956 y 1958 para dos empresas estadounidenses. Se aplica en proyectos y en situaciones donde las estimaciones tienen poca incertidumbre.

Este método propone la creación de una red de tareas y, a diferencia de PERT, se enfoca en proyectos donde las estimaciones de tiempo son más concretas. Además de la red, se trabaja con tiempos de inicio temprano y tiempos de inicio tardío para cada tarea.

El CPM es considerado determinístico, ya que busca determinar los tiempos precisos de inicio y finalización de cada tarea en función de sus dependencias y duraciones.

En la actualidad, se ha fusionado lo más beneficioso de los métodos PERT y CPM en una sola técnica conocida como Método del Camino Crítico o PERT-CPM. Este enfoque integrado combina las fortalezas de ambas metodologías para brindar una herramienta más completa y versátil en la planificación y gestión de proyectos.

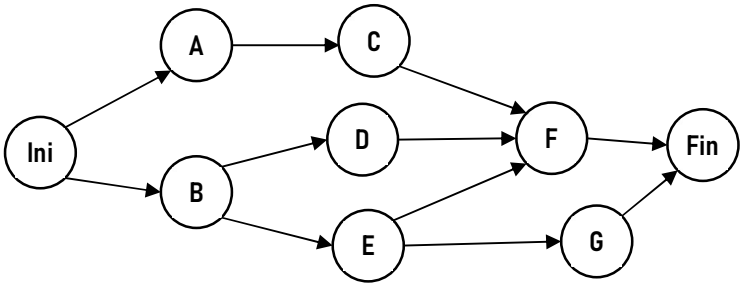
El Camino Crítico se compone de las tareas que no tienen margen de tiempo adicional para retrasos, es decir, cualquier demora en una tarea crítica retrasará la finalización del proyecto en su totalidad. Por lo tanto, identificar el Camino Crítico es esencial para la gestión efectiva del tiempo y los recursos en un proyecto.

1. **Establecer lista de tareas**
   Enlistar todas las tareas que conforman el proyecto. Esto ayuda a tener una visión completa de lo que se debe realizar.

2. **Fijar dependencia y duración**
   Crear una tabla para definir las relaciones de dependencia entre las tareas, es decir, qué tareas deben completarse antes de que otras puedan comenzar. También se debe asignar las duraciones estimadas para una en unidades de tiempo como horas, minutos, días o meses. Así, en la primera columna se enlistarán las tareas, en la segunda las dependencias, y en la tercera la duración de cada una.

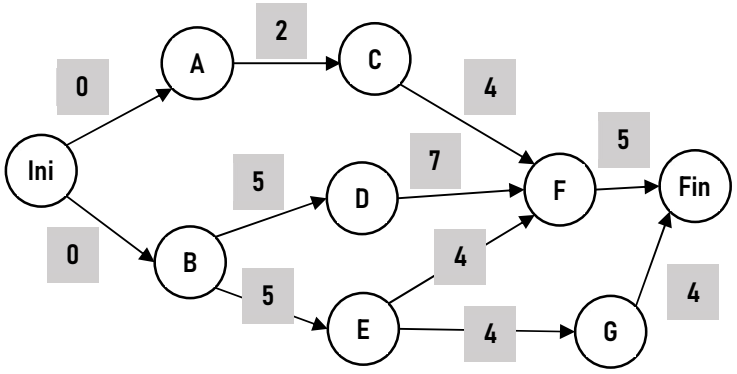
Tarea	Predecesor	Duración
A	-	2
B	-	5
C	A	4
D	B	7
E	B	4
F	C-D-E	5
G	E	4

3. **Construir la red**
   Crear un diagrama de red que ilustre las tareas y sus relaciones de dependencia. En el caso de haber más de una tarea inicial (que no tiene predecesoras), se asigna un nodo de inicio "vacío" que apunta a estas tareas sin dependencias. También se debe asignar un nodo fin.



4. Numerar los nodos y flechas

Asignar números únicos a cada nodo en la red, lo que facilita la identificación y referencia. El valor de cada flecha que emerge de un nodo está dado por la duración de la tarea del nodo en el que se originan. Si existe un nodo de inicio vacío, el valor de su fecha será 0.



En este ejemplo sólo se numeran las flechas, los nodos se identifican con las letras

5. Calcular fechas tempranas y tardías

Calcular las fechas tempranas y tardías de inicio y finalización para cada tarea en función de sus dependencias y duraciones. Las fechas tempranas definen la fecha más anticipada en la que una tarea puede comenzar o terminar sin retrasar el proyecto completo. Las fechas tardías son las fechas límites en las que una tarea puede comenzar o terminar sin retrasar el proyecto completo.

5.1 Fechas tempranas

Se comienza de alguna tarea inicial, sin dependencias, para la cual la fecha más temprana será igual a 0 ya que no posee nodo previo, o bien, su nodo previo es el nodo inicial de valor y duración 0. Luego se va avanzando sobre la red conforme se vayan conociendo sus fechas tempranas.

Para nodos con más de una dependencia:

1. Calcular TeE para cada uno de los nodos previos.
2. Elegir el máximo obtenido.

- **TeE:** Fecha más temprana del Nodo Evaluado.
- **TeP:** Fecha más temprana del Nodo Previo.
- **tPE:** Duración de la tarea desde el Nodo Previo al Nodo Evaluado.

Fórmula:

TeE = TeP + tPE

INI. TeIni = 0

A. TeA = TeIni + tIniA
TeA = 0 + 0 = 0

B. TeB = TeIni + tIniB
TeB = 0 + 0 = 0

C. TeC = TeA + tAC
TeC = 0 + 2 = 2

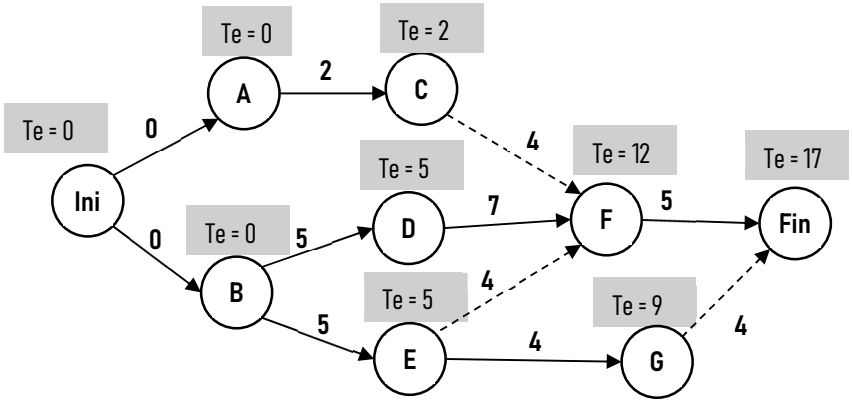
D. TeD = TeB + tBD
TeD = 0 + 5 = 5

E. TeE = TeB + tBE
TeE = 0 + 5 = 5

F. TeF — [tCF o tDF o tEF] → Más de una dependencia.
C. TeC + tCF = 2 + 4 = 6
D. TeD + tDF = 5 + 7 = 12 → TeF = 12 (máximo)
E. TeE + tEF = 5 + 4 = 9

G. TeG = TeE + tEG
TeG = 5 + 4 = 9

FIN. TeFin — [tFFin o tGFin] → Más de una dependencia.
F. TeF + tFFin = 12 + 5 = 17 → TeFin = 17 (máximo)
G. TeG + tGFin = 9 + 4 = 13



5.2 Fechas tardías

Para calcular una fecha tardía se deben conocer los valores de las fechas tempranas. Se parte por el nodo final y se retrocede hasta el principio. La fecha tardía del nodo fin es igual a la de su fecha temprana **(TaFin = TeFin)**

La fórmula se aplica a partir de algún nodo anterior al nodo final. Luego se va a retrocediendo sobre la red conforme se vayan conociendo sus fechas tardías.

Para nodos que poseen más de un nodo siguiente:

1. Calcular TaE para cada uno de los nodos siguientes.
2. Elegir el mínimo obtenido.

- **TaE:** Fecha más tardía del Nodo Evaluado.
- **TaS:** Fecha más tardía del Nodo Siguiente.
- **tES:** Duración de la tarea desde el Nodo Evaluado al Nodo Siguiente.

Fórmula:

TaE = TaS - tES

FIN. TeFin = TaFin = 17

G. TaG = TaFin – tGFin
TaG = 17 – 4 = 13

F. TaF = TaFin – tFFin
TaF = 17 – 5 = 12

E. TaE — [tEF o tEG] → Más de una dependencia.
F. TaF – tEF = 12 – 4 = 8 → TaE = 8 (mínimo)
G. TaG – tEG = 13 – 4 = 9

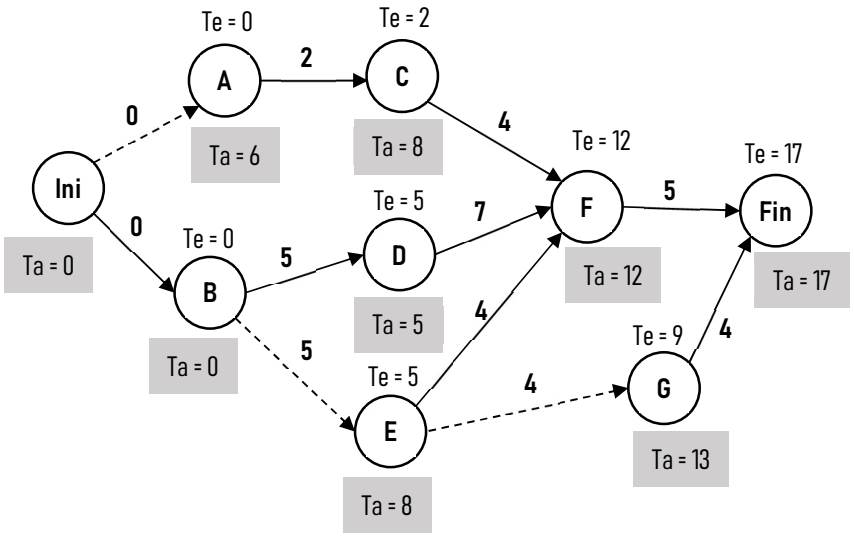
D. TaD = TaF – tDF
TaD = 12 – 7 = 5

C. TaC = TaF – tCF
TaC = 12 – 4 = 8

B. TaB — [tBD o tBE] → Más de una dependencia.
D. TaD – tBD = 5 – 5 = 0 → TaB = 0 (mínimo)
E. TaE – tBE = 8 – 5 = 3

A. TaA = TaC – tAC
TaA = 8 – 2 = 6

INI. TaIni — [tIniA o tIniB] → Más de una dependencia.
- TaA – tIniA = 6 – 0 = 6
- TaB – tIniB = 0 – 0 = 0 → TaIni = 0 (mínimo)



6. Calcular el Margen Total

El cálculo del margen total determina la cantidad de tiempo que una actividad puede o no retrasarse sin afectar la duración total del proyecto. Este cálculo puede hacerse con dos fórmulas distintas, una simple y otra más compleja.

- **TeE:** Fecha más temprana del Nodo Evaluado.
- **TaE:** Fecha más tardía del Nodo Evaluado.

Fórmula 1:

MtE = TaE - TeE

- **TaJ:** Fecha más tardía del Nodo Destino.
- **TeI:** Fecha más temprana del Nodo Origen.
- **tIJ:** Duración de la tarea desde el Nodo Origen al Nodo Destino.

Fórmula 2:

Mt = TaJ – TeI - tIJ

INI. MtIni = 0

A. MtA = TaA - TeA
   MtA = 6 – 0 = 6

B. MtB = TaB - TeB
   MtB = 0 – 0 = 0

C. MtC = TaC - TeC
   MtC = 8 – 2 = 6

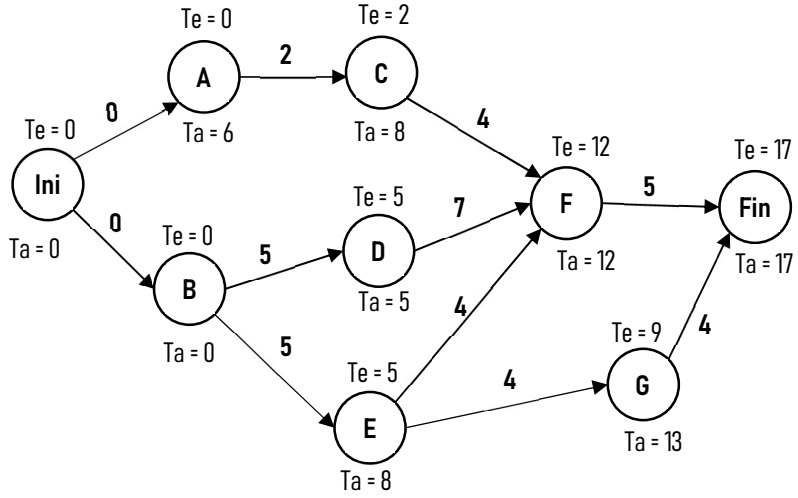
D. MtD = TaD - TeD
   MtD = 5 – 5 = 0

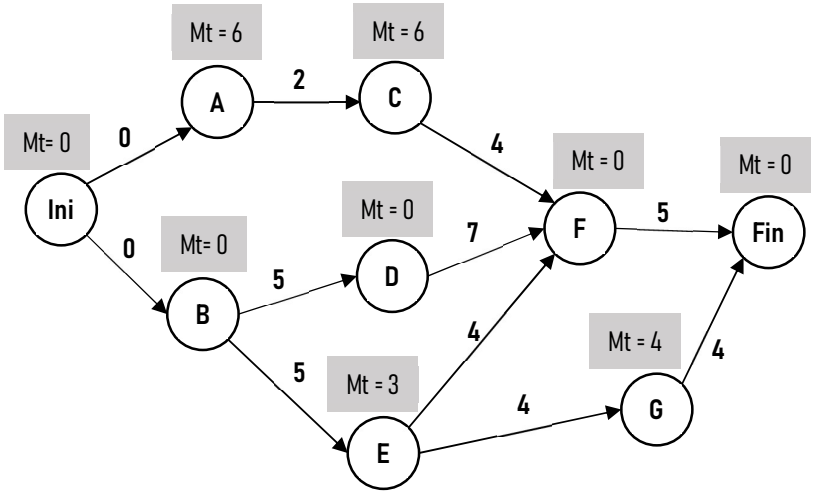
E. MtE = TaE - TeE
   MtE = 8 – 5 = 3

F. MtF = TaF - TeF
   MtF = 12 – 12 = 0

G. MtG = TaG - TeG
   MtG = 13 – 9 = 4

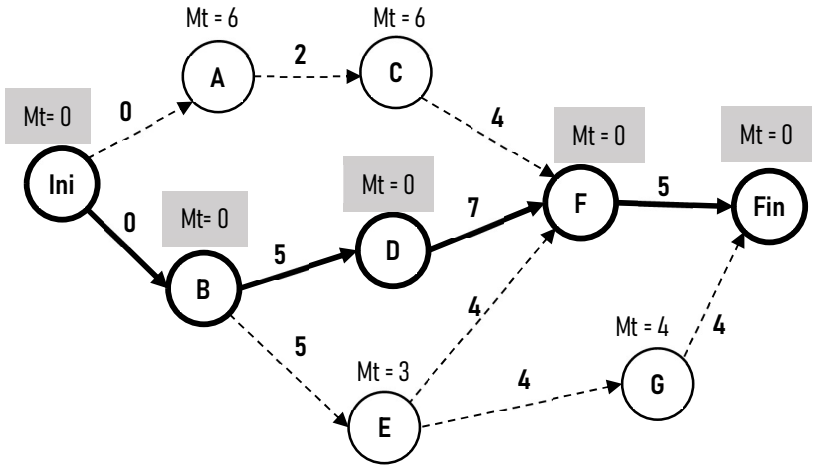
FIN. MtFin = TaFin - TeFin
   MtFin = 17 – 17 = 0





7. Identificar el camino crítico

Este camino consiste en una serie de tareas críticas conectadas que determinan la duración total mínima del proyecto al sumar las duraciones de todas ellas. Está conformado por nodos de margen cero conectados de principio a fin. Cualquier retraso en alguna de tareas retrasará todo el proyecto. Se debe trazar una línea que conduzca estos nodos de principio a fin y finalmente listar esas tareas para otorgarles mayor prioridad.



Datos que se obtienen del PERT - CPM:

- Camino crítico.
- Ventana temporal de cada tarea.
  - Inicio más temprano y más tardío.
  - Final más temprano y más tardío.
  - Margen total.

Tarea crítica:

TeE = TaE
MtE = 0

PLANIFICACIÓN ORGANIZATIVA

La planificación organizativa se refiere a la gestión del recurso humano en un proyecto. El personal es fundamental y su manejo adecuado es un factor crítico para el éxito del proyecto. El equipo de una organización de software es un activo valioso y representa el capital intelectual. Una mala gestión del personal puede uno de los factores principales para el fracaso de los proyectos.

PARTICIPANTES

Una planificación organizativa eficaz requiere la participación de varios roles y partes interesadas en una organización. Estos son algunos de los participantes comunes en un proceso de planificación organizativa:

- **Gerentes ejecutivos:** Definen los temas empresariales.
- **Gerentes de proyecto:** Planifican, motivan, organizan y controlan a los profesionales.
- **Profesionales:** Aportan habilidades técnicas.
- **Clientes:** Especifican los requerimientos.
- **Usuarios finales:** Interactúan con el software.

LÍDERES

El líder de equipo es una persona que desempeña un papel fundamental en la gestión y dirección de un grupo de individuos que trabajan juntos hacia un objetivo común. Tiene la responsabilidad de guiar, coordinar, motivar y supervisar a los miembros del equipo para asegurarse de que trabajen de manera efectiva y alcancen los resultados deseados. La organización del equipo de software es crucial para maximizar las habilidades y capacidades de cada miembro.

Modelo MOI de Liderazgo:

- **Motivación al personal:** Habilidad para incentivar al personal técnico a dar su máximo rendimiento.
- **Organización del equipo:** Habilidad para diseñar procesos que conduzcan al producto final.
- **Innovación:** Capacidad para fomentar la creatividad y la generación de ideas del personal.

Rasgo de un líder eficaz

- **Resolución de problemas:**
  Debe ser capaz de diagnosticar los conflictos técnicos y organizativos más relevantes, estructurar sistemáticamente soluciones o motivar adecuadamente a otros profesionales para desarrollarlas. Debe aplicar lecciones aprendidas de proyectos pasados a situaciones nuevas y ser lo suficientemente flexible para cambiar de dirección si los intentos de resolver el problema resultan infructuosos.
- **Identidad administrativa:**
  Debe tener la confianza para tomar el control cuando sea necesario y permitir que el personal técnico siga sus instintos.
- **Motivación:**
  Debe recompensar la iniciativa y el logro para optimizar la productividad de su equipo de proyecto. Además, debe demostrar a través de sus acciones que no habrá castigos por asumir riesgos de manera controlada. Así el líder motiva al equipo a esforzarse y a tomar iniciativas que beneficien al proyecto en su conjunto.
- **Influencia y construcción de equipo:**
  Debe tener la capacidad de leer señales verbales y no verbales, reaccionar a las necesidades de los demás y mantener la calma bajo estrés.

ESTRUCTURA DE UN EQUIPO DE SOFTWARE

No existe una única estructura de equipo que sea adecuada para todas las situaciones, ya que depende cada organización, sus objetivos y su industria. Algunos factores dan pautas generales que una organización debe considerar, pero la importancia de cada factor y cómo se apliquen puede variar.

1. Dificultad de problema a resolver.
2. Tamaño de programa resultante.
3. Tiempo que el equipo permanecerá unido.
4. Grado en que el equipo puede dividirse en módulos el problema a resolver.
5. Calidad y confiabilidad requerida por el sistema a construir.
6. Rigidez de la fecha de entrega.
7. Grado de sociabilidad requerido en para el proyecto.

COMUNICACIÓN GRUPAL

La comunicación en el equipo de software es un punto crítico y de gran importancia. La efectividad de las comunicaciones en un grupo puede verse influenciada por varios factores, como el estatus de los miembros del grupo, el tamaño del grupo, la composición de hombres y mujeres, las personalidades individuales y los canales de comunicación disponibles.

Para mejorar la productividad de los programadores, es esencial que cuenten con un entorno de trabajo que proporcione los recursos necesarios y áreas de comunicación adecuadas. Esto asegurará una interacción fluida y eficaz entre los miembros del equipo, lo que a su vez contribuirá al éxito del proyecto.

ORGANIGRAMAS DE EQUIPOS GENÉRICOS

Un organigrama de equipo genérico es una representación visual simplificada de la estructura de un equipo o grupo de trabajo que se utiliza como un modelo estándar o típico, en lugar de reflejar una organización o equipo específico. Cada tipo de organigrama tiene sus ventajas y desventajas, y la elección dependerá de la complejidad de la organización, los objetivos de comunicación y cómo se desea visualizar la estructura y las relaciones.

DESCENTRALIZADO DEMOCRÁTICO (DD)	DESCENTRALIZADO CONTROLADO (DC)	CENTRALIZADO CONTROLADO (CC)
No tiene un jefe permanente, se nombran coordinadores de tareas a corto plazo y se sustituyen por otro cuando cambia la tarea.	Tiene un jefe definido que coordina tareas específicas y jefes secundarios para subtareas.	El jefe del equipo se encarga de la resolución de problemas a alto nivel y la coordinación interna del equipo.
La resolución de problemas es una actividad compartida.	La resolución de problemas es una actividad del grupo, pero la implementación de soluciones se reparte entre subgrupos por el jefe del equipo.	Las decisiones y enfoques adoptados son tomados por el jefe.
Las decisiones se toman por consenso.	Las decisiones y enfoques adoptados se toman por conceso.	La resolución de problemas es dirigida por el jefe.
La comunicación entre los miembros del equipo es horizontal.	La comunicación entre los subgrupos e individuos es horizontal y vertical.	La comunicación entre el jefe y los miembros del equipo es vertical.
Adecuada para tareas rápidas y problemas sencillos.	Adecuada para problemas complejos.	Eficaz para abordar problemas difíciles.

Los proyectos muy grandes se gestionan mejor mediante equipos con estructura CC o DC, ya que permiten formar subgrupos fácilmente.

La duración del tiempo de trabajo influye en la moral del equipo. Los equipos de tipo DD son más adecuados para equipos que permanecerán juntos durante períodos prolongados.

RIESGOS

Un riesgo se define como un evento no deseado que conlleva consecuencias negativas. Los gerentes de proyectos tienen la responsabilidad de anticipar la posibilidad de que ocurran eventos no deseados durante el desarrollo o mantenimiento de un proyecto. Deben elaborar planes para prevenir la aparición de estos eventos, o en caso de que sean inevitables, minimizar sus efectos negativos en la medida de lo posible.

- **Anticipar**
- **Prevenir**
- **Evitar**
- **Minimizar**

COMPONENTES

- **Incertidumbre (Probabilidad)**
  Representa la probabilidad de que ocurra un evento riesgoso. Nunca llega al 100%, ya que ese sería un problema presente.

- **Pérdida (Impacto)**
  Indica el impacto y grado de consecuencias negativas que el evento riesgoso podría causar. Puede variar en intensidad, desde leve hasta grave.

ASPECTOS RELACIONADOS

Preocupación por el futuro

Identifica riesgos que podrían llevar al fracaso del proyecto.

Impacto de los cambios en el desarrollo

Cómo los cambios en los requisitos del cliente o en las tecnologías de desarrollo podrían afectar el éxito general y los plazos del proyecto.

Consideraciones en la toma de decisiones

Cómo las decisiones relacionadas con métodos, herramientas, recursos humanos y enfoque en la calidad pueden influir en el desarrollo del proyecto.

FACTORES DE RIESGOS

El grado de riesgo está directamente relacionado con la cantidad de factores de la lista que estén presentes. Cada uno de estos factores aumenta la posibilidad de que los riesgos se materialicen y afecten el desarrollo del proyecto.

- Gerentes de software y clientes no se reunieron formalmente para respaldar el proyecto.
- Los usuarios finales no se comprometen con el proyecto y sistema/producto que se va a construir
- El equipo y sus clientes no entienden por completo los requisitos
- Los clientes no se involucraron plenamente en la definición de los requisitos
- Los usuarios finales no tienen expectativas realistas
- El ámbito del proyecto no es estable
- El equipo no tiene combinación adecuada de habilidades
- Los requisitos del proyecto no son estables
- El equipo no tiene experiencia con la tecnología que se va a implementar
- El número de personas que hay en el equipo no es adecuado para hacer el trabajo
- Falta de consenso entre los clientes/usuarios respecto a la importancia del proyecto y los requisitos para el sistema/producto en desarrollo.

TIPOS DE RIESGOS

	GENÉRICOS	ESPECÍFICOS
Características	Estos riesgos son independientes del dominio en el que se trabaje y pueden aplicarse a cualquier proyecto.	Estos riesgos están relacionados con el dominio específico de desarrollo de software.
Conocidos	Son riesgos que se han experimentado en otros proyectos similares.	Son riesgos que los expertos en el dominio saben que podrían ocurrir.
Predecibles	Son riesgos que son fáciles de anticipar, como cambios en los requisitos.	Son riesgos que se pueden imaginar fácilmente en el contexto del dominio.
Impredecibles	Son riesgos difíciles de prever y pueden surgir en cualquier proyecto.	Son riesgos que podrían surgir, pero son muy difíciles de anticipar.

ADMINISTRACIÓN DE RIESGOS

La administración de riesgos es un proceso integral que involucra la identificación, evaluación, planificación y control de los riesgos que pueden afectar un proyecto, proceso o actividad. El objetivo principal de la administración de riesgos es minimizar la probabilidad de eventos adversos y reducir su impacto en caso de que ocurran. Esto implica tomar medidas proactivas para evitar, mitigar o gestionar los riesgos de manera efectiva, garantizando así la continuidad y el éxito de la empresa, proyecto o actividad.

ADMINISTRACIÓN DE RIESGOS | ESTRATEGIAS DE GESTIÓN DE RIESGOS

La gestión de riesgos implica estrategias para abordar las amenazas de manera efectiva. Hay dos enfoques principales, estrategias proactivas y reactivas. Una gestión efectiva de riesgos combina elementos proactivos y reactivos y se adapta a las necesidades de la organización y los riesgos involucrados. Es esencial para la toma de decisiones empresariales inteligentes y la protección de los intereses de la organización.

Reactivas

Enfoque en reaccionar ante un problema una vez que ocurre, manejando la crisis en el momento. No hay prevención, solo se toma acción cuando el problema se presenta. Por ejemplo, no tomar medidas anticipadas para evitar una posible falla en un dispositivo y solucionar el problema solo cuando se rompe.

Proactivas

Enfoque en anticipar y prevenir problemas. Se reconoce que los riesgos pueden surgir y se planifican estrategias de tratamiento para evitar que los riesgos tengan un impacto crítico. Por ejemplo, reservar recursos para posibles fallas de dispositivos, lo que ahorra tiempo y contempla los gastos potenciales.

ADMINISTRACIÓN DE RIESGOS | PROCESO DE GESTIÓN DE RIESGOS

El proceso de gestión de riesgos consta de 4 pasos: identificación, análisis, planeación y supervisión. Este proceso es iterativo, ya que después de cada supervisión, es posible que sea necesario reevaluar los riesgos. A medida que el desarrollo de software avanza, este proceso debe repetirse de manera constante y documentarse.

1. Identificación de Riesgos - Listado de Riesgos Potenciales

En la fase inicial de la gestión de riesgos, se busca identificar los posibles riesgos que podrían afectar al proyecto, producto o negocio. Esto se logra a través, generalmente, de un proceso de brainstorming en el cual el equipo de trabajo analiza y genera una lista de los riesgos que podrían surgir. Sin embargo, se enfoca en los "verdaderos riesgos", aquellos que tienen una alta probabilidad de ocurrencia y un impacto significativo.

Para facilitar esta identificación, se puede utilizar una "lista de comprobación de elementos de riesgo" que ofrece pautas sobre dónde podrían surgir los riesgos. Estos pueden abarcar aspectos tecnológicos, personales, organizacionales, de herramientas, requerimientos o estimación.

Cada riesgo identificado se clasifica en una categoría correspondiente, ya sea proyecto, producto o negocio. Además, se proporciona una descripción detallada de cada riesgo, incluyendo información sobre cómo podría afectar, cuáles podrían ser sus causas y cómo podría evolucionar a lo largo del tiempo.

2. Análisis de Riesgos - Listado de Priorización de Riesgos

En esta etapa, cada riesgo del listado se evalúa en términos de su impacto y probabilidad de ocurrencia (incertidumbre + pérdida). Luego, los riesgos se ordenan por prioridad en una tabla y se establece una línea de corte que divide los factores de riesgo.

1º. Columna – Riesgo
Todos los riesgos en desorden.

2º. Columna - Categoría

- **Proyecto:**
  Relacionados directamente con el desarrollo del proyecto, como estimaciones de tiempo incorrectas, cambios en los requerimientos por parte de los usuarios, falta de personal, efectos negativos del nuevo, etc. Afectan la planificación.
- **Producto:**
  Relacionados con el producto final, como funcionalidades faltantes, cambios en la tecnología, falta de dominio del lenguaje utilizado, etc. Afecta la calidad.
- **Negocio:**
  Relacionados con el entorno empresarial, como cambios en la dirección, modificaciones legales, regulaciones nuevas, etc.

3º. Columna - Probabilidad
Escala de probabilidad observada:

- **Bastante improbable**: < 10%
- **Improbable:** 10-25%
- **Moderado:** 25-50%
- **Probable** :50-75%
- **Bastante probable**: >75%

4°. Columna – Impacto

1. **Catastrófico:** Cancelación del proyecto.
2. **Serio:** Reducción significativa en el rendimiento, retrasos en la entrega, excesos importantes en costos.
3. **Tolerable:** Reducciones mínimas en el rendimiento, posibles retrasos, excesos en costos.
4. **Insignificante:** Mínimo impacto en el desarrollo.

2.1. Línea de Corte Y Boehm:

La lista de riesgos se ordena considerando su impacto y probabilidad, y se coloca una línea de corte para establecer cuáles riesgos serán atendidos. Mientras Boehm sugiere supervisar los 10 riesgos más altos, ese número debe ser manejable y adaptarse al proyecto.

Los riesgos por encima de la línea reciben atención primaria, mientras que los debajo se reevaluarán en supervisiones y tendrán una prioridad secundaria.

Es importante equilibrar impacto y probabilidad. Riesgos de baja probabilidad, así tengan un alto impacto, no deben consumir excesivo tiempo, por lo que se establece la línea de corte sobre estos. Riesgos con probabilidad moderada o alta, con poco o mucho impacto merecen mayor atención y se ubicarán sobre la línea de corte.

Gran impacto - Probabilidad moderada/alta
Poco impacto - Probabilidad muy alta

Línea de corte

Bajo o gran impacto - Probabilidad baja

Riesgo	Categoría	Probabilidad	Impacto
AAA	Proyecto	80%	2
FDS	Producto	80%	3
DFA	Negocio	75%	3
GFE	Proyecto	60%	2
GTR	Proyecto	50%	2
UID	Negocio	40%	1
HWR	Producto	20%	2

3. Planeación - Anulación de Riesgos y Planes de Contingencia

En esta etapa, se abordan los riesgos que están por encima de la línea de corte (los más prioritarios) y se determina la estrategia a seguir:

- **Evitar (anular) el riesgo**
  Esta estrategia implica diseñar el sistema de manera que el evento de riesgo no pueda ocurrir.
- **Minimizar el riesgo**
  Esta estrategia busca reducir la probabilidad de que el riesgo ocurra.
- **Plan de contingencia**
  En esta estrategia, se asume que el riesgo puede ocurrir y se establece un plan para mitigar las consecuencias. Los riesgos tratados con esta estrategia no pueden ser anulados ni minimizados.

3.1 Costos

Al tomar decisiones sobre el tratamiento de riesgos, es importante considerar el costo de aplicar las estrategias. La gestión de riesgos puede ser un proceso costoso.
En empresas pequeñas suele pasar que se enfrenten los riesgos de manera reactiva por tener presupuestos limitados.

- **EXPOSICIÓN:** Probabilidad que ocurra x costo del proyecto si sucede el riesgo.

Fórmula:

$$\text{INFLUENCIA} = \frac{(\text{EXPOSICION antes} - \text{EXPOSICION después})}{\text{COSTO de reducción}}$$

Para que se justifiquen las acciones de reducción del riesgo el valor de INFLUENCIA debe ser alto.

Riesgo	Estrategia
FGFH	
SDGS	
RGFS	
GFSE	
HTYU	

4. Supervisión

La supervisión de riesgos es esencial en todas las etapas del proyecto. En cada revisión administrativa, es necesario analizar y evaluar individualmente los riesgos clave. Se debe determinar si la probabilidad de ocurrencia de cada riesgo ha cambiado y si las consecuencias del riesgo han variado en gravedad.

Valoración de riesgos:

- Evaluar si ha cambiado la probabilidad de cada riesgo.
- Evaluar la efectividad de las estrategias de mitigación propuestas.
- Detectar la ocurrencia de riesgos previamente identificados.
- Verificar el cumplimiento de los pasos definidos para tratar cada riesgo.
- Recopilar información para el futuro.
- Determinar si existen nuevos riesgos.
- Reevaluar periódicamente los riesgos.

La supervisión constante asegura que el proyecto se adapte a las condiciones cambiantes y permite tomar acciones preventivas o correctivas según sea necesario para mantener bajo control los riesgos identificados. Tras esta etapa, algunos riesgos vuelven a la etapa de análisis ya que es un proceso iterativo con constantes cambios a partir de la evolución del proyecto.

DISEÑO DE SOFTWARE

El diseño de software es una etapa fundamental en el desarrollo de sistemas y aplicaciones de software. Es el proceso mediante el cual se crea una representación detallada y estructurada de cómo se construirá el software final, teniendo en cuenta los requisitos del sistema, las necesidades de los usuarios y las consideraciones técnicas. En otras palabras, el diseño de software se encarga de definir cómo se organizarán los componentes, módulos y funciones del software, así como las interacciones entre ellos, para lograr los objetivos planteados.

IMPORTANCIA

El diseño de software es esencial ya que representa la primera actividad técnica en el proceso de desarrollo. Se considera el núcleo técnico de la ingeniería de software, donde los problemas se convierten en soluciones.

En esta etapa, se puede establecer la calidad, ya que se generan múltiples representaciones del desarrollo que se evalúan en términos de esta. Un diseño deficiente o la ausencia de diseño conlleva el riesgo de crear un sistema inestable que podría fallar incluso con cambios mínimos en su implementación.

ÁREAS

Diseño de datos

Transforma el modelo del dominio, derivado del análisis, en estructuras concretas de datos, objetos de datos y relaciones. Este diseño define cómo se almacenarán y gestionarán los datos en el sistema.

Diseño arquitectónico

Establece la relación entre los componentes estructurales del software y selecciona los estilos arquitectónicos, patrones de diseño y otras decisiones clave para la estructura general del sistema.

Diseño a nivel de componentes

Convierte los elementos arquitectónicos en una descripción detallada de los componentes de software y sus funcionalidades. Se basa en modelos de clases, flujos y comportamientos para definir cómo funcionarán estos componentes.

Diseño de interfaz

Define cómo se comunicará el software tanto internamente como con sistemas externos y personas. Describe la forma en que los usuarios interactuarán con el sistema y cómo se intercambiará información con otras aplicaciones.

CRITERIOS PARA EVALUAR EL DISEÑO DE SOFTWARE

Cumplimiento de requerimientos

Un diseño correcto debe asegurarse de implementar todos los requisitos explícitos y también capturar los requisitos implícitos que son esenciales para satisfacer las necesidades del cliente. Es fundamental que el diseño aborde todos los aspectos necesarios y que esté en línea con las expectativas y necesidades de los stakeholders.

Legibilidad

El diseño debe ser claro y legible para todos los involucrados en el proceso de desarrollo y mantenimiento del software. Tanto los desarrolladores que crearán el código como el personal de soporte deben poder entender fácilmente las decisiones tomadas en el diseño sin ambigüedades.

Completo

El diseño debe ofrecer una visión completa y detallada del software. Esto significa que debe abarcar todos los componentes, módulos, interacciones y funcionalidades necesarios para cumplir con los objetivos del proyecto. Un diseño completo evita que se pasen por alto aspectos importantes del sistema.

CRITERIOS TÉCNICOS PARA CREAR UN BUEN DISEÑO

- **Estructura arquitectónica con patrones reconocibles:** El diseño debe tener una estructura arquitectónica basada en patrones de diseño reconocidos, con componentes bien diseñados y capaces de evolucionar.
- **Modularidad:** El diseño debe ser modular, dividiendo el sistema en componentes independientes y cohesivos.
- **Múltiples representaciones:** Debe incluir diversas representaciones para abordar diferentes aspectos del sistema.
- **Estructuras de datos adecuadas:** Debe conducir a la adopción de estructuras de datos apropiadas basadas en patrones reconocibles.
- **Componentes con características funcionales independientes:** Los componentes deben ser independientes y cumplir con funciones específicas.
- **Interfaces simplificadas:** Deberá crear interfaces que minimicen la complejidad de las conexiones entre módulos y con el entorno externo.
- **Método repetitivo controlado por análisis de requisitos:** El diseño debe ser resultado de un proceso repetitivo controlado por información obtenida en el análisis de requisitos.
- **Representación efectiva:** Debe ser representado mediante una notación que comunique eficazmente su significado.

PRINCIPIOS DEL MODELADO DE DISEÑO

- **Rastreo a requerimientos**: El diseño debe ser trazable hasta los requisitos del modelo, asegurando que se satisfagan las necesidades del cliente.
- **Considerar la arquitectura:** Siempre se debe tener en cuenta la arquitectura del sistema que se va a construir, garantizando una estructura sólida.
- **Igual importancia de datos y funciones:** El diseño de datos es tan relevante como el diseño de funciones, ya que ambos son componentes clave del sistema.
- **Diseño cuidadoso de interfaces:** Las interfaces entre componentes deben ser diseñadas cuidadosamente para garantizar una comunicación efectiva.
- **Enfoque en la facilidad de uso:** El diseño de interfaz debe adaptarse a las necesidades del usuario y ser fácil de usar.
- **Independencia funcional en diseño de componentes:** El diseño a nivel de componentes debe ser funcionalmente independiente para permitir reutilización.
- **Bajo acoplamiento:** Los componentes deben tener un acoplamiento mínimo para facilitar cambios y actualizaciones.
- **Representaciones comprensibles:** Las representaciones de diseño deben ser fáciles de entender para todos los involucrados.
- **Desarrollo iterativo:** El diseño debe ser desarrollado de manera iterativa, permitiendo ajustes a medida que el proyecto avanza.
- **Compatibilidad con metodologías ágiles:** El modelado de diseño no está en desacuerdo con las metodologías ágiles; puede adaptarse para encajar en estas prácticas.

EVOLUCIÓN DEL DISEÑO DE SOFTWARE

La evolución del diseño de software a lo largo de más de seis décadas ha sido un proceso constante que ha abarcado diversas metodologías y enfoques. Desde los programas modulares y la refinación de estructuras de software hasta los enfoques orientados a objetos, orientados a modelos y basados en pruebas, cada etapa ha traído consigo innovaciones en cómo se aborda el diseño de sistemas y aplicaciones.

A lo largo de estas diferentes fases, se han observado características comunes que se aplican a diversos enfoques de diseño:

- **Traducción del modelo de requerimientos:** En todos los casos, el diseño busca traducir el modelo de requerimientos en una representación concreta que guíe la construcción del software.
- **Notación para componentes e interfaces:** Cada enfoque utiliza una notación específica para representar los componentes funcionales y sus interacciones dentro del sistema.
- **Refinamiento heurístico:** Se emplean heurísticas y técnicas para refinar el diseño a lo largo del proceso, mejorando gradualmente la calidad y la efectividad del sistema.
- **Evaluación de calidad:** Se establecen lineamientos y criterios para evaluar la calidad del diseño, asegurando que cumpla con los requisitos y sea robusto.

CONCEPTOS DE DISEÑO

A pesar de las diferencias entre los tipos de diseño, existen conceptos básicos que son fundamentales independientemente del enfoque que se adopte.

Son importantes para realizar un buen diseño debido a diferentes motivos, entre ellos:

- Guían la elección de criterios para dividir el software en componentes individuales.
- Facilitan la extracción de detalles de funciones y estructuras de datos desde la representación conceptual del software.
- Ayudan a implementar criterios uniformes que definen la calidad técnica del diseño de software.

CONCEPTOS DE DISEÑO | ABSTRACCIÓN

La abstracción permite enfocarnos en problemas a un nivel de generalización sin preocuparnos por detalles de bajo nivel. A medida que resolvemos problemas, disminuye la abstracción. Los requerimientos son de alto nivel, mientras que el código está a un nivel más detallado.

Tipos:

- **Abstracción Procedimental:** Consiste en identificar y agrupar un conjunto de instrucciones relacionadas bajo un nombre específico.
- **Abstracción de Datos:** Implica dar un nombre a un conjunto de datos que están relacionados y tienen un propósito definido.

CONCEPTOS DE DISEÑO | ARQUITECTURA DEL SOFTWARE

La arquitectura del software es la estructura global que proporciona coherencia conceptual al sistema. Muestra cómo las partes se organizan e interactúan para formar el conjunto.

CONCEPTOS DE DISEÑO | PATRONES

Los patrones representan soluciones efectivas para problemas comunes que se presentan repetidamente en un entorno específico. Estas soluciones ya han sido probadas y validadas en situaciones similares. Los patrones describen una estructura de diseño que resuelve un problema particular dentro de un contexto específico.

Al considerar un patrón, es importante evaluar si:

- **Es Aplicable:** Determinar si el patrón es relevante y puede ser adaptado a la situación de trabajo actual.
- **Es Reutilizable:** Evaluar si el patrón puede ser utilizado en múltiples proyectos o situaciones, lo que ahorra tiempo y esfuerzo.
- **Sirve como Guía:** Reconocer si el patrón puede actuar como un modelo o dirección para desarrollar soluciones similares, pero con funcionalidades o estructuras ligeramente diferentes.

CONCEPTOS DE DISEÑO | MODULARIDAD

La modularidad en el diseño de software implica dividir el software en módulos, que son componentes identificados y abordados por separado. Estos módulos se integran para satisfacer los requisitos del problema en conjunto. El objetivo es evitar dos extremos: tener un software monolítico, es decir, un único módulo gigante, o tener una modularización excesiva con una cantidad exagerada de módulos pequeños y fragmentados.

A medida que se aumenta el número de módulos, el costo por cada módulo individual disminuye, ya que son más manejables. Sin embargo, el costo de integrar todos estos módulos crece. Tener solo uno o muy pocos módulos resultaría en un costo total elevado. Por lo tanto, se busca encontrar un equilibrio óptimo en la cantidad de módulos. Esto se basa en la complejidad específica de cada desarrollo. El objetivo es lograr una modularización eficiente que facilite el mantenimiento, la comprensión y la reutilización, mientras se minimizan los costos asociados con la integración y el desarrollo en su conjunto.

CONCEPTOS DE DISEÑO | OCULTAMIENTO DE INFORMACIÓN

Con el fin de lograr una modularización efectiva, es necesario especificar y diseñar módulos de manera que gestionen la información de manera accesible solo para aquellos que la requieran y, al mismo tiempo, sean inaccesibles para aquellos que no necesiten dicha información.

Este enfoque es similar a tener módulos independientes, donde cada módulo tiene su propia esfera de funcionamiento y está aislado del acceso no autorizado a su información interna. De esta manera, se crea una barrera que evita que la complejidad interna de un módulo se propague a otros, lo que facilita la comprensión y el mantenimiento del sistema en su conjunto.

CONCEPTOS DE DISEÑO | INDEPENDENCIA FUNCIONAL

La independencia funcional surge de la combinación de tres principios de diseño (Modularidad, Abstracción y Ocultamiento de la Información). Se logra al tener módulos que operan de manera independiente y que no tienen una dependencia excesiva con otros módulos. Esta independencia funcional se evalúa a través de dos aspectos: la Cohesión y el Acoplamiento entre los módulos. Se busca alcanzar una alta cohesión y un bajo acoplamiento.

CONCEPTOS DE DISEÑO | INDEPENDENCIA FUNCIONAL | COHESIÓN

La cohesión mide la fuerza de la relación funcional entre las sentencias o grupos de sentencias dentro de un mismo módulo. Indica qué tan centrado y especializado está un módulo en términos de su funcionalidad interna (las sentencias que contiene).

Un módulo se considera altamente cohesivo cuando se enfoca en una sola responsabilidad o tarea bien definida, evitando la sobrecarga con múltiples funciones y responsabilidades diversas. Esto facilita la comprensión y el mantenimiento del módulo, y permite su reutilización en futuros programas que requieran de esa misma tarea. Por otro lado, un módulo con baja cohesión realiza múltiples tareas que tienen poca o ninguna relación entre sí, lo que puede resultar en un código confuso, difícil de mantener y propenso a errores.

La meta es lograr una alta cohesión en cada módulo, lo que significa que cada uno debe tener una función clara y específica. Esto se refiere a la manera en que las responsabilidades están distribuidas entre los módulos. En otras palabras, se busca que cada módulo tenga una responsabilidad única y bien definida.

Niveles de Cohesión

- **Coincidental:** Las sentencias o tareas dentro del módulo carecen de una relación evidente o tienen una relación muy limitada.
- **Lógica:** Las sentencias o tareas dentro del módulo están agrupadas por una lógica común. La relación se basa en una estructura lógica compartida.
- **Temporal:** Las sentencias o tareas dentro del módulo están sincronizadas y relacionadas porque se ejecutan dentro de un intervalo de tiempo específico.
- **Procedimental:** Las sentencias o tareas dentro del módulo están relacionadas porque forman parte de un proceso más amplio y necesitan ejecutarse en un orden específico. Su cohesión está vinculada a un flujo de ejecución.
- **Comunicacional:** Las sentencias o tareas dentro del módulo están relacionadas porque comparten ciertos datos o parámetros. El enfoque está en los elementos de procesamiento centrados en la entrada y salida de datos.
- **Funcional:** Las sentencias o tareas de un módulo están relacionadas en el desarrollo de una única función. Su cohesión se basa en la realización de una tarea específica y bien definida.

CONCEPTOS DE DISEÑO | INDEPENDENCIA FUNCIONAL | ACOPLAMIENTO

El acoplamiento se refiere a la medida de interconexión entre los módulos de un sistema. Representa cómo interactúan y se comunican los módulos, así como el grado en que dependen entre sí. Esta interacción se lleva a cabo a través de entradas, referencias y el intercambio de datos, información y estructuras.

Un bajo acoplamiento se da cuando los módulos están diseñados de manera independiente, lo que les permite ser modificados, reemplazados o reutilizados con un impacto mínimo en otros módulos. Esto resulta en una arquitectura flexible y resistente a cambios. Por otro lado, un alto acoplamiento ocurre cuando los módulos están fuertemente interdependientes. En este caso, los cambios en un módulo pueden requerir ajustes significativos en otros módulos estrechamente conectados. Esto dificulta el mantenimiento, extensión y pruebas del sistema.

El nivel de acoplamiento entre los módulos se relaciona con las dependencias existentes. En la práctica, se busca lograr un acoplamiento bajo entre los módulos, lo que contribuye a una arquitectura más modular y flexible, capaz de adaptarse a las necesidades cambiantes del software.

Niveles de Acoplamiento:

- **Bajo:**
  - **Acoplamiento de datos:** Los módulos interactúan compartiendo datos entre sí.
  - **Acoplamiento de marca:** Los módulos interactúan a través de una estructura compartida.
- **Moderado:**
  - **Acoplamiento de control:** La interacción entre módulos se realiza mediante un indicador de control.
- **Alto:**
  - **Acoplamiento común:** Los módulos interactúan utilizando variables globales compartidas.
  - **Acoplamiento externo:** Los módulos interactúan a través de protocolos de integración o interfaces externas, como bibliotecas o APIs.
  - **Acoplamiento de contenido:** Los módulos interactúan manipulando datos internos de otro módulo.

Se busca una alta cohesión y bajo acoplamiento.

Se puede trabajar con alta cohesión y alto acoplamiento, o baja cohesión y bajo acoplamiento, pero no es lo recomendable.

Una baja cohesión con un alto acoplamiento impactaría directamente en la calidad del software y sería inviable.

CONCEPTOS DE DISEÑO | REFINAMIENTO

El proceso de refinamiento se lleva a cabo de manera gradual y en etapas sucesivas, a medida que se desciende a niveles de detalle más procedimentales. Se inicia con una descripción de alto nivel que aborda una funcionalidad específica, pero sin profundizar en los detalles operativos. Conforme se avanza, se trabaja en cada iteración con un mayor nivel de detalle, lo que permite adentrarse en el funcionamiento concreto de la funcionalidad en cuestión.

Es importante destacar que la abstracción y el refinamiento son conceptos que se complementan mutuamente. Mientras que la abstracción permite definir procedimientos y datos sin adentrarse en los aspectos menos detallados, el refinamiento revela esos detalles a medida que se desarrolla el diseño. En esencia, la abstracción establece una estructura general y el refinamiento aporta las capas de detalle necesarias para concretar esa estructura en un diseño funcional y coherente.

La refabricación, también conocida como rediseño o "refactoring" en inglés, es una técnica de reorganización que tiene como objetivo simplificar el diseño de un componente sin alterar su función ni su comportamiento. Esta técnica se emplea especialmente en metodologías ágiles y a lo largo de las distintas iteraciones del proceso de desarrollo. Su propósito principal es permitir que los módulos que lo requieran puedan alcanzar niveles más altos de cohesión y acoplamiento, a través de modificaciones internas.

Durante el proceso de refabricación, se analizan elementos como redundancias, componentes superfluos, algoritmos innecesarios y estructuras inapropiadas. A medida que se identifican estos aspectos que pueden ser mejorados, se generan nuevas versiones de los módulos o procedimientos involucrados. Esto resulta en una mejora en la cohesión (la relación interna de componentes) y en el acoplamiento (la interacción entre componentes) del diseño, lo que a su vez contribuye a un software más eficiente y mantenible.

El diseño arquitectónico define las relaciones entre los elementos estructurales para cumplir con los requisitos del sistema. Una vez que los elementos del sistema han sido identificados, el sistema arquitectónico se encarga de establecer las conexiones entre ellos para satisfacer los requisitos. Los sistemas complejos se dividen en subsistemas que ofrecen conjuntos de servicios interrelacionados. El proceso de diseño arquitectónico consiste en identificar estos subsistemas dentro del sistema y establecer el marco de control y comunicación entre ellos.

REQUISITOS NO FUNCIONALES

La arquitectura tiene un impacto directo en los requisitos no funcionales, siendo los más críticos el rendimiento, la seguridad, la protección, la disponibilidad y la mantenibilidad.

- **Rendimiento**
  Es necesario agrupar las operaciones críticas en un reducido número de subsistemas (componentes de granularidad gruesa y baja comunicación). Tener muchos subsistemas realizando tareas más pequeñas resulta en una comunicación excesiva, lo cual afecta negativamente el rendimiento.
- **Seguridad**
  Se debe emplear una arquitectura en capas para proteger los recursos críticos en las capas internas y menos expuestas. Los recursos menos críticos pueden ubicarse en capas más expuestas a la interacción con usuarios y el entorno general.
- **Protección**
  La arquitectura debe ser diseñada de modo que las operaciones relacionadas con la protección se encuentren en un único subsistema (o un pequeño grupo de subsistemas), con el fin de reducir los costos y los problemas de validación de la protección. La centralización de la protección permite tomar medidas en un solo lugar (o en un reducido número de lugares) y evita problemas dispersos en todo el sistema.
- **Disponibilidad**
  La arquitectura debe ser diseñada con componentes redundantes, lo que posibilita el reemplazo sin interrumpir el funcionamiento del sistema. Una arquitectura altamente tolerante a fallos contribuye a la disponibilidad del sistema.
- **Mantenibilidad**
  La arquitectura del sistema debe contemplar componentes autocontenidos de granularidad fina que puedan ser modificados con facilidad. Si bien esto puede entrar en conflicto con el rendimiento que busca componente de granularidad baja, ambos aspectos deben ser considerados en el diseño.

ASPECTOS DEL DISEÑO ARQUITECTÓNICO

1. Organización del sistema
2. Descomposición modular
3. Modelos de control
4. Arquitectura de los Sistemas Distribuidos

1. ORGANIZACIÓN DEL SISTEMA

La organización del sistema representa la estrategia fundamental utilizada para estructurar el sistema. Es la manera en que se dividen y conectan los subsistemas para lograr los objetivos y requisitos del proyecto. Una estructura organizada eficazmente facilita la comunicación, el control y la interacción entre los diferentes elementos del sistema. Los subsistemas deben intercambiar información de forma efectiva. Esto se puede lograr de diferentes maneras, entre ellas:

Datos compartidos en una base de datos central

En este enfoque, se establece una base de datos centralizada que actúa como fuente única para toda la información compartida. Los subsistemas interactúan con esta base de datos para acceder y actualizar la información. Esto permite una visión única y coherente de los datos en todo el sistema, pero también introduce una dependencia crítica en la disponibilidad y el rendimiento de la base de datos central.

Cada subsistema mantiene y comparte su propia información

En este enfoque, cada subsistema mantiene su propio conjunto de datos, que puede estar relacionado con su función específica. Los subsistemas se comunican intercambiando datos directamente entre sí, sin depender de una base de datos central. Esto puede mejorar la autonomía de los subsistemas y reducir la dependencia de una única fuente de datos, pero también puede requerir un mayor esfuerzo de coordinación para garantizar la coherencia de los datos compartidos.

PATRONES DE DISEÑO

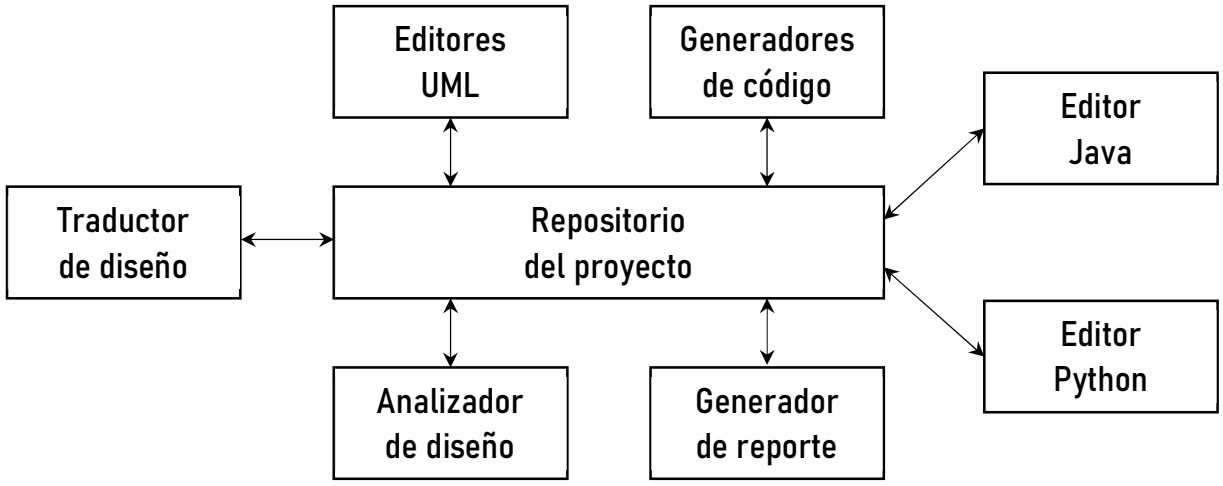
Los patrones arquitectónicos son estilos organizacionales utilizados en diversas arquitecturas de software. Algunos de los más comunes incluyen: repositorio, cliente-servidor, arquitectura en capas o combinaciones entre ellos, entre otros.

PATRÓN DE REPOSITORIO

El patrón de repositorio se basa en la organización de un sistema en torno a una base de datos central compartida, conocida como repositorio. En este enfoque, todos los subsistemas acceden a los datos a través de este repositorio central. Los datos o conjuntos de datos son generados por ciertos subsistemas, almacenados en el repositorio y pueden ser utilizados por otros subsistemas.

Es comúnmente empleado en sistemas que manejan grandes volúmenes de datos que necesitan ser compartidos y accedidos por varios subsistemas. Es especialmente efectivo en entornos donde se requiere una fuente única y centralizada de verdad para los datos.

Ejemplos de aplicación de este patrón incluyen sistemas de gestión, herramientas de diseño asistido por computadora (CAD), aplicaciones de ingeniería de software asistida por computadora (CASE), sistemas de administración de bases de datos, plataformas de recursos humanos, sistemas de inventario y soluciones de seguimiento de ventas, entre otros.



Ventajas

- **Eficiencia:** Compartir grandes cantidades de datos sin necesidad de transmitir información entre subsistemas.
- **Independencia funcional:** Los subsistemas productores de datos no necesitan conocer el uso futuro de los datos.
- **Centralización:** Actividades de backup, protección y control de acceso gestionadas de manera centralizada.
- **Representación visual:** Modelo de datos compartido ofrece una representación estructurada y visual.
- **Integración sencilla:** Facilita la incorporación de nuevas herramientas compatibles con el modelo de datos.

Desventajas

- **Limitación de flexibilidad:** Subsistemas deben ajustarse a los modelos del repositorio, lo que puede afectar su adaptabilidad.
- **Complejidad de evolución:** La acumulación de datos en el repositorio puede dificultar la evolución o migración del sistema, generando problemas de consistencia y errores.
- **Uniformidad de políticas:** Dificultad para acomodar variaciones en protección y políticas de seguridad de diferentes subsistemas.
- **Dificultad en distribución:** Problemas de redundancia, inconsistencias y mantenimiento al distribuir el repositorio en múltiples máquinas.

Aunque el patrón de repositorio puede ser beneficioso en términos de centralización y compartición de datos, también introduce desafíos en cuanto a flexibilidad, adaptabilidad y mantenibilidad a largo plazo.

PATRÓN DE CLIENTE-SERVIDOR

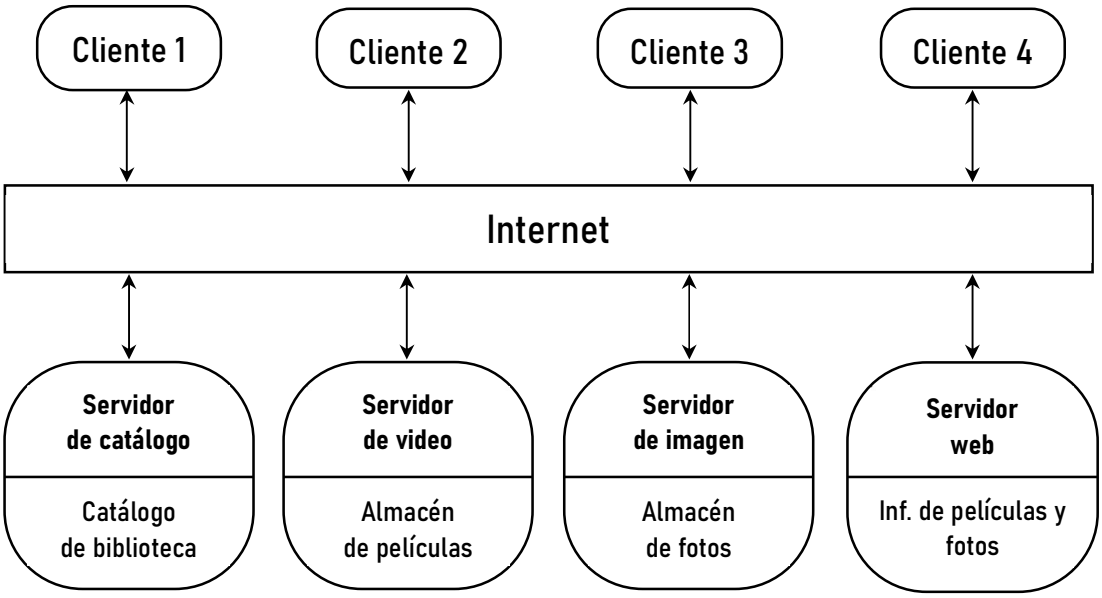
El patrón cliente-servidor es un modelo arquitectónico en el que el sistema se organiza como un conjunto de servicios y servidores asociados, junto con clientes (interfaces) que utilizan estos servicios a través de una red. Esta división de componentes facilita la escalabilidad y permite un desacoplamiento efectivo.

Componentes

- **Servidores de Red:** Proveedores de servicios que ofrecen funcionalidades específicas.
- **Clientes:** Consumidores de servicios que solicitan y utilizan las funciones proporcionadas por los servidores.
- **Red:** Infraestructura que permite a los clientes acceder a los servicios a través de la comunicación en la red.

El patrón cliente-servidor presenta una característica clave en la que los clientes necesitan conocer el nombre del servidor y el nombre del servicio específico que están buscando para acceder a ellos. Por otro lado, los servidores generalmente no requieren conocer la identidad de los clientes para brindar servicios, excepto en situaciones particulares relacionadas con seguridad, auditoría u otros casos específicos.

Este patrón es ampliamente utilizado en sistemas informáticos distribuidos, aplicaciones web y diversas arquitecturas donde se busca una clara división de tareas entre los proveedores de servicios (los servidores) y aquellos que consumen esos servicios (los clientes). Sin embargo, también es posible encontrar situaciones en las que los servicios y los clientes se ejecutan en la misma máquina.



Ventajas:

- **Escalabilidad:** El sistema puede escalarse de manera eficiente mediante la adición de nuevos servidores para manejar la carga de trabajo creciente.
- **Reutilización:** Los servicios proporcionados por los servidores pueden ser reutilizados por múltiples clientes.
- **Interoperabilidad:** Los clientes y servidores pueden estar en diferentes plataformas o lenguajes de programación. Las interfaces adaptan servicios.

Desventajas:

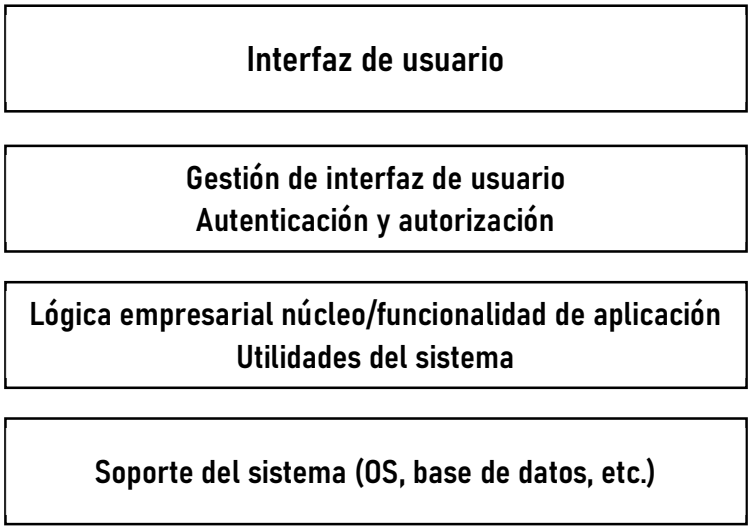
- **Complejidad de Implementación:** Configurar y mantener la infraestructura de servidores y la comunicación puede ser complejo.
- **Dependencia de la Red:** La comunicación constante entre clientes y servidores significa que la calidad y disponibilidad de la red son críticas para el funcionamiento del sistema.
- **Puntos Únicos de Fallo:** Si un servidor falla, los clientes que dependen de ese servidor pueden quedar afectados.

En resumen, el patrón Cliente-Servidor es poderoso para dividir responsabilidades y permitir escalabilidad y reutilización, pero también introduce desafíos relacionados con la complejidad, la dependencia de la red y la seguridad.

PATRÓN DE ARQUITECTURA EN CAPAS

El patrón de arquitectura en capas organiza y diseña un sistema en distintos niveles jerárquicos, cada uno con una función específica. Cada capa ofrece servicios a las capas adyacentes y la comunicación e interacción ocurren a través de sus interfaces. En este patrón generalmente se incluye una capa de soporte interna, una de lógica y funcionalidad, una de gestión de usuarios y una de interfaz de usuario, que es la más externa.

Normalmente se utiliza cuando se busca una separación clara de responsabilidades y una estructura modular, o en sistemas con múltiples niveles de abstracción, donde cada capa realiza una función específica y se comunica con las capas vecinas. Ejemplos de aplicación de este patrón incluyen sistemas de gestión de contenido, plataformas de aprendizaje en línea, aplicaciones empresariales y herramientas de procesamiento de imágenes, entre otros.



Ventajas

- **Desarrollo incremental:** Permite añadir capas gradualmente, facilitando el desarrollo iterativo.
- **Portabilidad y resistencia al cambio:** Mantener las interfaces entre capas permite cambiar o reemplazar capas sin afectar al sistema completo. Incluso en caso de no poder modificar una interface, se puede generar una capa de adaptación para permitir el acoplamiento.
- **Adaptabilidad multiplataforma:** Las capas internas son las únicas dependientes de la plataforma, permitiendo crear capas de interfaz específicas para adaptar otras plataformas.
- **Separación de responsabilidades:** Cada capa tiene una función clara e independiente, facilitando la comprensión y el mantenimiento del sistema.

Desventajas

- **Estructuración compleja:** Planificar qué hace cada capa, dónde ubicarla y cómo será su interfaz puede resultar complejo en la práctica.
- **Impacto en rendimiento y mantenibilidad:** Las capas internas a menudo brindan servicios necesarios para todas las capas. Sus cambios o errores pueden desencadenar fallas a otras unidades.
- **Rutas de acceso complejas:** Los servicios del usuario pueden pasar por múltiples capas adyacentes, pudiendo ocurrir problemas con el flujo y procesamiento de la información.
- **Interpretación repetida de servicios:** En sistemas con muchas capas, un servicio solicitado por la capa superior puede tener múltiples interpretaciones en capas diferentes, lo que reduce la eficiencia y fluidez de respuesta.

En resumen, el patrón de arquitectura en capas brinda una estructura modular y una separación clara de responsabilidades, pero también introduce desafíos en términos de diseño, rendimiento y complejidad de las interacciones entre capas.

2. DESCOMPOSICIÓN MODULAR

La descomposición modular es una estrategia para dividir los subsistemas en módulos más pequeños y manejables dentro de un sistema. A estos módulos se les puede aplicar los patrones y modelos clásicos, pero también permite utilizar otros estilos alternativos.

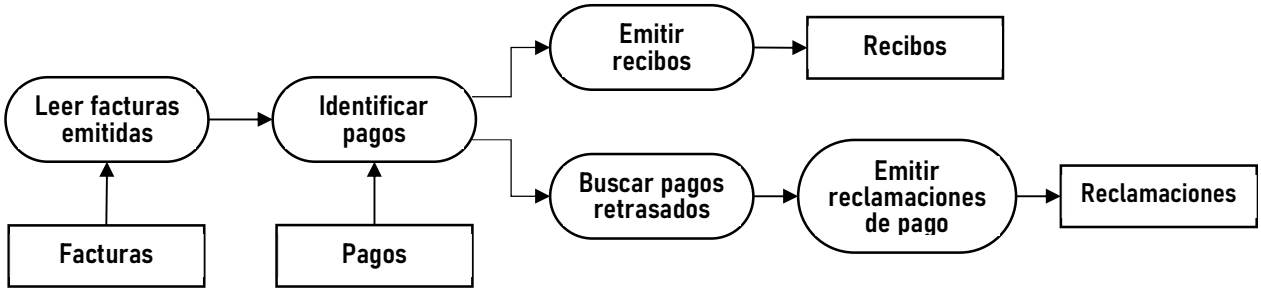
Un sistema es independiente cuando no depende de los servicios de otros para funcionar. Los subsistemas se componen de módulos con interfaces definidas que permiten la comunicación entre ellos. Por otro lado, un módulo es un componente de un subsistema que ofrece y utiliza servicios, pero no se considera un sistema independiente.

La descomposición modular posee dos estrategias: orientada a flujo de funciones y orientada a objetos.

DESCOMPOSICIÓN ORIENTADA A FLUJO DE FUNCIONES

En esta estrategia, los subsistemas se subdividen en un conjunto de módulos funcionales. Cada módulo toma datos de entrada, los procesa y produce una salida. Se centra en el flujo de datos y cómo se transforman a través de los módulos.

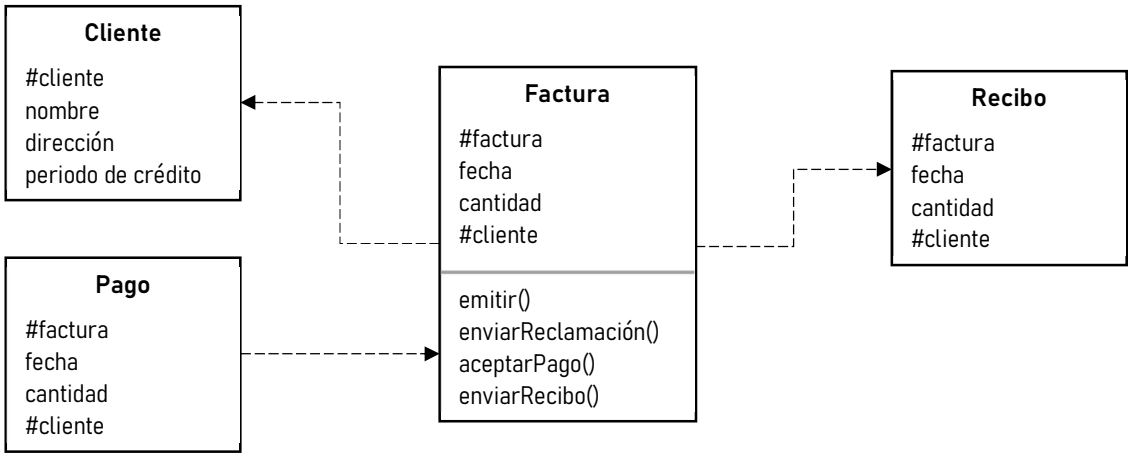
En un modelo orientado a flujo de funciones, los datos fluyen de una función a otra y se transforman a medida que pasan por una secuencia de funciones hasta llegar a los datos de salida. A lo largo de una secuencia de funciones la transformación sería mayor. Las transformaciones se pueden ejecutar en secuencial o en paralelo. Eso finaliza cuando llegar o producir los datos de salida. Este enfoque es especialmente útil para sistemas donde el flujo y la transformación de datos son esenciales.



DESCOMPOSICIÓN ORIENTADA A OBJETOS

En esta estrategia, los subsistemas se descomponen en conjuntos de objetos que interactúan entre sí. Cada objeto tiene su propio conjunto de propiedades y métodos, y la comunicación entre objetos se basa en el envío de mensajes y la invocación o llamadas a métodos. Esta estrategia está más enfocada en el encapsulamiento de la funcionalidad y en cómo los objetos colaboran para lograr las metas del sistema.

Un modelo arquitectónico orientado a objetos estructura al sistema en un conjunto de objetos débilmente acoplados y con interfaces bien definidas. Lo cual resulta muy ventajoso. Esta estrategia es especialmente útil cuando se busca un diseño modular, flexible y reutilizable, ya que los objetos representan unidades autónomas de funcionalidad.



3. MODELOS DE CONTROL

Un modelo de control refiere a un enfoque o estrategia que define cómo se coordina, regula y gestiona la ejecución de los componentes y módulos dentro de un sistema. El control determina cuándo y cómo se ejecutan los módulos, quién toma decisiones y si se permiten ejecuciones simultáneas.

Los modelos de control son esenciales para asegurar que los servicios de los subsistemas se entreguen de manera precisa y en el momento adecuado. A nivel arquitectónico, existen diferentes enfoques para establecer el control dentro del sistema.

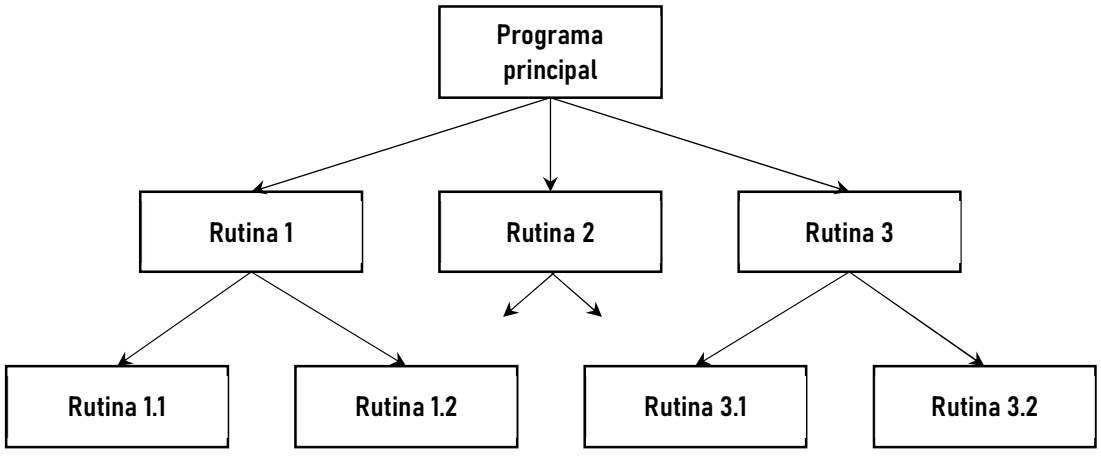
MODELO DE CONTROL CENTRALIZADO

En este modelo, un subsistema específico asume la responsabilidad de iniciar y detener otros subsistemas. Este subsistema controlador gestiona la ejecución de otros subsistemas, ya sea de manera secuencial o en paralelo. Actúa como el núcleo central que invoca a los subsistemas subsiguientes y coordina cuándo los demás subsistemas o módulos deben actuar.

El control centralizado es efectivo para sistemas que requieren una coordinación precisa y un flujo de ejecución bien definido. Sin embargo, puede llevar a cuellos de botella si el subsistema central se convierte en un punto de congestión.

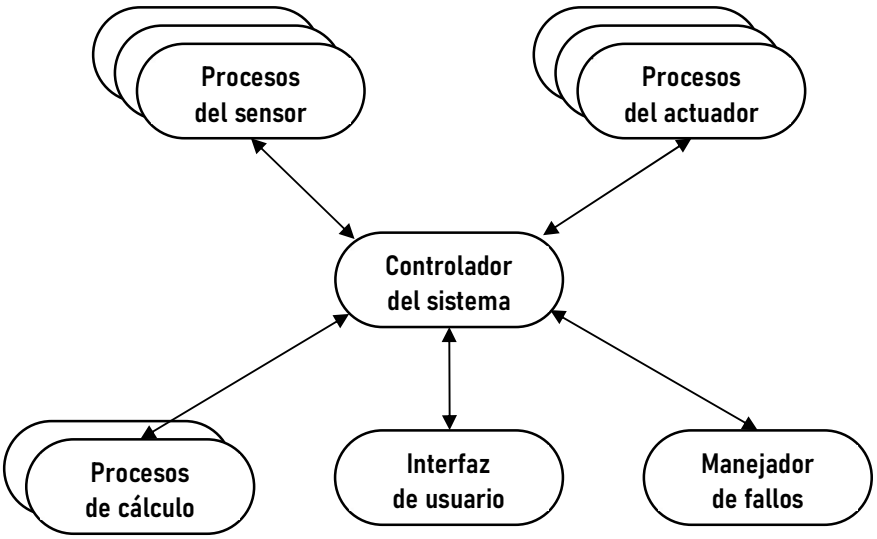
MODELO DE CONTROL CENTRALIZADO | MODELO DE LLAMADA Y RETORNO

El módulo controlador cede el control al módulo invocado y, cuando este último completa su ejecución, devuelve el control al módulo invocador. Este modelo es aplicable a flujos secuenciales de ejecución.



MODELO DE CONTROL CENTRALIZADO | MODELO DE GESTOR

Un gestor controla la ejecución de otros subsistemas y actúa como un coordinador central que inicia y detiene la ejecución de estos subsistemas secundarios según lo requiera el flujo del sistema. El gestor puede planificar y coordinar la ejecución secuencial o simultánea de los subsistemas. Es aplicable a sistemas con ejecuciones concurrentes.



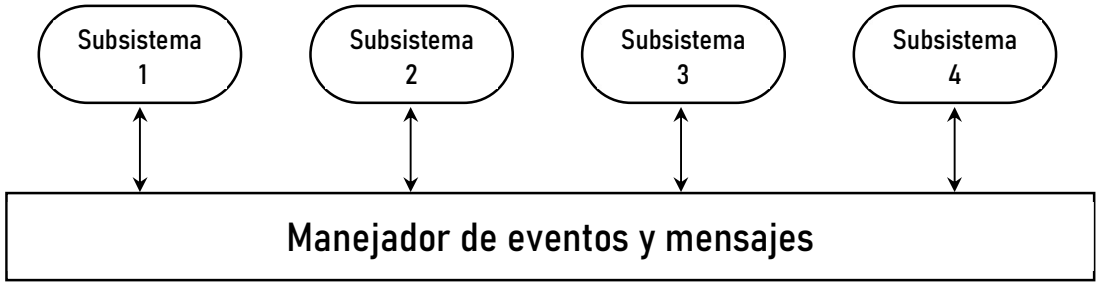
MODELO DE SISTEMAS DIRIGIDOS POR EVENTOS

En este modelo, cada subsistema responde a eventos generados externamente al subsistema. En lugar de depender de un subsistema central para la coordinación, los subsistemas están diseñados para reaccionar a eventos externos específicos. Cuando un evento se desencadena, los subsistemas relevantes responden tomando acciones predefinidas. Estos eventos pueden ser señales binarias, valores dentro de un rango, comandos de entrada o selecciones de menú. Los sistemas dirigidos por eventos se rigen por eventos externos al proceso mismo.

Este modelo es especialmente adecuado para sistemas dinámicos y reactivos, ya que permite una mayor flexibilidad y adaptabilidad a medida que cambian las condiciones.

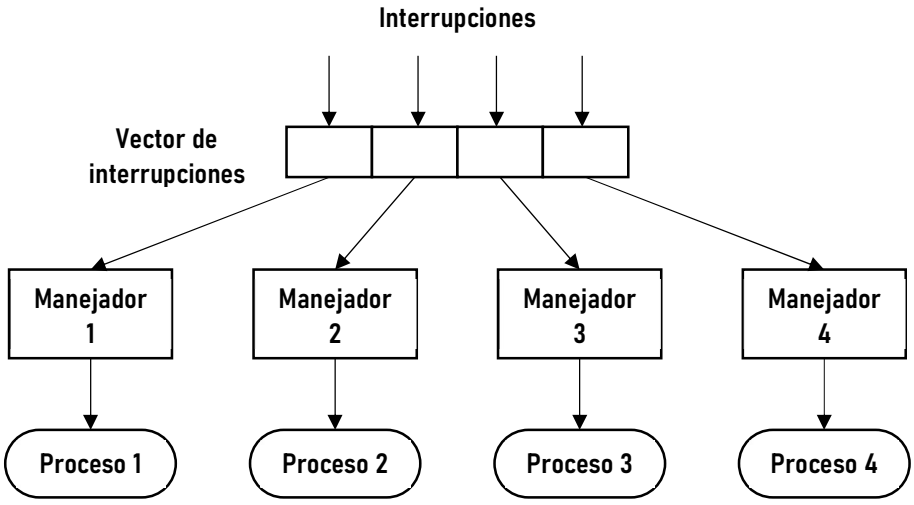
MODELO DE SISTEMAS DIRIGIDOS POR EVENTOS | BROADCAST

En el modelo de transmisión o Broadclast, un evento se emite o transmite a todos los subsistemas en el sistema, y cada subsistema decide si debe responder al evento en función de su programación. Cuando se produce un evento, todos los subsistemas son notificados de él. Los subsistemas que han sido programados para manejar ese tipo de evento específico tomarán medidas en respuesta.



MODELO DE SISTEMAS DIRIGIDOS POR EVENTOS | DIRIGIDO POR INTERRUPCIONES

Un manejador de interrupciones detecta interrupciones externas y direcciona el evento a un componente específico para su procesamiento. Este enfoque es común en sistemas de tiempo real, donde las interrupciones externas son detectadas por el manejador y los componentes adecuados son invocados para el procesamiento.



4. ARQUITECTURA DE LOS SISTEMAS DISTRIBUIDOS

Un sistema distribuido es una infraestructura en la que los componentes de software y hardware se encuentran en diferentes computadoras interconectadas y trabajan juntos para lograr un objetivo común. Estos sistemas se basan en la idea de dividir la carga de trabajo y la responsabilidad en varios nodos interconectados, lo que permite una mayor escalabilidad, redundancia y capacidad de procesamiento. Resumidamente un sistema en el que el procesamiento de información se distribuye sobre varias computadoras. Los tipos más genéricos de esta arquitectura son los de Cliente-Servidor y Componentes Distribuidos

Ventajas

- **Recursos compartidos:** Los sistemas distribuidos permiten compartir recursos como potencia de procesamiento, memoria y almacenamiento entre múltiples nodos, lo que mejora la eficiencia y la utilización de recursos.
- **Apertura:** Generalmente son sistemas abiertos que no dependen de una empresa en particular y se diseñan con protocolos estándar para simplificar la combinación de los recursos y facilitar la integración de diferentes componentes y tecnologías de diversas fuentes.
- **Concurrencia:** Varios procesos pueden ejecutarse simultáneamente en diferentes computadoras, lo que mejora la eficiencia y la capacidad de respuesta del sistema.
- **Escalabilidad:** Los sistemas distribuidos pueden escalarse verticalmente (añadiendo más recursos a una sola máquina) u horizontalmente (añadiendo más máquinas al sistema) para hacer frente a un aumento de la demanda y aumentar la capacidad de procesamiento.
- **Tolerancia a fallos:** Los sistemas distribuidos tienen mayor capacidad para manejar fallos, ya que cuentan con varias computadoras y la posibilidad de replicar información. Si una computadora falla, las demás pueden asumir su carga, aunque con una posible pérdida de rendimiento, permitiendo que el sistema completo continúe funcionando.

Desventajas

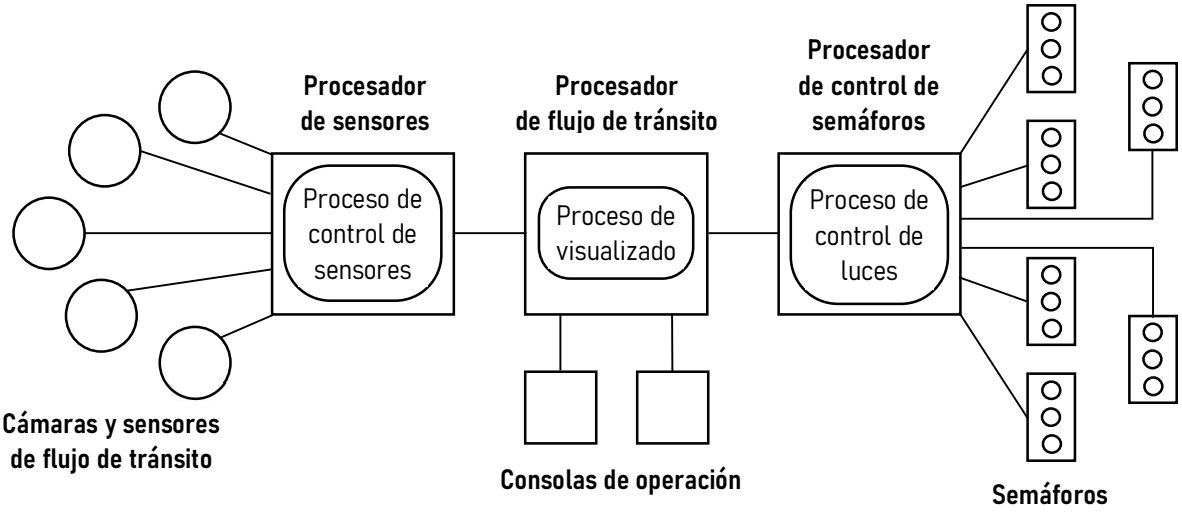
- **Complejidad:** Son más complejos que los centralizados, además del procesamiento hay que tener en cuenta problemas de comunicación, sincronización y coherencia entre los equipos.
- **Seguridad:** Se accede al sistema desde varias computadoras generando tráfico en la red que puede ser intervenido. Al tener múltiples puntos de acceso y comunicación, los sistemas distribuidos pueden ser más vulnerables a ataques y problemas de seguridad.
- **Manejabilidad:** Al ser sistemas abiertos con protocolos estándar las computadoras del sistema pueden ser de diferentes tipos a nivel de hardware y diferentes sistemas operativos, lo que genera más dificultades para gestionar y mantener el sistema.
- **Impredecibilidad:** El tiempo de respuesta de un sistema distribuido varía según la carga del sistema y el estado de la red, lo que dificulta predecir su comportamiento. Esto complica la depuración y la identificación del origen de los problemas, especialmente al determinar de qué computadora provienen, y está relacionado con la complejidad del sistema.

ARQUITECTURA MULTIPROCESADOR

En una arquitectura multiprocesador, el sistema de software se compone de uno o varios procesadores (CPU o núcleos) en un mismo sistema de cómputo. Estos procesadores trabajan de manera simultánea y en paralelo para llevar a cabo diversas tareas y procesos. Los procesos pueden distribuirse entre los distintos núcleos, en algunas situaciones, factores como la carga, los tiempos y la memoria pueden requerir que ciertos procesos se ejecuten en un mismo procesador.

La asignación de procesos a los procesadores puede ser previamente definida basándose en criterios específicos, o gestionada dinámicamente a través de un módulo llamado "dispatcher" o planificador. Este último toma decisiones en tiempo real para optimizar la distribución de la carga de trabajo. El dispatcher considera diversos factores, como las características de los procesos, la disponibilidad de recursos y la carga actual de cada procesador, para tomar decisiones.

Esta arquitectura se utiliza ampliamente en sistemas de tiempo real que requieren recopilación de información, toma de decisiones y emisión de señales para influir en el entorno. Además, se ha vuelto cada vez más común en contextos convencionales, como computadoras de escritorio y dispositivos móviles, como smartphones. La utilización de múltiples procesadores en estos dispositivos permite una ejecución más eficiente de tareas, mejorando la experiencia del usuario en términos de rendimiento y fluidez.



ARQUITECTURA CLIENTE-SERVIDOR

En la arquitectura Cliente-Servidor, se diseña el sistema como un conjunto de servicios proporcionados por servidores y un conjunto de clientes que acceden a estos servicios a través de una red de comunicación.

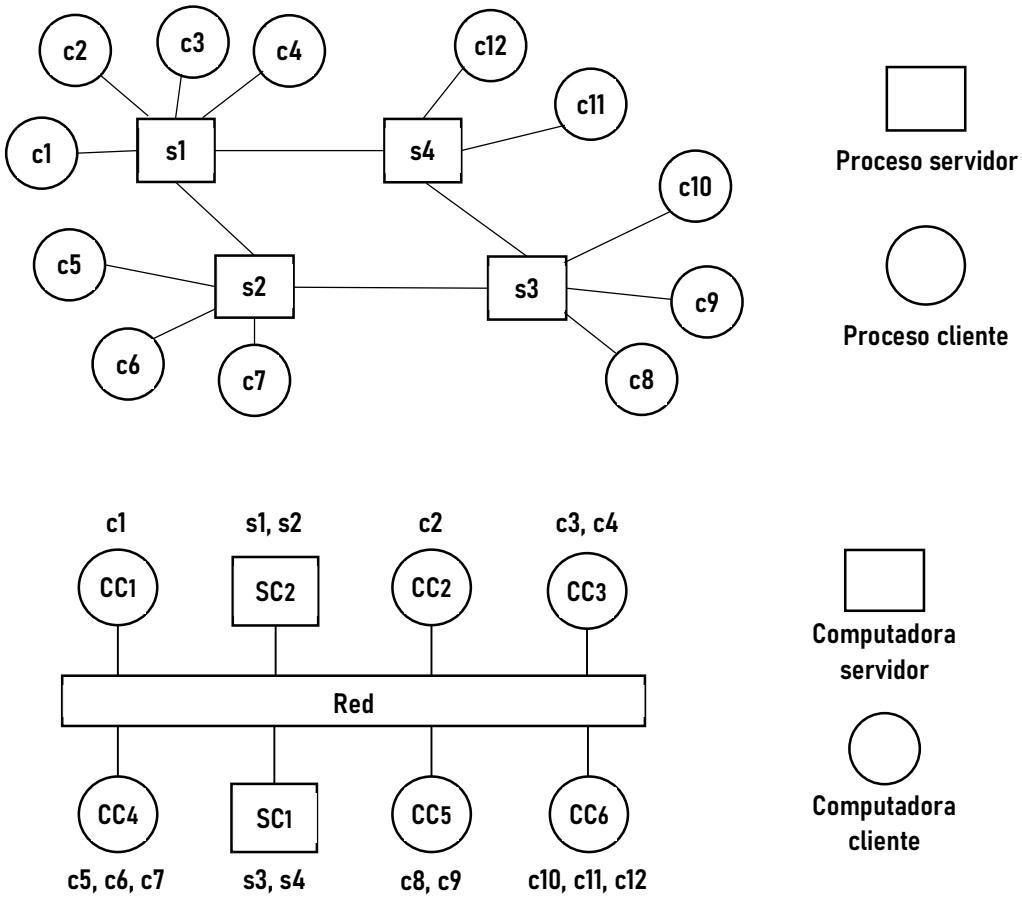
Los clientes y servidores son procesos diferentes desde la óptica de sistemas distribuidos con concurrencia. Operan en equipos y sistemas de cómputo distintos, lo que asegura que los procesos de cliente y servidor se ejecuten de manera independiente. Los servidores pueden atender varios clientes en forma concurrente, lo que mejora la capacidad de respuesta del sistema. Esta capacidad para manejar múltiples solicitudes simultáneamente es esencial en la arquitectura Cliente-Servidor, permitiendo a los clientes obtener servicios de manera ágil incluso en entornos con alta demanda.

Un servidor puede brindar varios servicios, lo que agrega flexibilidad a la arquitectura. Esta versatilidad agrega agilidad al sistema, simplifica la administración y optimiza la utilización de recursos. Los servidores multifuncionales pueden abordar diversas necesidades de los clientes sin requerir la implementación de múltiples servidores individuales.

Los clientes no se conocen entre sí. Cada cliente opera de manera independiente, solicitando servicios según sus necesidades particulares. La interacción entre los clientes y los servidores es directa. En algunas situaciones, es posible que en una misma computadora cliente se ejecuten múltiples instancias de clientes. Esta configuración puede surgir para atender distintos escenarios o necesidades sin requerir el despliegue de varias máquinas cliente.

Además, es posible que coexistan múltiples servidores que ofrezcan el mismo servicio o servicios similares en un mismo sistema de cómputo. Esto aumenta la disponibilidad y la redundancia, lo que resulta útil para garantizar la continuidad del servicio en caso de fallos.

En esta arquitectura, puede haber comunicación entre los servidores para compartir información, coordinar operaciones o asegurar la coherencia de los datos. Esta interacción puede ocurrir a través de la misma red que conecta a los clientes y los servidores, o de manera interna. El intercambio de datos entre servidores es esencial en sistemas más complejos que necesitan cooperación entre diferentes componentes para brindar servicios de manera eficiente y coherente.



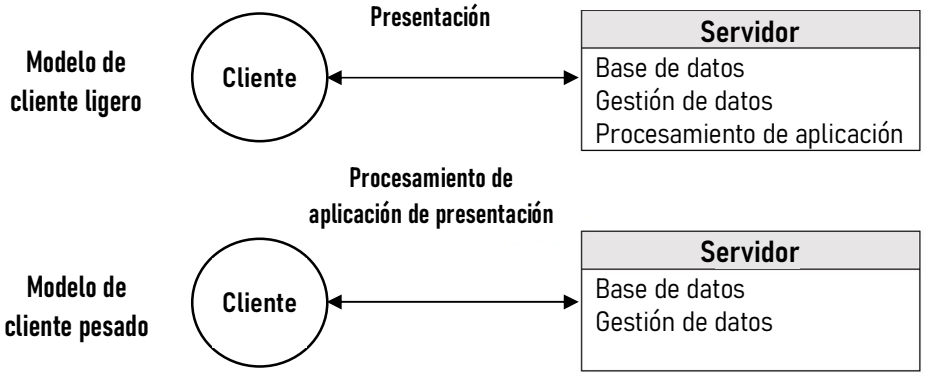
ARQUITECTURA CLIENTE-SERVIDOR | DOS NIVELES

Cliente ligero

El cliente se enfoca en la presentación de la información. El procesamiento y la gestión de datos más complejos o de aplicación son realizados principalmente por el servidor. El cliente ligero es responsable de la interfaz de usuario y la presentación visual de los datos. El cliente es "ligero" en términos de la carga de trabajo que realiza.

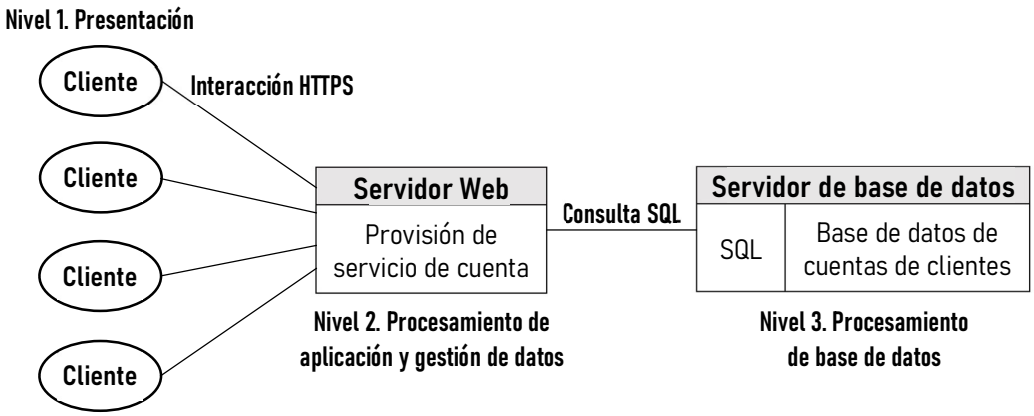
Cliente pesado

El proceso cliente no solo se ocupa de la presentación, es además quien implementa la lógica de la aplicación. Se encarga de parte del procesamiento de la aplicación y la lógica de presentación, mientras que el servidor se concentra en la gestión de datos. Este enfoque "pesado" implica que el cliente tiene una carga de trabajo más sustancial en comparación con el enfoque ligero.



ARQUITECTURA CLIENTE-SERVIDOR | MULTINIVEL

En esta clasificación, le presentación, el procesamiento y la gestión de datos se separan en niveles lógicos que pueden ejecutarse en sistemas diferentes. Puede haber varios niveles de servidores involucrados en esta arquitectura, cada uno con su propio conjunto de responsabilidades.

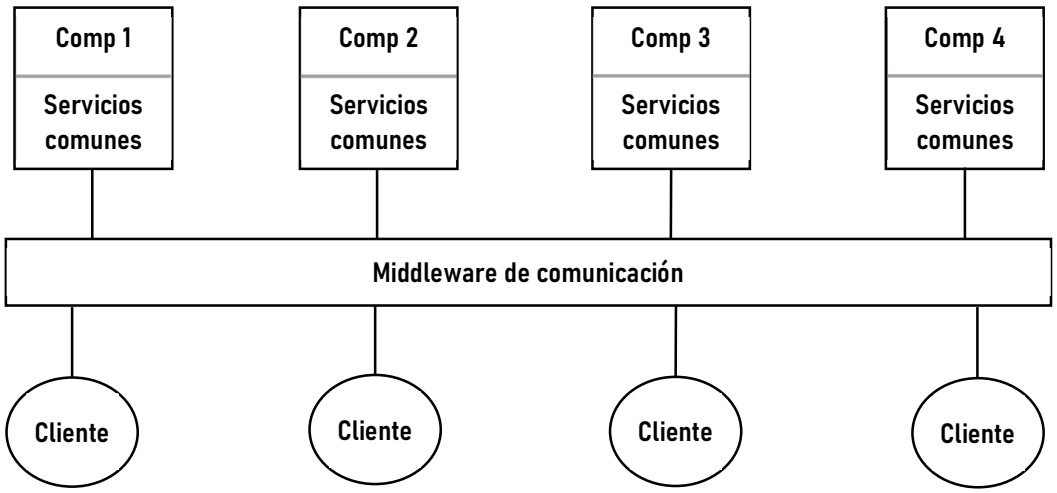


ARQUITECTURA DE COMPONENTES DISTRIBUIDOS

La arquitectura de Componentes Distribuidos se basa en el diseño de sistemas como conjuntos de componentes u objetos que ofrecen una interfaz para acceder a un conjunto de servicios que proporcionan. A diferencia de la arquitectura Cliente-Servidor, aquí no existe una distinción rígida entre clientes y servidores, sino que todos los componentes son considerados como objetos distribuidos.

En esta arquitectura, los componentes pueden residir en diferentes máquinas y equipos de cómputo, conectados a través de una red. Para gestionar la comunicación entre estos componentes distribuidos, se utiliza un middleware como intermediario de las peticiones. Este middleware facilita la comunicación y la interacción entre los componentes, permitiendo que los objetos distribuidos se comuniquen y colaboren entre sí.

El cliente es un componente que solicita servicios a otros componentes distribuidos. A su vez también puede actuar como componente ofreciendo servicios. El middleware actúa como un intermediario entre los componentes distribuidos. Facilita la comunicación, el enrutamiento de las peticiones y la gestión de la transferencia de datos entre los objetos distribuidos. Los servicios son componentes u objetos que ofrecen funcionalidades específicas a través de sus interfaces.



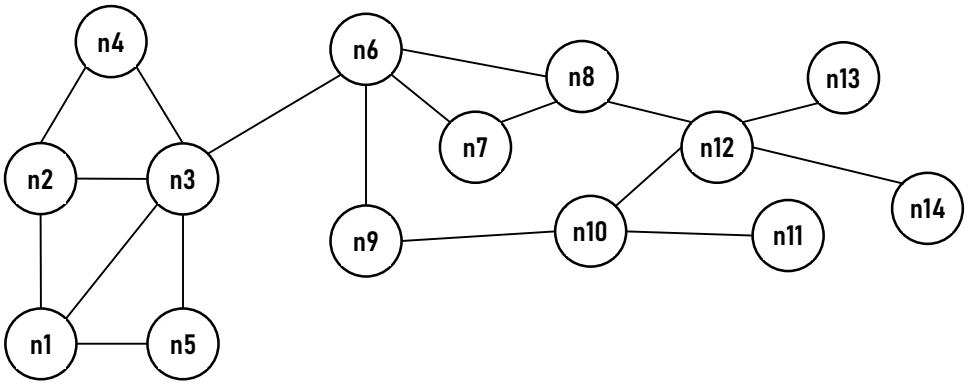
COMPUTACIÓN DISTRIBUIDA INTER-ORGANIZACIONAL

La Computación Distribuida Inter-Organizacional se refiere a la distribución de tareas computacionales entre varios servidores de diferentes organizaciones. En este enfoque, una organización puede distribuir la carga de trabajo entre sus propios servidores, y también colaborar con otras organizaciones para compartir recursos y tareas computacionales.

En los sistemas Peer-to-Peer, la computación se puede realizar en cualquier nodo de la red, y no hay un nodo centralizado que controle todo el proceso. Es similar a un enfoque de cliente pesado, donde cada nodo puede actuar como cliente y servidor a la vez. Este enfoque se aprovecha de la potencia de cálculo y almacenamiento distribuido a través de la red.

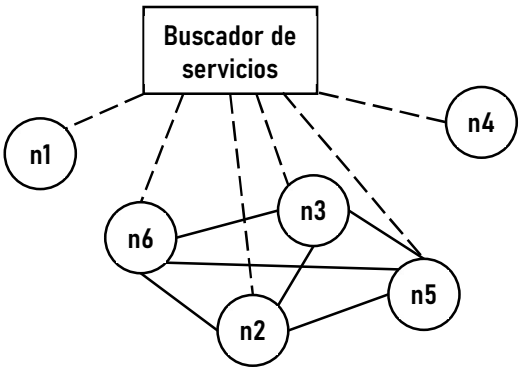
Descentralizados

No hay un sistema o nodo central que tenga información, controle la red o ayude a los nodos a comunicarse entre sí. Cada nodo puede enrutar paquetes hacia sus vecinos hasta que lleguen a su destino. El procesamiento y la toma de decisiones se realizan de manera distribuida entre los nodos de la red. Cada nodo actúa como un igual y puede contribuir al procesamiento y almacenamiento de datos, eliminando la necesidad de una entidad central.



Semi-Centralizados

Puede haber un servidor central que desempeña un papel de coordinación o soporte para los nodos en la red. Este servidor ayuda a conectar nodos entre sí o a coordinar resultados, pero no actúa como una entidad central que controla toda la red. El procesamiento principal y la inteligencia aún residen en los nodos distribuidos(descentralizado), lo que permite un enfoque más equilibrado y escalable.



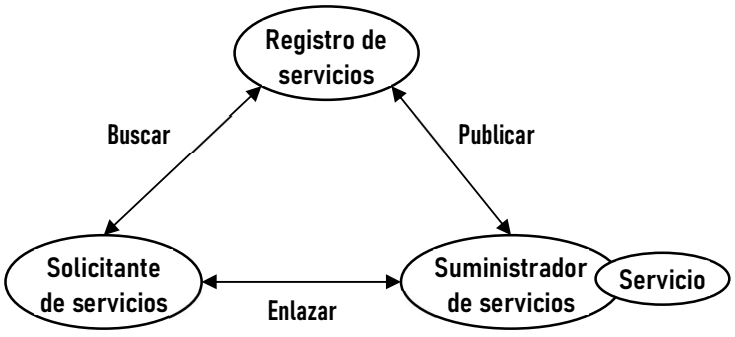
Los sistemas de intercambio de archivos a través de torrents son un ejemplo de P2P. Cada usuario que descarga o comparte un archivo desencadena su participación como un "nodo" en la red.

La arquitectura orientada a servicios se basa en la representación de recursos computacionales o de información como servicios que pueden ser utilizados por diferentes programas. Estos servicios son independientes de las aplicaciones que los utilizan y pueden ser compartidos entre múltiples organizaciones.

En esta arquitectura, las aplicaciones se construyen al enlazar diferentes servicios, formando así una composición de estos elementos. Las aplicaciones basadas en servicios web suelen tener una baja dependencia entre los componentes, lo que se traduce en una estructura débilmente acoplada.

El funcionamiento de esta arquitectura implica que un proveedor de servicios ofrece sus servicios a través de una interfaz definida. Estos servicios pueden ser utilizados por solicitantes externos, que los enlazan a sus propias aplicaciones. Para hacer que un servicio sea visible y accesible externamente, el proveedor lo registra en un "registro de servicio" junto con información relevante. Esto actúa como una especie de catálogo que detalla qué servicios ofrece la organización y cuál es su interfaz.

Cuando un solicitante necesita utilizar un servicio, enlaza ese servicio a su propia aplicación. Esto implica que el solicitante incorpora el código necesario para invocar el servicio y procesar su resultado que este le devuelve. Esta arquitectura fomenta la reutilización y la interoperabilidad de componentes, ya que los servicios pueden ser compartidos y combinados para construir aplicaciones más amplias y flexibles.



IMPLEMENTACIÓN

Una vez establecido el diseño, la siguiente etapa es la codificación del mismo. Esto implica la escritura de los programas que implementarán dicho diseño.

IMPLEMENTACIÓN | COMPLEJIZACIÓN DEL CÓDIGO

Este proceso puede presentar ciertas complejidades por diversas razones:

- **Falta de experiencia del equipo:** Los diseñadores pueden no haber tenido en cuenta las particularidades de la plataforma y el ambiente de programación.
- **Falta de detalle en el diseño:** Las estructuras y relaciones que son fáciles de describir mediante diagramas y representaciones visuales no siempre son tan simples de traducir a código. Esta discrepancia a menudo surge cuando el diseño carece de suficientes detalles.
- **Uso de buenas prácticas y documentación:** Es indispensable escribir el código de forma que resulte comprensible y legible para otras personas. Para lograr esto, es esencial completar documentación adecuada y tener buenas prácticas a la hora de codificar.
- **Aspirar a la reutilización:** Se debe aspirar a crear código que sea reutilizable, aprovechando al máximo las características de diseño. Esto permite que componentes o módulos de código puedan ser utilizados en diferentes partes del software o en otros softwares, lo que ahorra tiempo y esfuerzo durante el desarrollo.

IMPLEMENTACIÓN | PAUTAS GENERALES DE LA CODIFICACIÓN

Para mantener la calidad del diseño durante la etapa de codificación, se pueden adoptar diversas prácticas y enfoques:

- **Localización de entrada y salida:** Es deseable localizarlas en componentes separados del resto del código ya que generalmente son más difíciles de probar.
- **Utilización de pseudocódigo:** Emplear pseudocódigo puede ser beneficioso para traducir el diseño original en el lenguaje de programación elegido.
- **Revisión y reescritura:** Es aconsejable crear un borrador inicial del código, revisarlo en busca de mejoras y reescribirlo tantas veces como sea necesario. En lugar de realizar pequeñas correcciones, se recomienda repensar y reestructurar partes del código para garantizar su eficacia y comprensibilidad.
- **Reutilización productiva:** Se crean componentes nuevos con la intención de que puedan ser reutilizados por otra aplicación, parte del programa u otro sistema.
- **Reutilización consumidora:** Se usan componentes ya existentes originalmente desarrollados para otros proyectos. En lugar de crear algo nuevo desde cero, se utilizan componentes que están disponibles en repositorios o bibliotecas y responden a la misma necesidad.

La documentación en la codificación consiste en un conjunto de descripciones escritas que explican tanto la funcionalidad como el funcionamiento del programa. Esta documentación desempeña un papel fundamental al permitir que los lectores comprendan qué hace el programa y cómo lo logra. Su utilidad radica en facilitar futuras referencias, ampliaciones y correcciones del código. La documentación se compone de dos aspectos principales:

Documentación interna

Esta documentación está dirigida a los programadores y es concisa. Es escrita a nivel técnico y se enfoca en explicar los detalles del código fuente. Esto incluye descripciones de algoritmos, estructuras de control, flujos de ejecución y otros aspectos técnicos. La documentación interna es esencial para que los desarrolladores puedan entender y colaborar en el código existente, facilitando la corrección de errores y la ampliación del programa en el futuro.

Documentación externa

Se prepara para ser leída por quienes, tal vez, nunca verán el código real. Esto incluye a diseñadores, evaluadores o personas que trabajan en mejoras o modificaciones del programa. La documentación externa se centra más en las interfaces y funcionalidades generales del programa, permitiendo que aquellos que no están familiarizados con los detalles técnicos puedan entender cómo interactuar con el software y qué resultados esperar.

DISEÑO DE INTERFAZ DE USUARIO

El diseño de una interfaz de usuario (UI - User Interface) se centra en crear la manera en que los usuarios interactúan con computadoras, aplicaciones, dispositivos móviles y sitios web. Su enfoque principal es la experiencia del usuario y cómo se produce la comunicación entre la persona y la tecnología.

Normalmente es una actividad multidisciplinar que involucra a varias ramas del diseño y el conocimiento como el diseño gráfico, industrial, web, de software y la ergonomía. Se aplica en un amplio rango de proyectos, desde sistemas para computadoras, vehículos hasta aviones comerciales

OBJETIVO

El objetivo central de la Interfaz de Usuario (UI) es lograr una interacción atractiva y centrada en los usuarios. En esta búsqueda, el diseño gráfico y el diseño industrial se aplican para agilizar el aprendizaje de los usuarios sobre cómo operar el software. Estos campos se apoyan en recursos como gráficos, pictogramas, elementos estéticos y simbología, siempre asegurando que no comprometan la eficiencia técnica del sistema.

CONCEPTOS INICIALES

El diseño de la interfaz del usuario se ubica en la categoría de diseño que crea un medio de comunicación entre el hombre y la máquina. A través de la aplicación de un conjunto de principios de usabilidad, se desarrolla un formato de pantalla óptimo. Es esencial comprender las preferencias de los usuarios para desarrollar tecnología que se adapte a las necesidades humanas. No obstante, en la actualidad, a menudo se enfoca exclusivamente en la tecnología, lo que resulta en la exigencia de que las personas se adapten a ella en lugar de lo contrario. Es la tecnología la que debería adaptarse a las personas.

Evitar que los usuarios cometan errores al interactuar con la interfaz es crucial. Una interfaz compleja conduce a errores y puede llevar a que los usuarios eviten su uso por completo. Dado que las personas tienen diversos estilos de percepción, comprensión y trabajo, la interfaz debe ser inclusiva y permitir un acceso rápido al contenido sin sacrificar la comprensión. La interfaz debe facilitar el acceso eficiente a sistemas complejos, manteniendo la comprensión mientras el usuario navega por la información.

Se debe tener en cuenta la diversidad de tecnologías que deben integrarse para adaptarse al usuario, como hipertexto, sonido, presentaciones tridimensionales, video y realidad virtual. Además, se deben considerar diferentes configuraciones de hardware, como teclados, ratones, dispositivos de presentación gráfica, lápices digitales, anteojos de realidad virtual y reconocimiento de voz. Finalmente, el diseño de la interfaz debe ser adaptable a una variedad de dispositivos, como PCs, dispositivos específicos, teléfonos celulares y televisores.

LOS 6 PRINCIPIOS PARA EL DISEÑO UI

1. **Enfoque en la Experiencia del Usuario:** Comprender las necesidades y expectativas del usuario es esencial para crear una interfaz efectiva.

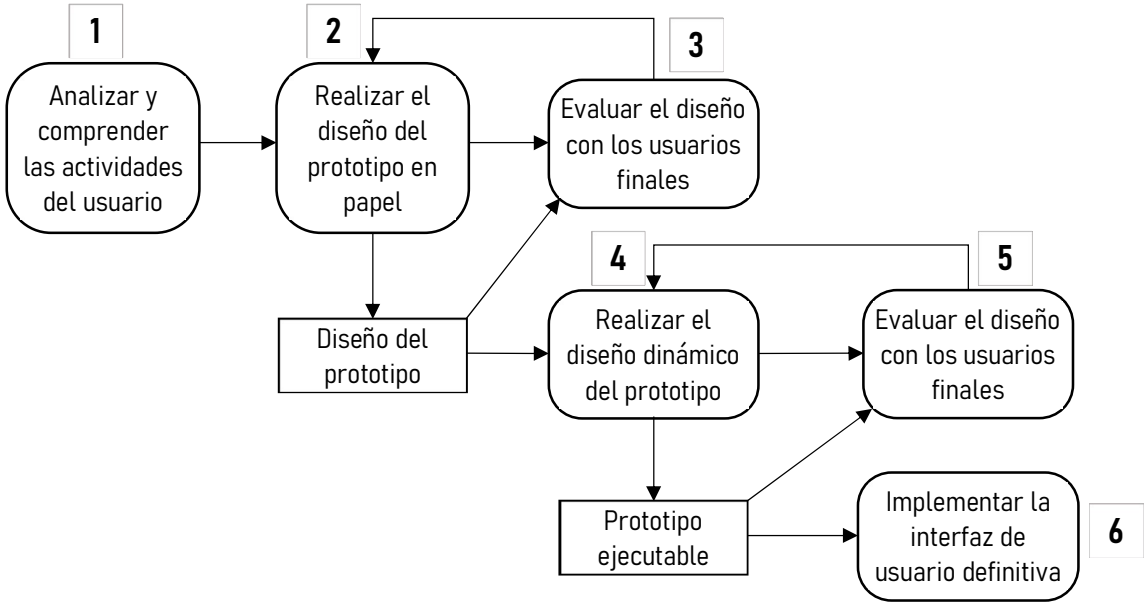
2. **Prototipos en Papel:** El diseño inicial se desarrolla en papel para crear prototipos rápidos y desechables.

3. **Iteración con Usuarios Finales:** Los diseños en papel se evalúan y refinan con los usuarios finales, permitiendo iteraciones antes de avanzar.

4. **Diseño Dinámico:** Los prototipos se convierten en diseños dinámicos utilizando software o herramientas como PowerPoint para simular interacciones.

5. **Evaluación Continua:** Los diseños dinámicos se someten a evaluación y refinamiento con los usuarios, posibilitando ajustes antes de avanzar.

6. **Prototipo Funcional:** Una vez se logra consenso, se desarrolla un prototipo funcional en el lenguaje final de implementación.

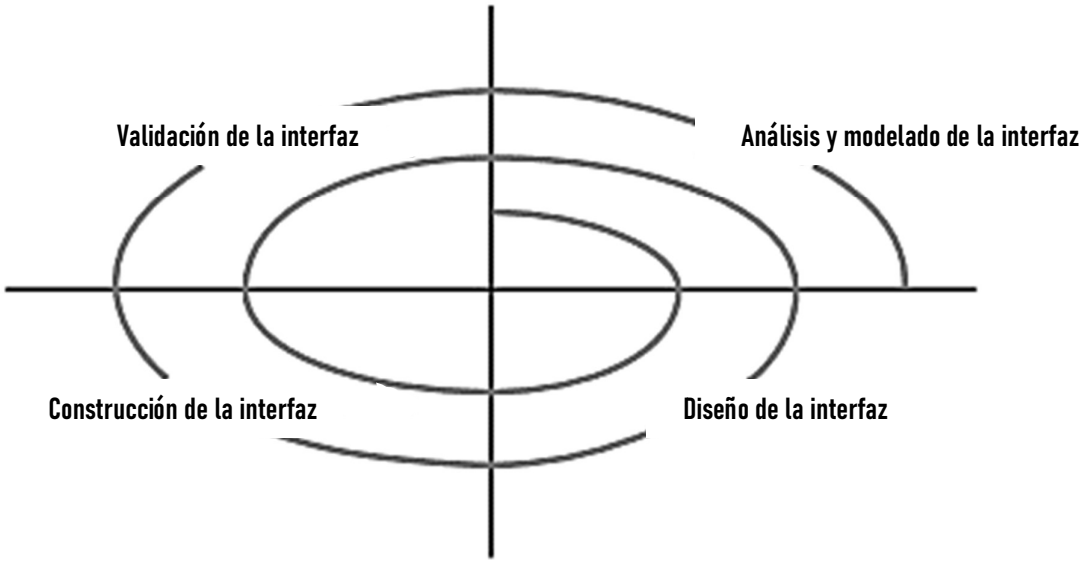


1. **Análisis y Modelado**
   Se definen los elementos y acciones de la interfaz, basándose en la información recopilada en el análisis previo. Se consideran tanto la naturaleza de la interfaz como las características del usuario.

2. **Diseño de la Interfaz**
   Se establecen los eventos o acciones que los usuarios realizarán para cambiar el estado de la interfaz. Este comportamiento se modela, ya sea en papel o mediante programas de diseño.

3. **Construcción de la Interfaz**
   Cada estado de la interfaz se ilustra de manera que coincida con la experiencia del usuario final. Este proceso se basa en los diseños previos.

4. **Validación**
   Se evalúa cómo el usuario interpreta el estado del sistema a partir de la información proporcionada por la interfaz. Esta fase es fundamental para asegurarse de que la interfaz cumple con su propósito.



Este ciclo se repite hasta que se logre una interfaz efectiva y eficiente que satisfaga las necesidades del usuario y cumpla con sus expectativas.

DISEÑO DE EXPERIENCIA DE USUARIO

El diseño de experiencias de usuario (UX) es un conjunto de métodos aplicados al proceso de diseño que buscan satisfacer las necesidades del cliente y proporcionar una buena experiencia a los usuarios destinatarios.

Se enfoca en la creación de productos y servicios con el objetivo de proporcionar interacciones significativas y satisfactorias para los usuarios. Implica considerar todos los aspectos de la interacción del usuario con un producto o servicio, desde la primera impresión hasta el uso continuo. Esto incluye factores como la usabilidad, la accesibilidad, el diseño visual, la arquitectura de la información y la empatía con las necesidades del usuario.

El objetivo principal del UX es proporcionar a los usuarios un entorno intuitivo, claro y de fácil comprensión, evitando transiciones bruscas que puedan generar desorientación. Su enfoque radica en crear interacciones fluidas y agradables que se alineen con las expectativas y necesidades de los usuarios.

TIPOS DE DISEÑO UX

Diseño de Interacción

Enfocado en las formas en que los usuarios interactúan con un producto o servicio, con el objetivo de hacer que la experiencia sea agradable y efectiva.

Diseño Visual

Se encarga de la apariencia visual de un producto o servicio, asegurándose de que sea estéticamente atractivo, eficiente y capaz de transmitir información de manera efectiva. Esto incluye aspectos como el equilibrio, el espacio, el contraste, el color, la forma y el tamaño.

Investigación del Usuario

Implica el estudio exhaustivo de las necesidades, deseos y comportamientos de los usuarios para comprender cómo se relacionarán con el producto o servicio. Esto ayuda a diseñar soluciones que sean relevantes y significativas para los usuarios.

Arquitectura de la Información

Se refiere a la organización y estructuración del contenido de manera que los usuarios puedan acceder y encontrar la información de manera eficiente. Un ejemplo es el mapa del subte de CABA, que ayuda a los usuarios a navegar por las estaciones y líneas de manera clara.

ETAPAS DE DISEÑO UX

1. **Investigación**
   En esta fase se recopila información detallada sobre el proyecto y el producto a diseñar. Se buscan insights (descubrimientos valiosos y reveladores) sobre las necesidades, deseos y comportamientos de los usuarios, así como análisis competitivos y otros datos relevantes.

2. **Organización**
   Los datos obtenidos en la fase de investigación se organizan y estructuran para convertirlos en una base sólida para el diseño. Esto puede incluir la creación de mapas de empatía, mapas de experiencia del usuario y otros artefactos que ayuden a visualizar los puntos clave.

3. **Diseño**
   En esta etapa se crea el diseño propuesto del producto en base a la información organizada. Se crean prototipos, se definen flujos de interacción y se establecen las interfaces. Es importante que el diseño refleje los objetivos del proyecto y las necesidades de los usuarios.

4. **Prueba**
   El diseño propuesto se somete a pruebas y evaluaciones para verificar su eficacia y su capacidad para satisfacer las necesidades de los usuarios. Se pueden realizar pruebas de usabilidad, pruebas de funcionalidad y otros métodos para asegurarse de que el diseño sea efectivo.

Cada una de estas etapas puede involucrar iteraciones y ajustes a medida que se obtienen más insights y se refina el diseño. El resultado final suele ser la creación de artefactos o entregables.

INVESTIGACIÓN DE USUARIOS

La investigación de usuarios implica la recopilación de información de diversas fuentes para comprender a fondo a los usuarios y sus necesidades. Algunas de las fuentes comunes de información incluyen:

- **Encuestas:** Se utilizan para obtener datos cuantitativos y cualitativos directamente de los usuarios.
- **Información de ventas:** Analizar datos de ventas puede proporcionar ideas sobre qué productos o características son más populares entre los usuarios.
- **Información de mercadotecnia:** Los datos de marketing, como el análisis de segmentos de mercado y la demografía, ayudan a comprender quiénes son los usuarios y cómo se relacionan con el producto.
- **Charlas de apoyo al usuario:** Las conversaciones con el equipo de soporte al cliente pueden revelar problemas recurrentes, dificultades y comentarios de los usuarios.

INVESTIGACIÓN DE USUARIOS | PERFIL DE USUARIO

Al crear un perfil de usuario sólido, es esencial recopilar información detallada que permita comprender a fondo a la audiencia objetivo. Algunos de los datos relevantes para desarrollar un perfil completo son:

- Franja de edad
- Etnia
- Género
- Experiencia
- Nivel de ingresos
- Idioma
- Nivel de Estudios
- Localización
- Ocupación o profesión
- Religión

Es importante manejar esta información con sensibilidad y respeto, especialmente en lo que respecta a aspectos culturales, religiosos y de género. Cuanta más información y aspectos se recopilen, más afinado estará el perfil de usuario y mejor diseñado estará el producto o servicio para satisfacer sus necesidades.

INVESTIGACIÓN DE USUARIOS | CONTEXTO Y AMBIENTE DE TRABAJO

El proceso de relevamiento del contexto y ambiente de trabajo implica varios pasos para comprender en detalle cómo el sistema será utilizado en situaciones prácticas. Algunos de los aspectos clave a considerar son:

- **Análisis de Actores:** Identificar y comprender a todas las partes involucradas.
- **Evaluación de Competencias del Producto**: Revisar y evaluar las capacidades y características del producto o sistema para determinar cómo se ajusta a las necesidades y expectativas de los usuarios.
- **Recorridos del Usuario:** Realizar recorridos virtuales o físicos para comprender cómo los usuarios interactúan con el sistema en su contexto real.
- **Desglose de Tareas y Subtareas:** Identificar las tareas principales y las subtareas que los usuarios realizarán mientras llevan a cabo su trabajo.
- **Tareas en Circunstancias Específicas**: Analizar qué tareas realizarán los usuarios en situaciones particulares y cómo el sistema puede facilitar estas actividades.
- **Secuencia y Jerarquía de Tareas:** Definir la secuencia en la que se realizan las tareas y establecer la jerarquía de importancia entre ellas.
- **Dominio de Problema Específico:** Comprender qué aspectos del problema o campo de conocimiento específico manejará el usuario durante su labor.

Es importante llevar a cabo esta etapa en paralelo con la generación de la especificación de requerimientos, ya que ambas se complementan para obtener una comprensión completa de cómo el sistema será utilizado en el mundo real.

DIFERENCIAS ENTRE DISEÑO UI Y UX

UI	UX
Se centra en el aspecto visual y la interacción de la pantalla y elementos visibles para el usuario.	Engloba la totalidad de la experiencia del usuario, considerando aspectos emocionales, funcionales y visuales.
Incluye botones, iconos, tipografía, colores y otros elementos visuales en la pantalla.	Aborda aspectos más amplios como la usabilidad, la navegación, la empatía con el usuario y la fluidez de la experiencia.
Busca crear una interfaz estéticamente agradable y fácil de usar para los usuarios.	Busca garantizar una experiencia completa y satisfactoria.

La UI y la UX colaboran para crear una interfaz que funcione de manera efectiva y brinde una experiencia positiva. Ambos trabajan juntos para lograr una experiencia cohesiva y satisfactoria para el usuario. La UI proporciona la base visual, mientras que la UX asegura que la interacción y la experiencia sean óptimas.

ICEBERG UX

El concepto del "Iceberg de UX" nos brinda una perspectiva reveladora sobre la experiencia de usuario en el diseño. Si imaginamos la experiencia como un iceberg, la punta visible representa la interfaz de usuario, la parte que los usuarios ven y con la que interactúan directamente. Sin embargo, esta es solo una fracción de la historia completa.

La verdadera complejidad reside bajo la superficie, fuera de la vista. Aquí es donde reside el diseño más profundo: la arquitectura de la información, la estructura de datos, la usabilidad y otros aspectos técnicos y conceptuales. Estos cimientos invisibles son lo que sostiene y hace que la experiencia de usuario sea efectiva y atractiva.

PRESENTACIÓN DE INFORMACIÓN EN PANTALLA

A través de las pantallas se muestra la información visual y textual de un dispositivo, como una computadora, un teléfono móvil o una tablet. Esta presentación tiene como objetivo principal comunicar de manera efectiva y eficiente la información al usuario, de modo que sea fácil de comprender y utilizar.

PRESENTACIÓN DE INFORMACIÓN EN PANTALLA | ENFOQUE UI

El uso del Modelo-Vista-Controlador (MVC) es una buena práctica en diseño de software que implica separar la lógica de procesamiento (modelo), la presentación visual (vista) y el control de la interacción (controlador). Esto facilita la gestión y el mantenimiento del sistema.

Modos de presentación:

- **Directo:** La presentación directa implica mostrar la información de manera textual o numérica de forma clara y directa.
- **Gráfica:** Utiliza elementos visuales como gráficos y diagramas para representar la información de manera más visual y fácilmente comprensible.

PRESENTACIÓN DE INFORMACIÓN EN PANTALLA | ENFOQUE UI | COLORES

Los colores desempeñan un papel crucial en el diseño de interfaces de usuario. Su elección y aplicación cuidadosas pueden influir en la forma en que los usuarios interactúan con una interfaz, cómo perciben la información y cómo se sienten al usarla. Son una herramienta versátil en el diseño de interfaces de usuario que va más allá de lo meramente estético.

Aspectos a tener en cuenta:

- **Número limitado de colores:** El uso de un número limitado de colores en una ventana o en toda la interfaz ayuda a mantener la coherencia visual y evita la sobrecarga de información. No se deberían utilizar más de 4 o 5 colores diferentes en una ventana, o más de 7 en la interfaz total del sistema.
- **Apoyo a la tarea del usuario:** Los colores deben tener un propósito claro y apoyar las tareas que los usuarios están tratando de llevar a cabo.
- **Coherencia de gama de colores:** Mantener una gama de colores coherente ayuda a crear una sensación de armonía en la interfaz.
- **Cautela al mezclar grupos de colores:** Mezclar colores de manera indiscriminada puede resultar en una interfaz visualmente desordenada y confusa. Es importante considerar cómo los colores interactúan entre sí y asegurarse de que no haya combinaciones que distraigan o dificulten la legibilidad.
- **Cuidado con la saturación y brillo:** Colores demasiado brillantes o saturados pueden causar fatiga visual y dificultar la lectura de la información. Mantener un equilibrio entre colores vivos y tonos más suaves es esencial para una experiencia cómoda.
- **Considerar la percepción de colores:** Alrededor del 10% de la población tiene dificultades para percibir ciertos colores debido a la discromatopsia (daltonismo). Utilizar colores opuestos (en dupla o triada) en el círculo cromático puede ayudar a que los usuarios con esta condición puedan distinguir elementos importantes.

- **Acompañar con información:** Se debe proporcionar con alguna información adicional que garantice que la experiencia sea accesible para todas las personas, independientemente de su capacidad para percibir los colores de manera convencional.
- **Combinación de colores cuidadosa:** Al elegir combinaciones de colores, es esencial considerar cómo se ven juntos y si proporcionan suficiente contraste para una buena legibilidad.
- **Consistencia en el uso de colores:** Los colores deben utilizarse de manera coherente en toda la interfaz. Un mismo color debe representar la misma acción o significado en diferentes partes del sistema para evitar confusión.
- **Cambios de color para mostrar estados:** Utilizar cambios de color para indicar cambios en el estado del sistema, como acciones realizadas con éxito o errores, puede proporcionar retroalimentación visual importante a los usuarios.

PRESENTACIÓN DE INFORMACIÓN EN PANTALLA | ENFOQUE UX

La presentación de información en pantalla busca mejorar la experiencia del usuario al hacer que la interfaz sea intuitiva, atractiva y fácil de usar. Un diseño efectivo se basa en una comprensión profunda de las necesidades y expectativas de los usuarios, así como en principios de diseño visual y usabilidad.

Aspectos a tener en cuenta:

- **Conocimiento del usuario:** Comprender a los usuarios es fundamental. Saber quiénes son, cuáles son sus necesidades, objetivos y cómo interactuarán con el sistema ayuda a diseñar una presentación de información relevante y útil.
- **Información precisa vs. relaciones entre valores:** Dependiendo del contexto, a veces es más importante presentar información precisa (valores exactos) mientras que en otros casos, mostrar relaciones entre valores (gráficos, comparaciones) puede ser más significativo para los usuarios.
- **Inmediatez de los cambios:** En algunos casos, es esencial mostrar los cambios inmediatamente, por ejemplo, en aplicaciones en tiempo real. En otros casos, los cambios pueden ser presentados de manera más asincrónica si no es crítico para la interacción del usuario.
- **Acciones basadas en cambios:** Si los usuarios deben tomar acciones en función de los cambios de información, es importante presentar esos cambios de manera clara y comprensible para que puedan tomar decisiones informadas.
- **Información textual o numérica:** La elección entre información textual o numérica depende de la naturaleza de la información y la facilidad de comprensión. En algunos casos, los números pueden transmitir información de manera más precisa, mientras que en otros, el texto puede ser más claro y comprensible.
- **Información estática o dinámica:** La información puede ser estática (no cambia) o dinámica (cambia con el tiempo o las interacciones del usuario). La presentación debe adaptarse a la naturaleza de la información y las necesidades del usuario.

ASPECTOS DEL DISEÑO DE INTERFACE

Los aspectos del diseño de interfaz son los elementos cruciales que determinan cómo los usuarios interactúan y experimentan un producto digital. Estos aspectos engloban la usabilidad, la estética, la navegación y la interacción, entre otros. Cada uno de ellos desempeña un papel fundamental en la creación de una experiencia de usuario efectiva y atractiva.

LAS 3 REGLAS DORADAS DEL DISEÑO DE THEO MANDEL

Theo Mandel (1997) propone tres reglas fundamentales:

1. Dar control al usuario.
2. Reducir la carga de memoria del usuario.
3. Lograr una interfaz consistente.

Además de estas reglas, es crucial considerar los Factores Humanos en el diseño. En conjunto, estas reglas y consideraciones aseguran que el diseño se enfoque en las necesidades y comodidad de los usuarios.

1. DAR CONTROL AL USUARIO

Brindar control al usuario es una práctica esencial en el diseño de productos y sistemas interactivos. Significa permitir que los usuarios tengan el poder de tomar decisiones y manipular la experiencia según sus necesidades y preferencias. En lugar de imponer restricciones rígidas, se les proporciona opciones claras para que puedan moldear su interacción de manera efectiva y personalizada. El usuario busca un sistema que se adapte a sus necesidades y lo asista en la realización de tareas. Esto se logra al proporcionar un nivel de control que permite al usuario:

- **Definir modos de interacción:** Diseñar modos de interacción que minimicen las acciones innecesarias por parte del usuario y faciliten el cumplimiento de tareas.
- **Interacción flexible:** Permitir una interacción flexible que se ajuste a los diferentes estilos y preferencias del usuario.
- **Opciones de interrumpir y deshacer:** Proporcionar opciones para interrumpir acciones y deshacer cambios, lo que brinda un mayor sentido de seguridad y control.
- **Depurar según destreza:** A medida que el usuario gana experiencia, depurar la interacción para que se adapte a su nivel de habilidad y conocimiento.
- **Ocultar elementos técnicos:** Ocultar elementos técnicos internos a usuarios ocasionales para no abrumarlos con detalles innecesarios.
- **Interacción directa:** diseñar interacciones directas con los objetos en pantalla, lo que simplifica el proceso y aumenta la sensación de control.

2. REDUCIR LA CARGA MENTAL DEL USUARIO

Reducir la carga de memoria del usuario es esencial para crear productos y sistemas que sean intuitivos y fáciles de usar. El objetivo es diseñar de tal manera que los usuarios no tengan que recordar información complicada o realizar esfuerzos mentales excesivos para interactuar con el producto. Para lograrlo, se pueden considerar las siguientes prácticas:

- **Reducir la demanda a corto plazo:** Evitar abrumar al usuario con una gran cantidad de información o decisiones al mismo tiempo.
- **Definir valores por defecto significativos:** Utilizar valores predeterminados que tengan sentido para la mayoría de los usuarios, de modo que no sea necesario cambiarlos a menos que se desee.
- **Definir accesos directos intuitivos:** Proporcionar atajos y accesos directos que sean fáciles de recordar y utilizar.
- **Metáforas visuales de la realidad:** Diseñar la interfaz visual utilizando metáforas que se asemejen a objetos y conceptos del mundo real.
- **Desglosar la información de manera progresiva:** Presentar la información de manera gradual y progresiva a medida que el usuario avanza en las tareas.

3. LOGRAR UNA INTERFAZ CONSISTENTE

Mantener la coherencia en el diseño y la interacción en todo el producto es un principio fundamental para garantizar una experiencia de usuario fluida y efectiva. La consistencia permite que los usuarios se familiaricen con patrones y flujos, lo que facilita su uso y minimiza la confusión. Para lograr una interfaz consistente, se pueden aplicar las siguientes estrategias:

- **Contextualización de la tarea actual:** Permitir que el usuario comprenda la tarea actual en un contexto más amplio que tenga significado.
- **Orientación y dirección:** Dar al usuario la capacidad de entender de dónde proviene y hacia dónde puede ir en el sistema.
- **Consistencia en la familia de aplicaciones:** Mantener una apariencia y un comportamiento coherentes en todas las aplicaciones relacionadas.
- **Reglas de diseño constantes:** Utilizar las mismas reglas y patrones de diseño para interacciones similares en todo el producto.
- **Mantener modelos prácticos:** A menos que sea absolutamente necesario, mantener modelos y patrones que sean familiares y prácticos para el usuario.
- **Uniformidad en estilo:** Mantener un estilo visual consistente en toda la interfaz para que los usuarios siempre sepan qué esperar y dónde buscar información.

4. FACTORES HUMANOS

Integrar los principios ergonómicos y de usabilidad es esencial para diseñar productos y sistemas que se adapten a las capacidades físicas y cognitivas de los usuarios. Esto implica considerar aspectos como la facilidad de uso, la accesibilidad y la comodidad para garantizar una experiencia satisfactoria y eficaz. Algunos de los factores humanos que deben ser tenidos en cuenta en el diseño incluyen:

- **Percepción visual/auditiva/táctil:** Diseñar interfaces que sean visibles, audibles y táctiles de manera efectiva para que los usuarios puedan interactuar de manera cómoda y precisa.
- **Memoria humana:** Reconocer las limitaciones de la memoria humana y diseñar de manera que la información importante sea fácil de recordar y recuperar. Existe una regla que sostiene que un individuo promedio puede retener en su memoria a corto plazo aproximadamente entre 5 y 9 elementos, con un valor medio de alrededor de 7.
- **Razonamiento:** Diseñar interfaces que sigan patrones lógicos y flujos de trabajo intuitivos para que los usuarios puedan tomar decisiones informadas sin confusión.
- **Capacitación:** Facilitar la curva de aprendizaje para que los usuarios puedan utilizar el producto sin una capacitación extensa. El diseño debe ser intuitivo y autoexplicativo.
- **Comportamiento/habilidades personales:** Considerar las capacidades y habilidades individuales de los usuarios al diseñar interacciones. Esto garantiza que tanto usuarios novatos como expertos encuentren el producto utilizable.
- **Diversidad de usuarios:** Reconocer la diversidad de los usuarios y diseñar para atender a diferentes grupos, incluyendo personas con discapacidades, diferentes niveles de experiencia y necesidades específicas.
- **Usuarios casuales vs. experimentados:** Adaptar el diseño para usuarios casuales que necesitan guía y apoyo, así como para usuarios experimentados que buscan interfaces rápidas y ágiles.

USABILIDAD

La usabilidad se refiere a la medida en que un producto, sistema o servicio puede ser utilizado por los usuarios de manera efectiva, eficiente y satisfactoria. En otras palabras, se trata de la facilidad con la que los usuarios pueden interactuar con algo para lograr sus objetivos de manera exitosa.

USABILIDAD | CONCEPTO

Donahue la define: "La usabilidad es una medida de cuán bien un sistema de cómputo [...] facilita el aprendizaje, ayuda a quienes lo emplean a recordar lo aprendido, reduce la probabilidad de cometer errores, les permite ser eficientes y los deja satisfechos con el sistema."

La usabilidad no surge de la mera estética, sino de la implementación de mecanismos de interacción avanzados o interfaces inteligentes. En lugar de eso, se logra cuando la arquitectura de la interfaz se adapta a las necesidades de las personas que la utilizan. La usabilidad es la justa y necesaria interacción entre el usuario y la interfaz.

Definir la usabilidad de manera formal es ilusorio. Forma parte de la semántica del software y en las necesidades individuales de las personas. Por lo tanto, lograr que una interfaz satisfaga a todos es un desafío complejo, pero es posible esforzarse por hacerla utilizable para la mayoría.

USABILIDAD | FACTORES QUE DETERMINAN LA USABILIDAD

La usabilidad en un sistema se determina a través de evaluaciones y pruebas que involucran la interacción de los usuarios con el producto. Durante este proceso, se consideran una serie de criterios que definen si la usabilidad está presente.

Para saber si existe usabilidad, los usuarios interactúan con el sistema y responden a una serie de preguntas en base a su experiencia:

- ¿El sistema es utilizable sin ayuda o enseñanza continua?
- ¿Las reglas de interacción ayudan a un usuario preparado a trabajar con eficiencia?
- ¿Los mecanismos de interacción se hacen más flexibles a medida que los usuarios conocen más?
- ¿Se ha adaptado el sistema al ambiente físico y social en el que se usará?
- ¿El usuario está al tanto del estado del sistema? ¿Sabe en todo momento dónde está?
- ¿La interfaz está estructurada de manera lógica y consistente?
- ¿Los mecanismos, iconos y procedimientos de interacción son consistentes en toda la interfaz?
- ¿La interacción prevé errores y ayuda al usuario a corregirlos?
- ¿La interfaz es tolerante a los errores que se cometen?
- ¿Es sencilla la interacción?

Si cada uno de estos puntos refleja una respuesta positiva en la evaluación, es probable que el sistema haya alcanzado un nivel adecuado de usabilidad, lo que garantiza una experiencia positiva y satisfactoria para los usuarios.

Jacob Nielsen, un influyente experto en usabilidad, ha propuesto una serie de principios de diseño que establecen las pautas para el diálogo efectivo que debe proporcionar una interfaz de usuario. Estos principios son ampliamente utilizados en el diseño y la evaluación de interfaces, y aunque se originaron para interfaces textuales, siguen siendo fundamentales para diseñar diversos tipos de interfaces.

1. DIÁLOGO SIMPLE Y NATURAL

Este principio se refiere a cómo se lleva a cabo la comunicación entre el usuario y la interfaz para lograr un diálogo eficiente y natural. Al asegurarse de que el diálogo entre el usuario y la interfaz sea simple, claro y correcto, se crea una experiencia más fluida y satisfactoria para el usuario

Estrategias:

- **Escritura correcta:** Se debe garantizar que la información presentada esté libre de errores de escritura o tipografía.
- **Separación de información:** Es fundamental evitar mezclar información importante con datos irrelevantes.
- **Distribución Adecuada:** La información debe estar organizada y distribuida de manera lógica en la interfaz.
- **Prompts lógicos:** Los mensajes y preguntas dirigidos al usuario (prompts) deben estar diseñados de manera coherente y lógica, guiando al usuario en sus acciones.
- **Evitar mayúsculas y abreviaturas excesivas:** El uso exagerado de mayúsculas y abreviaturas puede dificultar la lectura y comprensión.
- **Unificación de funciones predefinidas:** Mantener la consistencia en el uso de funciones predefinidas para acciones comunes.

2. LENGUAJE DE USUARIO

Este principio enfatiza el uso de un lenguaje familiar para el usuario en la interfaz, y en evitar términos técnicos o confusos. Se logra así una comunicación efectiva entre el usuario y la interfaz. Esta última debe hablar el lenguaje del usuario y reflejar el mundo real, utilizando términos, conceptos y acciones familiares.

Estrategias:

- **Evitar Términos Técnicos o Extranjeros:** Se debe evitar el uso de jerga técnica o palabras en otros idiomas que los usuarios no comprendan.
- **Truncamiento Adecuado**: Evitar acortar palabras de manera excesiva, ya que esto puede dificultar la comprensión.
- **Diseño de Entradas de Datos:** Diseñar adecuadamente los campos de entrada de datos para que sean claros y proporcionen pistas sobre qué información se espera.
- **Información en la Medida Justa:** Proporcionar la cantidad apropiada de información. No abrumar al usuario con detalles excesivos ni dejarlo confundido con información insuficiente.

3. MINIMIZAR EL USO DE LA MEMORIA DEL USUARIO

Este principio se centra en reducir la carga cognitiva del usuario, evitando que tenga que esforzarse para recordar información o realizar acciones. Al minimizar la carga de memoria del usuario y brindar información de contexto y soporte, se crea una experiencia más cómoda y fluida.

Estrategias:

- **Información de Contexto:** Proporcionar información contextual que guíe al usuario y le ayude a comprender dónde se encuentra en el sistema y cómo interactuar con él.
- **Información de Navegación y Sesión:** Mantener al usuario informado sobre su ubicación en el sistema y el estado actual de su sesión, lo que evita que tenga que recordar detalles.
- **Visualización de Rangos Admisibles:** Mostrar de manera clara los rangos de valores o datos que son aceptables en campos de entrada. También proporcionar ejemplos y formatos para facilitar la entrada correcta.

4. CONSISTENCIA

La consistencia es esencial para evitar ambigüedad y confusiones en la interfaz, tanto en su aspecto visual como en su funcionamiento. Mantener la coherencia brinda confiabilidad y seguridad al sistema y promueve una experiencia de usuario más fluida, evitando que los usuarios se sientan desorientados. Además, construye una sensación de familiaridad que facilita la interacción y la comprensión de la interfaz.

Estrategias:

- **Consistencia Visual:** Asegurar que los elementos de diseño como colores, tipografías y estilos gráficos sean coherentes en toda la interfaz.
- **Consistencia Tecnológica:** Garantizar que las interacciones y las funciones se comporten de manera predecible y coherente en todo el sistema.
- **Consistencia Terminológica:** Utilizar el mismo lenguaje y términos a lo largo de la interfaz para describir acciones y conceptos similares, evitando confusiones.

5. FEEDBACK

El feedback es esencial para mantener a los usuarios informados sobre las acciones que están realizando y el estado del sistema. Proporcionar respuestas visuales o textuales en la pantalla es fundamental para mantener una comunicación efectiva con el usuario. Mantener a los usuarios informados en tiempo real les permite tomar decisiones más seguras y reducir la posibilidad de errores.

Estrategias:

- **Información de Estados de Procesos:** Mantener al usuario al tanto de los estados y progresos de procesos en curso, evitando que se sienta confundido o inseguro.
- **Información del Estado del Sistema y del Usuario:** Brindar información clara sobre el estado actual del sistema y cómo afecta al usuario. Esto ayuda a evitar malentendidos.

- **Mensajes de Aclaración, Validaciones, Confirmación y Cierre:** Utilizar mensajes adecuados para aclarar dudas, validar entradas de datos, confirmar acciones importantes y cerrar procesos.
- **Validación de Datos Ingresados:** Realizar verificaciones de los datos ingresados por el usuario para evitar errores y proporcionar retroalimentación inmediata en caso de problemas.

6. SALIDAS EVIDENTES

Es importante garantizar que el usuario siempre tenga opciones claras y accesibles para abandonar una pantalla, contexto, acción o tarea. Los usuarios deben poder controlar y deshacer acciones no deseadas, brindándoles una vía de escape en caso de cometer errores o simplemente querer salir del sistema. Esto previene frustraciones y ayuda a crear una experiencia en la que los usuarios se sientan cómodos y confiados.

Estrategias:

- **Salidas en Cada Pantalla:** Proporcionar una opción evidente para salir o navegar a otra parte de la aplicación en cada pantalla.
- **Salidas para Cada Contexto:** En cada contexto o sección, asegurarse de que los usuarios puedan regresar fácilmente a la navegación principal o a otras áreas relevantes.
- **Salidas para Cada Acción, Tarea o Transacción:** Después de completar una acción o tarea, brindar opciones claras para continuar o salir, según las necesidades del usuario.
- **Salidas en Cada Estado:** Sea cual sea el estado en el que se encuentre el usuario, asegurarse de que tenga acceso a opciones de salida claras.
- **Visualización de Opciones Varias:** Mostrar de manera clara opciones como cancelar, salir, suspender, deshacer, modificar, etc.

7. MENSAJES DE ERROR

Los mensajes de error son una parte crucial de la retroalimentación del sistema, ya que informan sobre problemas y proporcionan una guía para resolverlos. Ayudan a los usuarios a superar obstáculos y a sentir que el sistema les está proporcionando el soporte necesario para resolver cualquier inconveniente.

Estrategias:

- **Mensajes de Acuerdo al Contexto:** Asegurarse de que haya mensajes de error diseñados para situaciones relevantes y que guíen al usuario hacia una solución.
- **Brindar Información Detallada:** Los mensajes de error deben explicar claramente el problema y ofrecer alternativas para corregirlo.
- **Categorización de Mensajes:** Clasificar los mensajes de error según su gravedad y relevancia para facilitar la identificación y solución de problemas.
- **Evitar Mensajes Intimidatorios:** Los mensajes de error no deben asustar ni intimidar al usuario. Deben ser claros y constructivos en lugar de culpar al usuario.
- **Gestión Adecuada de Aparición:** Los mensajes de error deben aparecer en el momento y lugar adecuados, sin interrumpir bruscamente la tarea del usuario.

8. PREVENSIÓN DE ERRORES

Evitar que los usuarios lleguen a situaciones de error es fundamental para garantizar una experiencia de usuario fluida, sin frustraciones ni obstáculos innecesarios. Al guiar a los usuarios hacia el camino correcto y ofrecer correcciones automáticas, se reducen las posibilidades de errores.

Estrategias:

- **Rangos de Entradas Posibles:** Ofrecer opciones de selección en lugar de requerir que los usuarios escriban información. Esto reduce la posibilidad de errores tipográficos y asegura que se introduzcan datos válidos.
- **Ejemplos y Valores por Defecto:** Mostrar ejemplos de entradas válidas, así como valores predefinidos, para guiar a los usuarios y evitar confusiones.
- **Formatos de Entrada Admisibles:** Indicar claramente los formatos aceptados para las entradas (por ejemplo, fechas, números de teléfono) y validar automáticamente los datos ingresados.
- **Mecanismos de Corrección Automática:** Proporcionar correcciones automáticas o sugerencias cuando los usuarios ingresan datos incorrectos, reduciendo la necesidad de intervención manual.
- **Flexibilidad en las Entradas:** Permitir cierta flexibilidad en las entradas, como aceptar mayúsculas o minúsculas en campos de texto, para acomodar diferentes estilos de entrada.

9. ATAJOS

Proporcionar atajos y opciones alternativas en la interfaz es una práctica esencial para adaptarse a las distintas necesidades de usuarios, tanto novatos como experimentados. Al ofrecer estas alternativas, se brinda a los usuarios la capacidad de interactuar con la interfaz de manera más eficiente y cómoda, según sus preferencias y niveles de experiencia.

Estrategias:

- **Mecanismos Alternativos de Interacción:** Ofrecer formas alternativas para que los usuarios puedan interactuar con el sistema de manera más rápida y eficiente, como atajos de teclado.
- **Reorganización de Elementos:** Permitir a los usuarios personalizar la disposición de barras de herramientas, menús y otros elementos según sus preferencias y flujo de trabajo.
- **Mecanismos de Macros:** Brindar la capacidad de crear y utilizar macros para automatizar secuencias de acciones, lo que ahorra tiempo y esfuerzo en tareas repetitivas.
- **Atajos y Teclas de Función:** Definir atajos de teclado y teclas de función para acciones frecuentes, facilitando la navegación y la ejecución de tareas.

10. AYUDAS

Las ayudas en una interfaz son componentes vitales para brindar asistencia al usuario. Sin embargo, su diseño debe ser cuidadoso para no obstaculizar la usabilidad. Un diseño cuidadoso de las ayudas garantiza que los usuarios puedan acceder a la información que necesitan de manera efectiva y sin agregar complejidad innecesaria. Las ayudas bien implementadas contribuyen a una experiencia de usuario más fluida y confiable, al tiempo que brindan apoyo en momentos de dificultad.

Estrategias:

- **Presencia de Ayudas:** Asegurarse de que las ayudas estén disponibles para los usuarios en momentos de necesidad, proporcionando opciones para acceder a ellas.
- **Diversidad de Ayudas:** Ofrecer diferentes tipos de ayudas para abordar diversas situaciones, como ayudas generales, contextuales y específicas en línea.
- **Variedad en Formato:** Brindar ayudas en diferentes formatos para acomodar a diversos tipos de usuarios, como texto, imágenes, videos, audio u otros medios.
- **Mecanismos de Asistencia:** Proporcionar diversos mecanismos de asistencia, como búsqueda en línea, soporte en línea a través de chats, correo electrónico de soporte técnico y acceso a preguntas frecuentes.

ESTILOS DE INTERFACE

Los estilos de interfaces se refieren a las diferentes formas en que los usuarios pueden interactuar con un sistema o software. Cada estilo tiene sus propias características y métodos de interacción que pueden variar desde comandos de texto hasta interfaces gráficas más intuitivas. Cada estilo de interfaz tiene sus ventajas y desafíos, y la elección del estilo depende de factores como el tipo de aplicación, el público objetivo y los objetivos de diseño.

ESTILOS DE INTERFACE | INTERFAZ DE COMANDOS

La interfaz de comandos es uno de los estilos más básicos de interacción con sistemas informáticos. En esta interfaz, los usuarios interactúan mediante la entrada de texto a través de una línea de comandos (prompt).

Ventaja:

- **Flexibilidad y poder:** La interfaz de comandos ofrece una amplia funcionalidad y capacidad de realizar tareas diversas a través de comandos específicos.

Desventajas:

- **Gestión de errores pobre:** Los errores pueden ser poco claros y, en algunos casos, pueden resultar en fallos del sistema debido a la falta de manejo de errores.
- **Dificultad de aprendizaje:** Debido a la necesidad de aprender comandos específicos y su sintaxis, la interfaz de comandos puede ser difícil de aprender para usuarios no familiarizados.

Con el tiempo, la interfaz de comandos ha evolucionado para incluir características más amigables. Las interfaces de comandos de tipo pregunta-respuesta brindan una experiencia más guiada al orientar a los usuarios en sus dudas.

Una variante actual combina la interfaz de comandos con una interfaz gráfica, permitiendo el uso de comandos a través de ventanas y menús, como el símbolo del sistema de Windows o la interfaz SQL en la que se pueden ingresar comandos a través de una ventana gráfica.

ESTILOS DE INTERFACE | INTERFAZ DE SELECCIÓN DE MENÚ

La interfaz de selección de menú es otro estilo de interacción que simplifica la forma en que los usuarios interactúan con un sistema informático. En esta interfaz se presentan opciones predefinidas en forma de menú y los usuarios pueden elegir entre estas mediante el ingreso de caracteres o números asociados a cada opción. Esta interfaz suele utilizarse para aplicaciones que realizan tareas específicas y donde las opciones son claras y predefinidas.

Ventaja:

- **Evita errores:** Al limitar las opciones a las presentadas en el menú, se reducen las posibilidades de errores de entrada por parte del usuario.

Desventaja:

- **Lento para usuarios experimentados:** Para usuarios experimentados, navegar paso por paso a través de menús puede resultar más lento en comparación con otros métodos de interacción más directos.

ESTILOS DE INTERFACE | INTERFAZ DE GRÁFICA DE USUARIOS - GUI

La Interfaz Gráfica de Usuarios (GUI - Graphical User Interface) es uno de los estilos de interfaz más utilizados y reconocibles en la actualidad. Se caracteriza por emplear elementos visuales, personalizable y de manipulación directa, como ventanas, iconos, botones y menús para interactuar con los usuarios.

Ventajas:

- **Facilidad de aprendizaje:** Las GUI son relativamente fáciles de aprender y utilizar, ya que utilizan representaciones visuales familiares para los usuarios.
- **Acceso a pantallas múltiples:** Los usuarios pueden interactuar con varias pantallas o ventanas al mismo tiempo, lo que facilita la multitarea y la organización de tareas.
- **Acceso rápido:** Los usuarios pueden acceder rápidamente a diferentes partes de la pantalla utilizando ventanas, pestañas y menús.

Desventajas:

- **Consumo de recursos:** Las GUI pueden requerir una cantidad significativa de recursos del sistema, como memoria y capacidad de procesamiento.
- **Distracciones visuales:** La rica interfaz visual de las GUI puede llevar a la distracción y la sobrecarga de información, especialmente si se presentan demasiados elementos en la pantalla.

ESTILOS DE INTERFACE | INTERFAZ DE RELLENO DE FORMULARIOS

La Interfaz de Relleno de Formularios es un estilo de interfaz que se enfoca en la introducción de datos en campos específicos de un formulario para recopilar datos estructurados y estandarizados. Este enfoque simplifica la tarea de ingresar información puntual, como nombres, direcciones, fechas y otros datos relevantes, al proporcionar campos designados para cada tipo de información.

Ventaja:

- **Facilidad de aprendizaje:** Los campos del formulario están claramente etiquetados, lo que ayuda a los usuarios a identificar y completar los datos requeridos de manera ordenada. Además, la mayoría de las personas están familiarizadas con la estructura de los formularios, así sea en papel.

Desventaja:

- **Espacio en pantalla:** Aunque es eficiente para la entrada de datos, esta interfaz puede ocupar mucho espacio en la pantalla, especialmente cuando se presentan varios campos en un solo formulario.

ESTILOS DE INTERFACE | INTERFAZ DE MANIPULACIÓN DIRECTA

A través de íconos y menús, estas interfaces permiten a los usuarios de un hardware específico interactuar especifico de manera directa con los elementos en la pantalla, utilizando dispositivos de entrada. Estas interfaces son comunes en los paneles de control de algunos electrodomésticos con pantallas u otros dispositivos.

Estas interfaces han evolucionado aportando interacción táctil con pantallas sensibles al tacto, lo que permite a los usuarios tocar y manipular directamente los elementos en la pantalla, facilitando así uso. Algunos cajeros automáticos son un ejemplo de adaptación a este tipo de interfaz. Las interfaces táctiles son cada vez son más comunes, habiendo llegado a dispositivos móviles o incluso a computadoras de escritorio.

ESTILOS DE INTERFACE | INTERFAZ DE RECONOCIMIENTO DE VOZ

Una interfaz de reconocimiento de voz es un sistema que permite a los usuarios interactuar con dispositivos electrónicos utilizando su voz como medio de entrada. Esta tecnología convierte las palabras habladas por los usuarios en texto o comandos que la máquina puede comprender y ejecutar. Ejemplos populares de interfaces de reconocimiento de voz incluyen a Siri, Google Assistant, Amazon Alexa, Cortana, entre otras.

ESTILOS DE INTERFACE | INTERFAZ INTELIGENTE

Las interfaces inteligentes, también conocidas como interfaces de inteligencia artificial (IA), son sistemas que utilizan algoritmos y técnicas de inteligencia artificial para interpretar y responder a las interacciones de los usuarios de una manera más sofisticada y adaptativa. Estas interfaces

no solo se limitan a responder a comandos específicos, sino que también pueden comprender el contexto y las intenciones del usuario para proporcionar respuestas más precisas y relevantes.

Estas interfaces pueden aprender y mejorar a medida que interactúan con los usuarios. Utilizan técnicas de aprendizaje automático para adaptarse a las preferencias y necesidades del usuario con el tiempo. Pueden comprender diferentes formas de preguntar o expresar lo mismo.

ESTILOS DE INTERFACE | INTERFAZ RESPONSIVE

Las interfaces responsive, o interfaces adaptativas, son diseñadas de manera que se ajusten y funcionen correctamente en una variedad de dispositivos y tamaños de pantalla. Esto es particularmente importante en la era actual, donde los usuarios acceden a sitios web y aplicaciones desde una amplia gama de dispositivos, como computadoras de escritorio, laptops, tablets y smartphones.

El diseño responsive es esencial para garantizar que los usuarios puedan acceder y utilizar una interfaz de manera efectiva, independientemente del dispositivo que utilicen. Al adaptarse automáticamente a diferentes pantallas, se mejora la experiencia del usuario y se reduce la necesidad de crear versiones separadas de una interfaz para cada dispositivo.

ESTILOS DE INTERFACE | INTERFAZ DE ACCESIBILIDAD

Las interfaces accesibles son aquellas que están diseñadas y desarrolladas de manera que puedan ser utilizadas por una amplia variedad de usuarios, incluyendo aquellos con discapacidades físicas, sensoriales o cognitivas. El objetivo principal de las interfaces accesibles es garantizar que todos los usuarios, independientemente de sus capacidades, puedan interactuar con la tecnología y obtener la misma información y funcionalidad.

El diseño de interfaces accesibles es crucial para asegurarse de que todos los usuarios tengan la oportunidad de utilizar y beneficiarse de las tecnologías, promoviendo la igualdad y la inclusión digital.

SOPORTE

El soporte al usuario se refiere a los servicios, recursos y asistencia proporcionados a los usuarios de un producto, servicio o sistema para ayudarles a utilizarlo de manera efectiva, resolver problemas, obtener respuestas a sus preguntas y tener una experiencia satisfactoria en general. El soporte al usuario es esencial para asegurarse de que los usuarios puedan aprovechar al máximo lo que están utilizando y superar cualquier obstáculo que puedan enfrentar durante su interacción.

Mensajes del sistema por acciones del usuario

Los sistemas deben comunicar claramente los resultados de las acciones del usuario. Los mensajes de confirmación, advertencia o error deben ser informativos y comprensibles, ayudando a los usuarios a entender qué ha sucedido y cómo proceder.

Ayudas en línea

Ofrecer ayudas contextuales en línea puede ser muy útil para los usuarios mientras navegan por la interfaz. Estas ayudas pueden ser en forma de información emergente, descripciones de herramientas (tooltips) o enlaces a páginas de ayuda detallada.

Documentación del sistema

Proporcionar documentación detallada sobre el sistema, su funcionalidad y cómo realizar diversas tareas es esencial para los usuarios que desean aprender más o resolver problemas por sí mismos. Esta documentación puede ser en forma de manuales, tutoriales en video o guías en línea.

PRUEBAS DEL SOFTWARE

Una prueba de software es un proceso sistemático y controlado que tiene como objetivo evaluar un programa de software, una aplicación o un sistema para identificar defectos, errores o fallos, y garantizar que cumpla con los requisitos y las expectativas establecidas. Las pruebas de software se realizan para verificar y validar que el software funcione según lo previsto y que sea confiable, seguro y de alta calidad.

¿Qué significa que el software ha fallado?

Que no hace lo que especifican los requerimientos

La fase de pruebas no es la única instancia en la que se identifican defectos en el proceso de desarrollo de software. Las revisiones de requerimientos y el diseño también juegan un papel crucial al descubrir problemas en etapas tempranas. Aunque las pruebas de software tienen como objetivo principal identificar fallos o defectos en el software, es decir, situaciones en las que el software no cumple con lo especificado en los requerimientos, resultando en un mal funcionamiento, es importante recordar que asegurar que un sistema funcione no garantiza la ausencia de defectos o errores en él

ASPECTOS

Finalidad

Una prueba tiene éxito cuando se descubre un error. La finalidad de una prueba es descubrir errores en el software, ya que los sistemas nunca son libres de estos. Las pruebas no garantizan la ausencia de defectos, ya que los problemas pueden surgir debido a cambios tecnológicos, de acceso u otros imprevistos. Se buscan casos de prueba que permitan encontrar errores y mejorar la calidad del software.

Objetivos

Los objetivos de las pruebas son identificar cualquier tipo de error mediante una estrategia definida. Se diseñan pruebas para exponer diversas clases de errores de manera eficiente y en el menor tiempo posible.

Beneficios

Los beneficios de las pruebas incluyen la detección temprana de errores antes de que el software salga del entorno de desarrollo. Esto ayuda a reducir los costos de corrección de errores durante la etapa de mantenimiento y evita la aparición de errores no descubiertos hasta después del lanzamiento.

PRINCIPIOS

Las pruebas de software se rigen por una serie de principios que guían su planificación, ejecución y evaluación. Estos principios son pautas fundamentales que buscan asegurar la calidad del proceso de prueba y la confiabilidad del software. Algunos de estos son:

- **Rastreabilidad de pruebas:** Todas las pruebas deben ser trazables hasta los requisitos del cliente. Esto implica establecer una clara relación entre codificación, diseño, especificación y elicitación de requisitos.
- **Planificación anticipada:** La planificación de pruebas debe realizarse con anticipación, incluso en etapas tempranas del desarrollo, como al trabajar con historias de usuario.
- **Principio de Pareto:** Aplicar el principio de Pareto, que establece que aproximadamente el 80% de los errores en un software provienen del 20% del código. Dado que ciertas partes del código generan la mayoría de los errores, es esencial prestar especial atención a estas áreas.
- **Enfoque bottom-up:** Iniciar las pruebas desde los componentes más pequeños y avanzar gradualmente hacia los más grandes, siguiendo un enfoque bottom-up. Esto involucra probar las unidades individuales primero.
- **Cobertura exhaustiva:** Asegurarse de que se prueben todas las condiciones a nivel de componente para garantizar una cobertura exhaustiva.
- **Pruebas independientes:** Preferiblemente, las pruebas deben ser realizadas por un equipo independiente, idealmente externo al equipo de desarrollo. Esto brinda una perspectiva más imparcial y cercana a la experiencia del usuario. En grupos de desarrollo más pequeños esto podría no ser posible y ellos mismos realizarían las pruebas.

EQUIPO INDEPENDIENTE DE PRUEBAS

Un equipo independiente de pruebas es un grupo de profesionales encargado de llevar a cabo las actividades de prueba en un proyecto de desarrollo de software de manera separada e imparcial del equipo de desarrollo principal. Este equipo tiene como objetivo evaluar objetivamente la calidad y el rendimiento del software, identificar defectos y asegurar que el producto cumpla con los requisitos y expectativas establecidos. Varios factores justifican un equipo independiente de pruebas, entre ellos:

Evitar conflictos

Un equipo independiente de pruebas evita el conflicto de intereses y la responsabilidad por los defectos encontrados. Al estar separado del equipo de desarrollo, puede detectar y reportar defectos de manera imparcial, sin involucrarse emocionalmente en el proceso de creación.

Pruebas concurrentes

Realizar pruebas de manera simultánea al proceso de codificación acelera la detección y corrección de errores. Esto permite que los defectos se identifiquen y se corrijan en un tiempo más corto, lo que mejora la calidad del software y reduce el tiempo de desarrollo.

Colaboración

La comunicación y retroalimentación entre el equipo de desarrollo y el equipo independiente de pruebas son esenciales. La detección temprana de defectos por parte del equipo de pruebas contribuye a evitar que los problemas se amplifiquen con el tiempo. La colaboración también garantiza que los problemas se comprendan correctamente y se aborden de manera efectiva.

ORIGEN DE ERRORES

La fase de pruebas busca identificar errores antes de que el software sea implementado en producción, permitiendo corregirlos y garantizar que el software cumpla con las expectativas y requerimientos establecidos. Los orígenes de estos podrían tener origen en distintos puntos:

- Especificaciones incorrectas o ambiguas.
- Requerimientos incompatibles con las estructuras y funcionalidades planificadas.
- Defectos en el diseño del sistema.
- Defectos en el diseño del programa.
- Errores en el código implementado.

TIPOS DE ERRORES

- Algorítmicos
- Sintácticos.
- Semánticos.
- De precisión o cálculos.
- De documentación
- De sobrecarga de procesamiento.
- De capacidad excedida de datos.
- De coordinación o sincronización

- De rendimiento
- De recuperación tras fallos o interrupciones
- De incompatibilidad o falla en comunicación hardware-software
- De incumplimiento de estándares

CLASIFICACIÓN OROGONAL DE ERRORES

Los defectos se organizan en categorías basadas en la naturaleza del problema.

Defecto por omisión:

- Ocurren cuando falta algún aspecto clave del código.
- Ejemplo: Variable no inicializada.

Defecto de cometido:

- Ocurren cuando algún aspecto es incorrecto.
- Ejemplo: Variable inicializada con un valor erróneo.

CLASIFICACIÓN DE DEFECTOS DE PFLEEGLER

Pfleeger clasifica los defectos ortogonales en diferentes tipos basados en distintos aspectos del software:

Función	Afecta la capacidad funcional que ofrecen las interfaces.
Interfaz	Afecta la interacción con otros componentes.
Comprobación	Afecta la lógica del programa.
Asignación	Afecta la estructura de datos.
Sincronización	Involucra sincronización de recursos compartidos y de tiempo real.
Construcción	Ocurre debido a problemas en repositorios, gestión de cambios o control de versiones.
Documentación:	Afecta las publicaciones y documentación de los desarrollos.
Algorítmico	Involucra la eficiencia o exactitud de un algoritmo.

CAJA BLANCA

La prueba de Caja Blanca (también conocida como prueba de Cristal o Abierta) se enfoca en derivar casos de prueba al seguir el flujo interno del programa y verificar detalles procedimentales. A través de esta técnica, los ingenieros de software pueden generar casos de prueba para asegurarse de que se recorren todos los caminos independientes dentro de cada módulo. El objetivo es examinar y probar el algoritmo a través de diferentes rutas, tomando decisiones variadas, atravesando bucles y analizando las estructuras internas de datos para confirmar su validez.

La prueba de Caja Blanca se basa en un examen exhaustivo de los detalles procedimentales. Se analizan los caminos lógicos del software al proponer casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles. Durante esta prueba, se observa el código fuente y su estructura, evaluando cómo fluye y cómo se ejecutan las diferentes partes del programa. En esencia, la prueba de Caja Blanca busca garantizar que el software funcione correctamente en función de su lógica interna y su implementación técnica.

Objetivos:

- Ejercitar todas las decisiones lógicas presentes en el código.
- Ejecutar todos los bucles en sus límites establecidos.
- Probar exhaustivamente las estructuras internas de datos utilizadas.

Para llevar a cabo estas pruebas:

- Se hace foco en los casos especiales, ya que tienden a ser propensos a errores debido a su naturaleza única o límites específicos.
- Se identifican errores tipográficos aleatorios que podrían pasar desapercibidos en el código.
- Se reconoce y se aborda la posibilidad de que el flujo de control intuitivo en el diseño pueda diferir del flujo real del programa, revelando discrepancias entre las expectativas y la realidad de la ejecución del software.

CAJA BLANCA | PRUEBA DEL CAMINO BÁSICO

La Prueba del Camino Básico propuesta por Tom McCabe, es una técnica que permite obtener pruebas de la complejidad lógica y usarla como guía para definir caminos de ejecución en un programa. Esto resulta en una métrica llamada complejidad ciclomática, que mide la complejidad de los ciclos de ejecución en el código.

CAJA BLANCA | PRUEBA DEL CAMINO BÁSICO | COMPLEJIDAD CICLOMÁTICA

La complejidad ciclomática es una métrica utilizada en la ingeniería de software para medir la complejidad lógica de un programa. Se calcula mediante el análisis del grafo de control del programa y se basa en el número de caminos independientes presentes en su estructura.

Un **camino independiente** se refiere a cualquier recorrido dentro del programa que introduce al menos un conjunto nuevo de sentencias de procesamiento o una nueva condición. En otras palabras, cada vez que se agrega una nueva rama de decisión o un conjunto de instrucciones, se considera un camino independiente. Estos caminos son rutas únicas y no redundantes dentro de un programa. Su consideración es fundamental para lograr una cobertura completa y efectiva en las pruebas de software.

Esta métrica establece un límite superior que ayuda a estimar la cantidad mínima de pruebas necesarias para garantizar una cobertura adecuada del programa, asegurando que cada sentencia se ejecute al menos una vez. En esencia, la complejidad ciclomática proporciona información sobre la cantidad de tomas de decisiones y bifurcaciones presentes en el código. Cuanto mayor sea la complejidad ciclomática, más complicado y potencialmente difícil de entender puede ser el programa. Esta métrica es útil para evaluar la facilidad de mantenimiento, comprensión y prueba de un software.

CAJA BLANCA | PRUEBA DEL CAMINO BÁSICO | COMPONENTES

Nodos

Los nodos son puntos en el código donde se realizan acciones. Pueden representar una o más sentencias procedimentales. Cada nodo representa una etapa en la ejecución del programa.

Aristas

Las aristas son conexiones que enlazan los nodos en el grafo. Representan el flujo de control del programa, indicando el camino que sigue la ejecución desde un nodo hasta otro.

Nodos Predicado

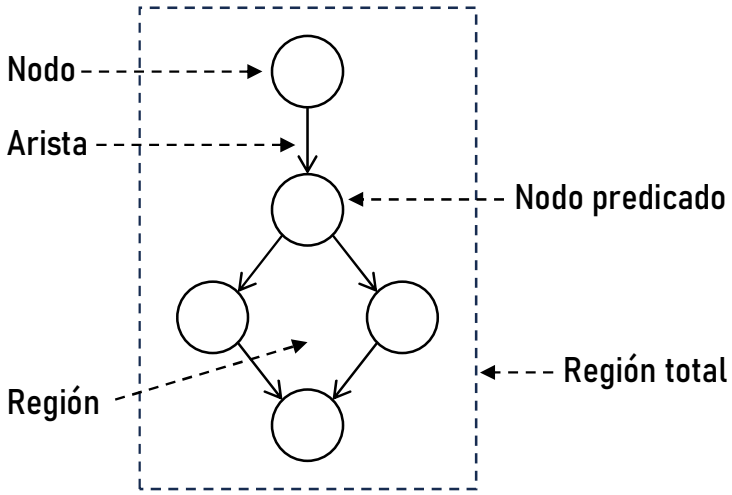
Los nodos predicados contienen condiciones (predicados) y tienen dos o más aristas salientes. Cada arista saliente corresponde a un posible resultado de la condición. Los nodos predicados se utilizan para representar bifurcaciones en el flujo de control, como estructuras if-else o bucles.

Región

Una región es un área dentro del grafo del programa que está delimitada por nodos y aristas. Representa una sección específica del flujo de control. Las regiones son útiles para analizar y calcular la complejidad del camino y asegurarse de que todas las rutas sean consideradas durante la prueba.

Región Total

La región total es el área exterior del grafo y puede estar abierta, lo que significa que no necesariamente todas las rutas conducen a un único punto de salida. Esta región engloba todo el flujo de control y puede incluir múltiples caminos de entrada y salida.



CAJA BLANCA | PRUEBA DEL CAMINO BÁSICO | CÁLCULO

El cálculo de la complejidad ciclomática se realiza utilizando tres fórmulas distintas. Es esencial verificar la precisión de la complejidad ciclomática utilizando todas estas fórmulas, ya que se espera que todas proporcionen el mismo resultado.

- **A:** Aristas.
- **N:** Nodos (Incluyendo Predicados)
- **P:** Nodos Predicados

Fórmulas:

1. V(g) = Región total + Regiones internas
2. V(g) = A - N + 2
3. V(g) = P + 1

Pasos a seguir

1. Encontrar Nodos. Línea a línea enumerar los nodos identificando los predicados (estructuras de control condicionales). Las secuencias se enumeran como un bloque y el corte de control (end) de predicados se enumera individualmente.

2. Dibujar el grafo de flujo correspondiente al programa. En base a los nodos identificados, realizar el grafo correspondiendo las aristas con el flujo de control.

3. Calcular la complejidad ciclomática utilizando las tres fórmulas para obtener el número de caminos independientes.

4. Determinar un conjunto básico de caminos independientes basado en los resultados anteriores. Recorrer el grafo y anotar la secuencia de nodos que forman cada uno de esos caminos.

5. Preparar los casos de prueba del código que forzarán la ejecución de cada camino del conjunto.

6. Ejecutar cada caso de prueba y comparar los resultados obtenidos con los esperados.

Este enfoque garantiza que el algoritmo sea probado a lo largo de todos sus caminos independientes.

Grafos asociados a condiciones lógicas compuestas:

A y B son los dos escenarios de un OR y ambos toman valores que pueden ser TRUE o FALSE, en cualquier combinación.

- Si A es TRUE, se pasa el control al nodo de escenario verdadero, sin pasar por B, ya es suficiente con un escenario verdadero para cumplir la condición.
- Si A es FALSE, se pasa el control a B y se evalúa su valor.
- Si B es TRUE, se pasa el control al nodo de escenario verdadero.
- Si B es FALSE, se pasa el control al nodo de escenario falso.

A y B son los dos escenarios de un AND y ambos toman valores que pueden ser TRUE o FALSE, en cualquier combinación.

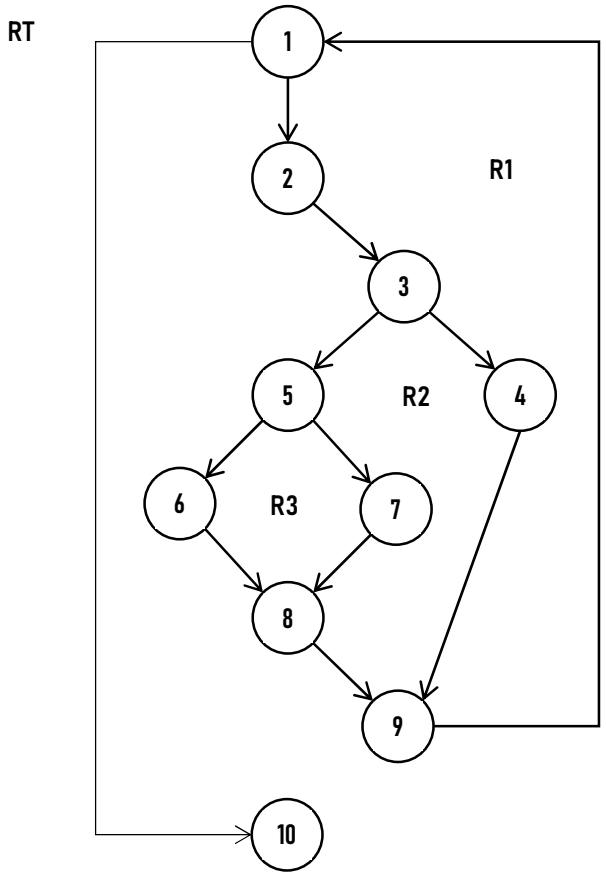
- Si A es TRUE, se pasa el control a B y se evalúa su valor, ya que ambos escenarios deben ser verdaderos para cumplir la condición.
- Si A es FALSE, se pasa el control al nodo de escenario falso.
- Si B es TRUE, se pasa el control al nodo Verdadero.
- Si B es FALSE, se pasa el control al nodo de escenario falso.

Ejemplo:

1. Encontrar nodos línea a línea:

```
Procedure Ordenar
        do while not eof() begin                    //1 – Nodo Predicado - Bifurcación
                leer registros;                     //2 - Nodo
                if (campo 1 de registro = 0) then   //3 – Nodo Predicado - Bifurcación
                        procesar registro           //4 - Nodo
                        Guardar en buffer;
                        Incrementar contador;
                else
                        if (campo 2 de registro = 0) then   //5 - Nodo Predicado - Bifurcación
                                reiniciar contador          //6 - Nodo
                        else
                                procesar registro;          //7 - Nodo
                                Guardar en archivo;
                        endif                               //8 - Nodo - Fin 5
                endif                                       //9 - Nodo - Fin 3
        end;                                                //10 - Nodo
end;
```

2. Dibujar el grafo:



3. Calcular la complejidad ciclomática:

   V(g) = Región total + Regiones internas
   V(g) = 1 + 3 = 4

   V(g) = A – N + 2
   V(g) = 12 - 10 + 2 = 4

   V(g) = P + 1
   V(g) = 3 + 1 = 4

   Complejidad ciclomática = 4

- Nodos: 10
- Nodos Predicados: 3
- Aristas: 12
- Regiones internas: 3

4. Conjunto básico de caminos independientes:

   1 - 10
   1 – 2 – 3 – 4 – 9 - 1 - 10
   1 – 2 – 3 – 5 – 7 – 8 – 9 - 1 - 10
   1 – 2 – 3 – 5 – 7 – 8 – 9 - 1 – 10

Los puntos 5 y 6 se hacen ejecutando código.

CAJA NEGRA

La prueba de Caja Negra (también conocida como prueba cerrada) se enfoca en los requisitos funcionales del software y se lleva a cabo sin tener en cuenta el funcionamiento interno del programa.

En las pruebas de Caja Negra, se examina la interfaz del software. Esto implica observar las entradas y salidas de datos, evaluando cómo responde el software a diferentes tipos de entradas. El tester no necesita conocer la lógica interna del código; solo se centra en la funcionalidad visible desde fuera. Concentra la prueba en el dominio de información y no en el desarrollo del algoritmo.

¿Qué errores busca?

- 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

CAJA NEGRA | PRUEBA DE PARTICIÓN EQUIVALENTE

Una prueba de equivalencia es una técnica de prueba de software que se utiliza para diseñar casos de prueba de manera eficiente. Para una condición de entrada se evalúan distintos escenarios contemplados en clases de equivalencia (que agrupan todos los posibles casos dentro de un conjunto particular) y se verifican si cumplen o no con esa condición, o si producen el resultado esperado. La idea es que, si un caso de prueba de una clase de equivalencia es válido y produce un resultado esperado, entonces se asume que otros casos dentro de la misma clase también lo harán.

Definición de resultados

- **Rango:** Cuando una condición de entrada establece un rango de valores. Se establecen tres clases de equivalencia, una válida que está dentro del rango y dos no válidas, una por debajo y otra por encima del rango.
- **Valor Específico:** Cuando una condición de entrada requiere un valor específico. Se crean tres clases de equivalencia, una válida con el valor específico y dos no válidas, una con un valor menor y otra con un valor mayor.
- **Conjunto:** Cuando una condición de entrada se relaciona con un conjunto de valores. Se establecen dos clases de equivalencia, una válida con elementos que pertenecen al conjunto y otra no válida con elementos que no pertenecen al conjunto.
- **Lógico:** Cuando una condición de entrada es lógica. Se definen dos clases de equivalencia, una válida que cumple con la condición lógica y otra no válida que no la cumple.

CAJA NEGRA | PRUEBA DE PARTICIÓN EQUIVALENTE | ANÁLISIS DE VALORES LÍMITE

El Análisis de Valores Límite (AVL) es una técnica de prueba que se complementa con la de partición equivalente. Se basa en la observación de que los errores son más propensos a ocurrir en los límites de los campos de entrada en lugar de en el centro. El AVL se centra en seleccionar casos de prueba en los extremos de las clases de equivalencia, en lugar de elegir cualquier elemento dentro de ellas. Algunos de los valores límites son: null, nill, puntero de vector, EOF, etc.

Esta técnica también se aplica a los campos de salida. En este contexto, se utilizan casos de prueba que generen valores de salida para verificar si el software responde adecuadamente. Los casos de prueba en los bordes de las clases se diseñan de la siguiente manera:

- Para una condición de entrada con un rango entre a y b, se prueban los valores a, b, valores menores que a y valores mayores que b.
- Para una salida con un rango entre a y b, se utilizan casos de prueba que generen valores de salida a y b.
- Se exploran los límites de las estructuras de datos internas.

ESTRATEGIAS DE PRUEBA

El enfoque estratégico de pruebas en el desarrollo de software es crucial para garantizar la calidad y la adecuación del producto. Proporciona una guía detallada sobre cómo planificar, llevar a cabo y evaluar las actividades de prueba a lo largo del ciclo de vida del software.

Proporciona

- Planificación o estrategia de las pruebas
- Diseño de los casos de prueba
- Ejecución de las pruebas
- Recolección y evaluación de los datos resultantes

Las estrategias de pruebas se integran en el ciclo de vida del software, asegurando que se realicen y planeen con anticipación pruebas de calidad en todas las etapas del proceso. Incluye tanto pruebas de bajo nivel, que examinan componentes individuales, como pruebas de alto nivel, que evalúan la funcionalidad general y la interacción del sistema. Un enfoque estratégico de pruebas abarca un conjunto de pasos que incorporan técnicas y métodos específicos para el diseño de casos de prueba.

Estas actividades de prueba son esenciales para la verificación y validación del software, formando parte del aseguramiento de la calidad en el desarrollo y garantizando que el producto final cumpla con los requisitos y expectativas del cliente.

CONCEPTOS

Verificación

Refiere a un conjunto de acciones destinadas a confirmar si el software se ha implementado de manera correcta y coherente con sus especificaciones. Durante la verificación, se llevan a cabo actividades para asegurar que el software cumpla con todos los requisitos establecidos, tanto los funcionales como los no funcionales. Se evalúa si el código, diseño y otros aspectos técnicos están en línea con la planificación y las expectativas.

Validación

Refiere a un conjunto de acciones destinadas que aseguran que el software construido se alinee de manera precisa con las necesidades del cliente y las metas del negocio. La validación implica un enfoque más amplio y abarcativo, que va más allá de las especificaciones técnicas. Se verifica si el software cumple con los objetivos definidos y si realmente satisface las expectativas y requisitos del cliente y los usuarios finales.

ESTRATEGIAS DE PRUEBA

Las estrategias de pruebas se adaptan a diferentes etapas del proceso de desarrollo y se orientan de lo particular a lo general.

ETAPA	NIVEL	CARACTERÍSTICAS
Código	**Pruebas de unidad**	Verifican el correcto funcionamiento de cada componente individual mediante la ejecución de varios casos de prueba. Estos componentes son módulos o partes del código.
Diseño y arquitectura de software	**Pruebas de integridad**	Verifican que los componentes trabajen correctamente en forma conjunta. Aunque los componentes que pasaron la prueba de unidad sean individualmente correctos, es posible que presenten fallos cuando colaboran entre sí
Sistema	**Prueba del sistema**	Verifica que cada elemento encaja de forma adecuada y que se alcanza la funcionalidad y el rendimiento del sistema total. Prueba el software como un todo, incluso en su interacción con el ambiente. Implica aspectos de ingeniería del sistema, requisitos de alto nivel e incluso interacción hardware-software.
Requisitos	**Pruebas de validación**	Proporcionan una seguridad final de que el software satisface todos los requisitos funcionales y no funcionales.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE UNIDAD

Las pruebas de unidad están diseñadas para evaluar la funcionalidad individual de los componentes del software. En este enfoque, se crean casos de prueba específicos para cada componente, los cuales incluyen datos de entrada y un conjunto de resultados esperados para esas entradas.

Durante las pruebas de unidad, se verifica la interfaz del módulo para asegurarse de que la información fluye correctamente. También se examinan las estructuras de datos locales utilizadas por el componente. Se ponen a prueba las condiciones límite para garantizar que el módulo funcione adecuadamente incluso en situaciones extremas. Esto implica analizar el código y explorar las condiciones presentes en las estructuras de control, intentando llevarlas a sus límites para asegurar su robustez. Además, se buscan recorrer todos los caminos independientes del componente para garantizar una cobertura exhaustiva.

Entre los errores más comunes detectados por las pruebas de unidad se encuentran:

- Cálculos incorrectos, como aplicación inadecuada de operaciones aritméticas, operaciones mezcladas, inicializaciones erróneas, falta de precisión o representación simbólica incorrecta.
- Comparaciones erróneas entre valores.
- Flujos de control inapropiados que pueden llevar a resultados incorrectos o a situaciones inesperadas.

Los casos de prueba en esta etapa deben estar diseñados para revelar errores como:

- Comparaciones entre diferentes tipos de datos.
- Comparaciones incorrectas entre variables.
- Uso incorrecto de operadores lógicos.
- Errores en las expectativas de igualdad y precisión.
- Finalización inadecuada o ausente de bucles.
- Fallos en la salida cuando se encuentra una iteración divergente.
- Variables de bucle modificadas inapropiadamente

ESTRATEGIAS DE PRUEBA | PRUEBAS DE UNIDAD | PROCEDIMIENTO

Dado que un componente no opera de manera independiente, en las pruebas de unidad se hace necesario desarrollar software adicional para controlar cada componente específico que se evalúa.

Un **controlador** desempeña el papel de un "programa principal". Su propósito radica en invocar y ejecutar el componente en evaluación. En este sentido, este programa recibe los datos del caso de prueba, los pasa al módulo bajo prueba como parámetros de entrada y, posteriormente, recolecta los resultados de salida para mostrarlos y/o guardarlos.

Paralelamente, cuando un componente en evaluación depende de otro componente para funcionar, el controlador crea un resguardo. Este resguardo actúa como un sustituto que emula el comportamiento del componente necesario. Esta práctica facilita el aislamiento del componente en prueba y garantiza un entorno controlado y predecible. Los resguardos suplantan a los módulos subordinados y cada resguardo se diseña para simular el comportamiento de un módulo subordinado en particular.

El uso de controladores y resguardos introduce una carga adicional de trabajo en el proceso de pruebas, ya que son elementos adicionales que no formarán parte del sistema final ni del producto. Si los controladores y resguardos son sencillos, el trabajo adicional es relativamente pequeño. Sin embargo, cuando los resguardos son más complicados, con una cohesión baja, y existen numerosos parámetros de entrada y salida, la cantidad de esfuerzo adicional requerida será mayor.

Es importante destacar que la prueba de unidad puede simplificarse si se diseña el módulo con un alto grado de cohesión, lo que significa que el módulo realiza una tarea específica y limitada, reduciendo la necesidad de controladores y resguardos complejos.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE UNIDAD | ORIENTADAS A OBJETOS

En el contexto de la programación orientada a objetos, la prueba de clase sería el equivalente a la tradicional prueba de unidad. Esta prueba está enfocada en evaluar el comportamiento y estado de una clase, así como todas las operaciones encapsuladas que esta contiene. Generalmente, el enfoque de la prueba de unidad se centra en una clase encapsulada, aunque los métodos individuales de esta son las unidades más pequeñas que pueden ser sometidas a pruebas.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE INTEGRACIÓN

Las pruebas de integración implican combinar los componentes que han pasado las pruebas de unidad según el diseño o estrategia establecido.

Durante esta etapa, pueden surgir diversos problemas resultantes de la combinación de módulos:

- **Pérdida de datos en interfaces:** Puede haber problemas en la transferencia de datos a través de interfaces. La salida de un módulo no necesariamente llega intacta al siguiente.
- **Efectos adversos e inadvertidos entre módulos:** Debido a una falla en la aplicación de buenas prácticas de diseño y codificación, los módulos con acoplamiento elevado y cohesión baja pueden tener un impacto negativo e involuntario en otros módulos.
- **Fallos en la secuencia de ejecución:** La combinación o encadenamiento de subfunciones y la transformación de los datos en una secuencia de ejecución de un conjunto de módulos puede no producir el resultado esperado.

Para abordar las pruebas de integración, se evalúan segmentos pequeños del programa o subconjuntos de componentes. Este aislamiento permite la corrección de errores, ya que los problemas pueden ser detectados y tratados de manera más eficiente en entornos controlados.

A modo de controlador se crea un conductor o módulo conductor que utiliza la interface del módulo contenedor de un segmento. Esto permite invocar y coordinar la ejecución de los módulos subordinados, facilitando la prueba de integración de manera controlada.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE INTEGRACIÓN | REGRESIÓN

Dado que las pruebas de integración se llevan a cabo de manera incremental, añadiendo componentes en pasos sucesivos, cada vez que se agrega un nuevo módulo, el software experimenta cambios. Estos cambios pueden dar lugar a nuevos caminos de ejecución, a nuevas entradas y salidas, o incluso a invocar nuevas lógicas de control. Estas alteraciones pueden introducir errores y causar problemas en funciones que antes trabajaban perfectamente.

Ante cada cambio realizado por una prueba de integración se aplica una prueba de regresión con el propósito de asegurarse de que las modificaciones no hayan propagado efectos secundarios no deseados. Estas pruebas también son ejecutadas durante el mantenimiento continuo del software.

Por ejemplo, en las pruebas de integración cuando se reemplaza un resguardo o un módulo conductor es común que aparezcan errores, y es aquí donde las pruebas de regresión desempeñan un papel crítico. Por eso, es esencial contar con un diseño de casos de prueba bien definido que registre y realice pruebas luego de estos cambios.

Las pruebas de regresión pueden ser realizadas manualmente, implicando la repetición de un subconjunto específico de todos los casos de prueba previos, o de manera automatizada utilizando herramientas específicas.

El conjunto de pruebas de regresión abarca tres categorías distintas de casos de prueba:

- **Componente:** Pruebas centradas en los componentes del software que han sido modificados.
- **Contexto:** Pruebas adicionales enfocadas en las funciones del software que probablemente se vean afectadas por los cambios.
- **Software:** Una muestra representativa que ejercite todas las funciones del software.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE INTEGRACIÓN | CRITICIDAD

En el proceso de pruebas de integración, es esencial identificar los módulos críticos, que pueden ser aquellos que cumplen con las siguientes condiciones:

- Módulos que abarcan o gestionan una gran cantidad de requerimientos de software.
- Módulos con un alto nivel de control que contienen varios módulos subordinados.
- Módulos complejos o propensos a errores, o que han experimentado problemas y están siendo atendidos en la etapa de mantenimiento.
- Módulos con requerimientos de rendimiento específicos, es decir, requisitos no funcionales asociados.

Estos módulos críticos deben someterse a pruebas de integración lo antes posible en el proceso. Además, es fundamental que las pruebas de regresión se concentren en estos módulos, dada su importancia y su potencial impacto en el sistema en su conjunto.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE INTEGRACIÓN | ESTRATEGIAS

En primera instancia, las pruebas de integración se llevan a cabo siguiendo alguna estrategia, que puede ser ascendente o descendente.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE INTEGRACIÓN | ESTRATEGIAS | DESCENDENTE

En esta estrategia, los módulos se integran comenzando desde el programa principal y descendiendo por la jerarquía de control. Se pueden adoptar dos enfoques distintos: el enfoque en profundidad y el enfoque en anchura. Para ambos enfoques, es esencial contar con resguardos.

En profundidad

Primero-en-profundidad integra todos los módulos de un camino de control principal de la estructura.

1. **Conductor**
   Se crea el conductor a partir del módulo principal en la jerarquía.

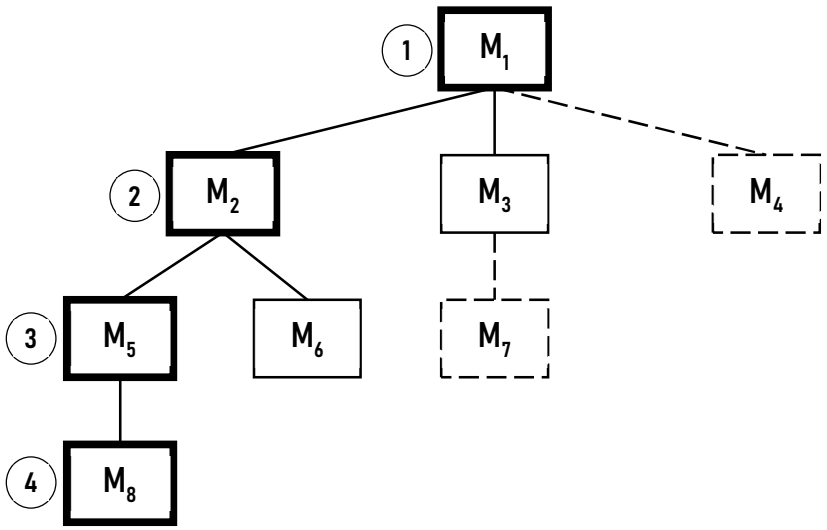
2. **Resguardos**
   Se crean resguardos de todos los módulos subordinados.

3. **Prueba de integración**
   Se realiza la prueba integración total con todos los resguardos de los módulos subordinados.

4. **Reemplazo**
   Una vez realizada la prueba, se procede a reemplazar el primer resguardo en la jerarquía (perteneciente al camino de control principal) por su módulo real.

5. **Prueba de regresión**
   Se realiza la prueba de integración con el módulo subordinado real para asegurarse de que el reemplazo no haya introducido errores y de que estos módulos trabajen correctamente en conjunto.

6. **Ciclos de pruebas de regresión**
   Siguiendo el camino de control principal de la estructura, se desciende en la jerarquía hacia el siguiente resguardo y se lo reemplaza por su módulo real. Se vuelve a realizar la correspondiente prueba de integración con la nueva incorporación. Este proceso se repite uno a uno para cada resguardo, hasta que todos hayan sido reemplazados por su módulo verdadero y probados en la integración completa.



En anchura

Primero-en-anchura incorpora todos los módulos directamente subordinados a cada nivel.

1. **Conductor**
   Se crea el conductor a partir del módulo principal en la jerarquía.

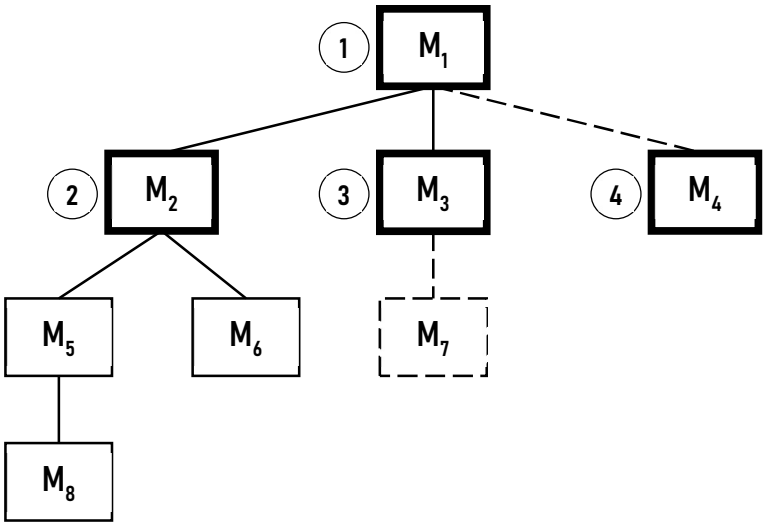
2. **Resguardos**
   Se crean resguardos de todos los módulos subordinados.

3. **Prueba de integración**
   Se realiza la prueba integración total con todos los resguardos de los módulos subordinados.

4. **Reemplazo**
   Una vez hecha la prueba, se procede a reemplazar el primer resguardo directamente vinculado al módulo principal con su módulo real correspondiente.

5. **Pruebas de regresión**
   Se realiza la prueba de integración con el módulo subordinado real para asegurarse de que el reemplazo no haya introducido errores y de que estos módulos trabajen correctamente en conjunto.

6. **Ciclos de pruebas de regresión**
   Se toma el siguiente resguardo directamente vinculado al módulo principal y se lo reemplaza por su módulo real. Se vuelve a realizar la correspondiente prueba de integración con la nueva incorporación. Este proceso se repite uno a uno para cada resguardo, hasta que todos hayan sido reemplazados por su módulo verdadero y probados en la integración completa.



PROFUNDIDAD	ANCHURA
Puede ser implementado cuando se tiene una rama o camino principal completo de una estructura de control.	Puede ser implementado cuando se tienen completos los submódulos más directos de un módulo principal.
La incorporación sucesiva de módulos ocurre verticalmente: uno a uno, descendiendo por la jerarquía, siguiendo los módulos de la rama de control principal.	La incorporación sucesiva de módulos ocurre horizontalmente: uno a uno, siguiendo los módulos directamente subordinados y por orden de invocación.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE INTEGRACIÓN | ESTRATEGIAS | ASCENDENTE

En la estrategia de pruebas ascendente, el proceso de prueba se inicia con los módulos atómicos, que son aquellos ubicados en los niveles más bajos de la estructura del programa y que no invocan a otros módulos. En esta fase, se prueban y validan individualmente los módulos más simples antes de proceder con la integración de componentes más complejos.

No se requiere el uso de resguardos para las componentes subordinadas, ya que antes de su integración cada componente fue probado de manera aislada. Sin embargo, los módulos conductores son fundamentales en esta estrategia, ya que brindan un marco para controlar la integración de conjuntos de módulos.

1. **Pruebas iniciales**
   Se someten a prueba todos los módulos atómicos que se encuentran en los niveles más bajos de la jerarquía, es decir, aquellos que no hacen llamados a otros módulos.

2. **Agrupación de módulos**
   Una vez superadas las pruebas iniciales, los módulos de bajo nivel se agrupan en segmentos o subconjuntos asociados a un módulo del nivel superior.

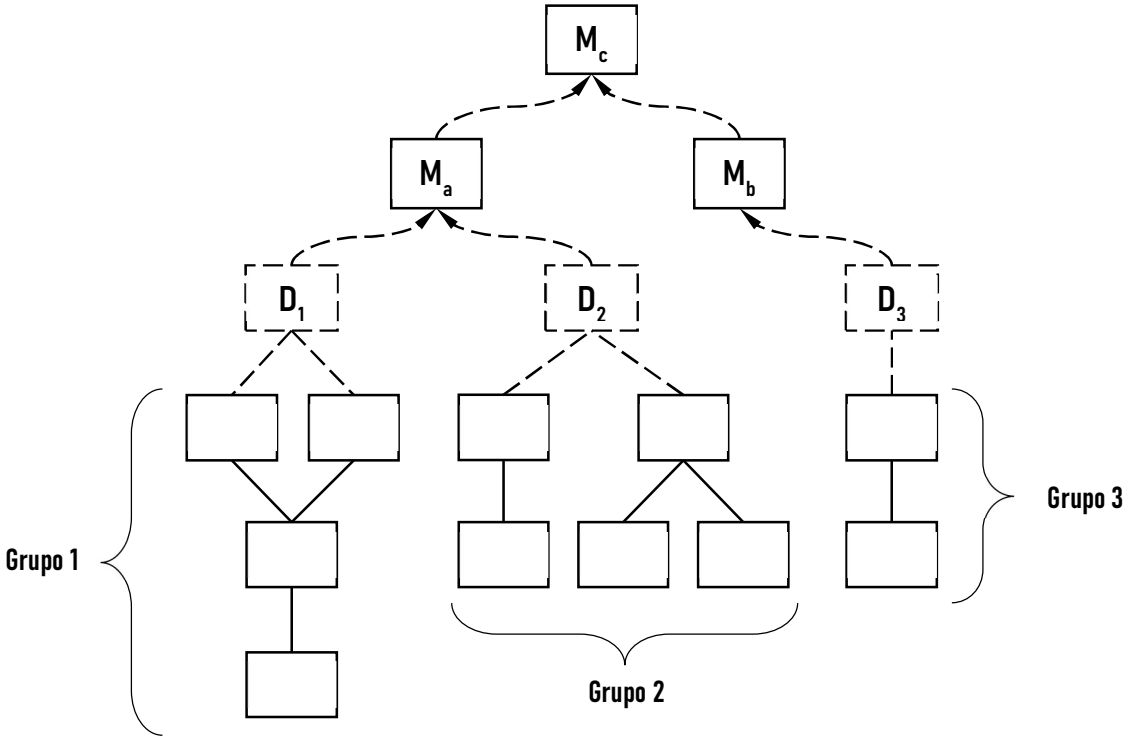
3. **Creación de conductores**
   Para cada uno de los grupos se crean conductores en base a la interface del módulo que contiene a ese grupo, coordinando así la entrada y salida de datos de prueba.

4. **Pruebas de integración**
   Se realizan las pruebas de integración de cada grupo de módulos a través de su correspondiente conductor. Esto prueba que los módulos colaboren adecuadamente.

5. **Eliminación de conductores**
   Luego de las pruebas de integración, se eliminan los conductores asociados a cada grupo de módulos.

6. **Prueba de regresión**
   Se realiza la prueba de integración al módulo contenedor del grupo, verificando que no se hayan introducido errores con los cambios y que los módulos trabajen correctamente en conjunto.

7. **Ciclos de pruebas de regresión:**
   El proceso se repite partiendo de los módulos integrados ya probados. Se vuelven a agrupar en módulos asociados, creando conductores y realizando las pruebas de integración a partir de estos. Nuevamente se eliminan los conductores y posteriormente se prueban los módulos integradores que correspondían a uno. Se avanza en la jerarquía a medida que se prueban módulos integradores de niveles más altos. El ciclo finaliza a partir de la creación del conductor que representa al programa principal.



ESTRATEGIAS DE PRUEBA | PRUEBAS DE INTEGRACIÓN | ESTRATEGIAS | SANDWICH

La principal desventaja del enfoque Descendente es la necesidad de crear resguardos para cada módulo asociado, lo que aumenta la carga de trabajo. La principal desventaja del enfoque Ascendente es que no se realiza una prueba del funcionamiento real o parcial del programa principal hasta que no se llega al último módulo conductor.

Una opción que combina ambos enfoques, conocida como "prueba sándwich", puede ser la más adecuada. En este enfoque se inicia desde los extremos hasta llegar al centro, realizando todas las pruebas de integración a partir de ambos caminos.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE INTEGRACIÓN ORIENTADAS A OBJETOS

El enfoque de prueba de integración en el software orientado a objetos no siempre es directamente aplicable debido a la falta de una estructura de control jerárquica clara, pero existen pruebas con comportamientos similares.

Pruebas basadas en hebras

Este enfoque equivale a la prueba de integración descendente. Se trata de la integración de un conjunto de clases que trabajan juntas para responder a una entrada o evento específico.

Pruebas basadas en uso

Este enfoque se asemeja a la prueba de integración ascendente. Comienza con las clases independientes y luego avanza a través de las clases dependientes.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE SISTEMA

La prueba del sistema está compuesta por una serie de pruebas distintas, cada una con un propósito específico. Aunque estas pruebas tienen objetivos diferentes, todas contribuyen a verificar la adecuada integración de todos los elementos del sistema y su capacidad para llevar a cabo las funciones correspondientes.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE SISTEMA | TIPOS

Pruebas de recuperación

Estas pruebas evalúan la capacidad del sistema para recuperarse de fallas y el tiempo que lleva retomar el procesamiento normal. Por lo general, se provocan fallos de manera controlada para verificar la efectividad de los mecanismos de recuperación.

Pruebas de seguridad

En estas pruebas, se examinan los mecanismos de protección y seguridad incorporados en el sistema para garantizar que los datos y la funcionalidad estén adecuadamente resguardados frente a amenazas y accesos no autorizados.

Pruebas de resistencia (Stress)

Las pruebas de resistencia se diseñan para someter al sistema a situaciones anormales o condiciones extremas. Se aplican cargas significativas para evaluar cómo responde el sistema bajo estrés y verificar si es capaz de mantener su funcionamiento estable.

Prueba de rendimiento

Este tipo de prueba se enfoca en evaluar el desempeño del sistema en tiempo real. En ocasiones, se combina con pruebas de resistencia para evaluar cómo el sistema se comporta en situaciones de alto estrés y carga. Se busca medir la velocidad, eficiencia y capacidad de respuesta del sistema en diversas condiciones.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE VALIDACIÓN

Al concluir con las pruebas de integración, se da paso a la fase de validación. Esta etapa implica una revisión exhaustiva de la configuración del software, asegurándose de que todos sus componentes se hayan desarrollado adecuadamente, estén debidamente catalogados y cuenten con el nivel de detalle necesario para respaldar la fase de soporte.

La validación del software se logra a través de una serie de pruebas que demuestran su conformidad con los requisitos establecidos. Al llevar a cabo cada caso de prueba de validación, pueden surgir dos condiciones:

- Las características de funcionamiento y rendimiento cumplen con las especificaciones y son consideradas aceptables.
- Se detecta una desviación respecto a las especificaciones y se procede a documentar las deficiencias identificadas para su posterior corrección.

ESTRATEGIAS DE PRUEBA | PRUEBAS DE VALIDACIÓN | PRUEBAS ACEPTACIÓN

Las realiza el usuario final en lugar del equipo 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 con casos de prueba concreto. Dentro de las pruebas de aceptación se pueden encontrar las pruebas ALFA y las pruebas BETA.

ALFA	BETA
Realizadas por el cliente.	Realizadas por los usuarios finales del software.
Ejecutadas en el lugar de desarrollo.	Llevadas a cabo en los lugares de trabajo de los clientes o en ambientes de producción. En ocasiones la versión beta se libera en el mercado para ser probada.
Entorno controlado: Se usa el software de forma natural, similar a como lo utilizaría un usuario en situaciones normales o reales, pero de manera controlada.	Entorno no controlado: la prueba beta es una aplicación en vivo del software, es decir, que este es presentado a los usuarios finales en un entorno real o de producción sin poder controlar su uso.
El desarrollador participa como observador.	El desarrollador normalmente no está presente.
El desarrollador es quien registrando errores y problemas de uso.	Los clientes registran los problemas encontrados durante la prueba beta y los informan regularmente al desarrollador.
Es posible añadir funciones y realizar ajustes en función tras las observaciones.	Se realizan ajustes según los comentarios de usuarios y clientes, y una vez no se necesiten más cambios en el software, se lanza la versión final.
No es la versión final del software ni la última etapa de prueba	Las pruebas beta constituyen la última fase de las pruebas y emplean técnicas de caja negra (pruebas de la interfaz).

PRUEBAS EN ENTORNOS PERSONALIZADOS

A medida que el software se vuelve más complejo, surge la necesidad de adoptar enfoques de pruebas especializados para garantizar la calidad del software en diversos aspectos.

PRUEBA DE ARQUITECTURA CLIENTE-SERVIDOR

La prueba de arquitectura cliente-servidor se enfoca en verificar el correcto funcionamiento y la interacción entre los componentes de una arquitectura cliente-servidor. Esta arquitectura involucra la comunicación y la colaboración entre clientes (interfaces de usuario) y servidores (que almacenan y procesan datos o ejecutan funciones).

Pruebas de Funcionalidad de la Aplicación

Se evalúa si las funciones y características de la aplicación en su conjunto operan de manera correcta. Esto implica la interacción entre los clientes y los servidores para asegurarse de que las operaciones se realicen según lo previsto.

Prueba de Servidor

Se verifica el correcto funcionamiento del servidor en términos de coordinación, manejo de datos y ejecución de funciones. Esta prueba se enfoca en las funciones que realiza el servidor de manera independiente de los clientes. Se evalúa el rendimiento del servidor, incluyendo el tiempo de respuesta y el procesamiento total de los datos. Esta fase incluye pruebas de rendimiento y pruebas de estrés para determinar cómo el servidor maneja diferentes cargas de trabajo y situaciones de alta demanda.

Prueba de Base de Datos

Se verifica la precisión y la integridad de los datos almacenados en la base de datos. Se examinan las transacciones, se asegura que los datos se almacenen, actualicen y recuperen correctamente, y se evalúa la consistencia de la información.

Pruebas de Transacciones

Se diseñan pruebas específicas para cada transacción con el objetivo de asegurar que se procesen de acuerdo a los requisitos. Se verifica la integridad y la precisión de los datos manipulados en cada transacción.

Pruebas de Comunicación de Red

Se verifica la comunicación entre los nodos de la red, incluyendo el paso de mensajes, las transacciones y el tráfico en la red. Se busca asegurar que no haya errores en la comunicación y que los datos se transmitan de manera adecuada.

PRUEBA DE SISTEMAS EN TIEMPO REAL

Las pruebas de sistemas en tiempo real se centran en verificar el funcionamiento correcto de sistemas que deben responder a eventos en tiempo real. Estos sistemas están diseñados para actuar y tomar decisiones en función del tiempo que transcurre entre la detección de un evento y la respuesta correspondiente. Se evalúan interrupciones, temporización de los datos, paralelismo entre las tareas, etc.

Pruebas de Tareas

Se realizan pruebas individuales de las tareas que componen el sistema de tiempo real. Estas pruebas buscan identificar errores lógicos y evaluar el funcionamiento sin tener en cuenta tiempo o el comportamiento.

Pruebas Inter-Tareas

Se prueban las tareas asincrónicas y se verifica la comunicación y coordinación entre estas tareas, especialmente cuando hay intercambio de datos o información.

Pruebas de Comportamiento

Se simula y examina el comportamiento del sistema de tiempo real en respuesta a eventos. Esto permite verificar si el sistema responde adecuadamente dentro de los límites de tiempo especificados.

Pruebas de Sistemas

Se prueba el sistema completo, que incluye tanto el software como el hardware integrados. Se evalúa cómo interactúan y funcionan en conjunto en un entorno de tiempo real.

PRUEBA INTERFACES GRÁFICAS

Las pruebas de interfaces gráficas se centran en evaluar la interacción y funcionalidad de las interfaces gráficas de usuario en un software. A continuación, se detallan los aspectos esenciales de estas pruebas:

Enfoque en la Interacción y Funcionalidad

El objetivo principal es verificar cómo los usuarios interactúan con las interfaces gráficas y si estas cumplen su propósito funcional. Se busca asegurar que las acciones del usuario generen las respuestas esperadas en la interfaz.

Verificación de la Interfaz

Se verifica que la interfaz gráfica esté correctamente diseñada y que los elementos visuales estén dispuestos de manera adecuada en la pantalla. Esto incluye aspectos como la alineación, el tamaño y el formato de los elementos.

Navegación entre Pantallas

Se evalúa la navegación entre diferentes pantallas o secciones de la aplicación. Se verifica que los enlaces, botones u otros elementos de navegación funcionen como se espera y lleven al usuario a la pantalla correspondiente.

Respuesta a Acciones del Usuario

Se comprueba cómo responde la interfaz gráfica ante las acciones realizadas por el usuario, como hacer clic en botones, introducir datos en campos o seleccionar opciones. Se busca que las respuestas sean coherentes y adecuadas.

PRUEBA DE LA DOCUMENTACIÓN Y FUNCIONES DE AYUDA

La prueba de documentación y funciones de ayuda es esencial para garantizar que el software esté acompañado de información precisa y útil que facilite su comprensión y uso por parte de los usuarios. Esta fase de prueba se centra en verificar la calidad y efectividad de la documentación asociada al programa. La prueba de documentación y funciones de ayuda suele dividirse en dos fases:

1. **Revisar e Inspeccionar**
   En esta fase, se examina la documentación de manera offline. Se evalúa la estructura, el estilo editorial, la organización de la información y la coherencia del contenido. El objetivo es asegurarse de que la documentación esté bien diseñada y sea comprensible.

2. **Prueba en Vivo**
   En esta fase, se interactúa con el programa real utilizando la documentación y las funciones de ayuda como guía. Se evalúa la efectividad de la documentación en la práctica, verificando si las instrucciones y explicaciones son útiles para realizar tareas específicas en el software.

DEPURACIÓN

La depuración de programas se refiere al proceso de identificar y corregir errores en programas informáticos que surgen durante las pruebas y a lo largo del ciclo de vida del software. La depuración se realiza después de encontrar un error, y el objetivo es solucionar ese error específico para que el software funcione como se espera.

El proceso de depuración puede tener dos resultados:

- Se identifica la causa del error, se corrige y se elimina.
- No se logra identificar la causa del error de inmediato. En este caso, el encargado en la depuración puede tener sospechas sobre la posible causa. A partir de esto se diseñan casos de prueba adicionales para confirmar las sospechas y, en caso de confirmación, se procederá a corregir el error. Este proceso puede ser iterativo.

DEPURACIÓN | ORIGEN DE LOS ERRORES

- Síntoma lejano de la causa. El error puede manifestarse en una parte del código distinta de su origen real.
- Síntoma desaparece temporalmente al corregir otro error, pero vuelve a aparecer eventualmente.
- Síntoma producido por error imprevisto.
- Síntoma causado por error humano ante un uso incorrecto
- Síntoma causado por problemas de tiempo o sincronización.
- Condiciones de entrada difíciles de reproducir, en alguna situación particular.
- Síntoma intermitente, especialmente en desarrollos hardware-software.
- El síntoma se debe a causas distribuidas entre varias tareas que se ejecutan en diferentes procesadores en sistemas concurrentes.

DEPURACIÓN | ENFOQUES DE LA DEPURACIÓN

- Diseñar programas de prueba adicionales que repliquen la falla original y ayuden a descubrir su fuente.
- Rastrear manualmente la ejecución del programa para simular su funcionamiento.
- Utilizar herramientas interactivas, como software de depuración integrado en el entorno de desarrollo (IDE).
- Después de corregir un error, es esencial reevaluar el sistema, realizar inspecciones y repetir las pruebas, incluyendo pruebas de regresión para verificar que las correcciones no hayan introducido nuevos errores.

GESTIÓN DE LA CONFIGURACIÓN DEL SOFTWARE

La Gestión de la Configuración del Software (GCS) es un conjunto de prácticas y procesos utilizados en ingeniería de software para gestionar y controlar los cambios realizados en el software durante su ciclo de vida. El objetivo principal de la GCS es garantizar que el software se desarrolle, mantenga y entregue de manera coherente, controlada y rastreable. Esto es esencial para mantener la integridad y la calidad del software a medida que evoluciona a lo largo del tiempo.

UTILIDAD

El GCS se usa para identificar y definir los elementos en el sistema, controlando el cambio de estos elementos a lo largo de su ciclo de vida, registrando y reportando el estado de los elementos y las solicitudes de cambio, y verificando que los elementos estén completos y que sean los correctos. En resumen, identifica el cambio, controla este, garantiza que el cambio se implemente adecuadamente e informa del cambio a todos aquellos que puedan estar afectados.

Es una actividad de autoprotección que se aplica durante el proceso del software.

IMPORTANCIA DEL GCS

La importancia de la Gestión de la Configuración del Software (GCS) radica en su capacidad para organizar, controlar y rastrear los cambios en el software a lo largo de su ciclo de vida, manteniendo la calidad y la integridad de este. Algunas de las razones clave por las cuales el GCS es esencial incluyen:

Eficiente Gestión de Cambios

El GCS ayuda a las organizaciones a manejar de manera eficiente la introducción de cambios en el software al documentarlos junto con las versiones existentes. Esto permite un seguimiento preciso de los cambios realizados, sus motivos y el impacto que tienen en el sistema.

Control de Cambios

El GCS controla la organización de los cambios antes y después de que el software sea distribuido al cliente. Esto garantiza que los cambios sean revisados y aprobados antes de ser implementados, lo que contribuye a evitar errores y garantiza la calidad del software entregado.

Responsabilidad y Priorización

El GCS asigna responsabilidades claras para aprobar y priorizar los cambios. Esto evita confusiones y asegura que los cambios sean evaluados adecuadamente antes de ser implementados.

Garantía de Implementación Adecuada

El GCS busca garantizar que los cambios se lleven a cabo de manera adecuada, siguiendo los procedimientos y estándares establecidos. Esto ayuda a evitar errores y conflictos en el software.

Comunicación de Cambios

El GCS proporciona mecanismos para comunicar y notificar a otros miembros del equipo sobre los cambios realizados. Esto es esencial para mantener a todos los involucrados informados y sincronizados.

Rastreabilidad

El GCS permite rastrear los cambios desde su origen hasta su implementación, lo que ayuda a comprender el motivo detrás de cada cambio y a mantener un historial claro de las modificaciones realizadas.

LÍNEA BASE

Una línea base es un punto de referencia fundamental en el desarrollo del software. Esta línea base se establece cuando uno o más Elementos de Configuración de Software (ECS) han sido completados y aprobados. Una vez que estos elementos se convierten en una línea base, cualquier modificación o cambio posterior en el documento se somete a la gestión y control del GCS.

En otras palabras, una línea base en GCS es una versión específica de un sistema o componente de software que se toma como referencia para desarrollos y cambios futuros. La línea base representa un estado estable y conocido del software en un punto determinado en el tiempo. Cualquier modificación realizada después de establecer una línea base se registra y rastrea meticulosamente. Esto permite comparar los cambios con la versión de la línea base y mantener

un historial claro de las alteraciones efectuadas. Esta práctica es esencial para asegurar que el software pueda evolucionar de manera controlada y que los cambios sean gestionados de manera efectiva y coherente a lo largo del tiempo.

ETAPAS DEL PROCESO DE GCS

1. Identificación de los elementos en la GCS

En esta etapa se identifican y definen los componentes del software que serán gestionados a lo largo del proyecto. Cada componente se describe con información.

- Nombre: cadena de caracteres sin ambigüedad
- Tipo de ECS (documento, código fuente, datos)
- Identificador del proyecto
- Información de la versión y/o cambio

2. Control de versiones

El control de versiones se basa en sistemas y herramientas específicas para rastrear y gestionar las modificaciones realizadas en el código fuente y otros elementos de configuración. Esto permite llevar un registro detallado de los cambios efectuados, quién los realizó, cuándo y por qué. Además, el control de versiones puede involucrar la creación de ramificaciones o bifurcaciones para trabajar en diferentes variantes del software. Combina procedimientos y herramientas para gestionar las versiones de los ECS que se crean a lo largo del proceso de software.

3. Control de cambios

En esta etapa, se establece un proceso formal para solicitar, evaluar, aprobar y realizar cambios en los elementos de configuración. Este proceso asegura que los cambios sean gestionados de manera controlada y que se comprenda el impacto de cada modificación en el software. A lo largo del proyecto los cambios son inevitables y el control es vital para el desarrollo del mismo. La autoridad de control de cambios (ACC) evalúa diversos factores antes de aprobar un cambio, como el impacto en el hardware, rendimiento, satisfacción del cliente, calidad y fiabilidad.

4. Auditoría de la configuración

Las auditorías y revisiones regulares se llevan a cabo para evaluar el cumplimiento de los procesos de GCS y asegurarse de que las prácticas establecidas se sigan adecuadamente. Esto incluye la revisión de aspectos como la identificación y el control de versiones, el control de cambios y la adhesión a los estándares y procedimientos establecidos. Las auditorías pueden realizarse tanto en forma de revisiones técnicas formales como a través de auditorías de configuración.

Actividades de la auditoría:

- Se verifica si el cambio realizado coincide con lo especificado en la Orden de Cambio y si se han incorporado modificaciones adicionales según corresponda.
- Se evalúa si se ha realizado una Revisión Técnica Formal (RTF) para analizar y validar la corrección técnica del cambio propuesto.
- Se verifica si se han seguido de manera adecuada los estándares de Ingeniería de Software (IS) en la implementación del cambio.
- Se comprueba si los cambios realizados se han registrado en los ECS incluyendo la fecha, el autor y otros atributos relevantes.
- Se evalúa si se han seguido los procedimientos establecidos en la GCS para señalar, registrar y comunicar el cambio.
- Se verifica si todos los ECS relacionados han sido actualizados de manera adecuada y coherente tras la realización del cambio.

5. Generación de informes de estado de la configuración

Esta etapa implica mantener actualizada la documentación relacionada con el software, incluyendo manuales de usuario, manuales técnicos, diagramas y cualquier otra documentación relevante. La generación de informes de estado de la configuración proporciona respuestas a preguntas clave sobre los cambios, como qué ocurrió, quién hizo el cambio, cuándo sucedió y cómo afectó a otros aspectos del software.

MANTENIMIENTO

El mantenimiento representa la última etapa en el ciclo de vida de un modelo de proceso de software después de haber sido entregado. También se conoce como la Evolución del Sistema. Durante esta etapa, se abordan diversas tareas, como solucionar errores, añadir mejoras y optimizar el sistema en respuesta a los cambios en los requisitos o el entorno.

El mantenimiento implica diversas acciones, que incluyen:

- Corregir errores detectados en el software.
- Realizar adaptaciones para satisfacer las necesidades del entorno.
- Añadir mejoras mediante la modificación de partes del software.
- Optimizar el software para lograr un funcionamiento más eficiente y eficaz.

Es común que se deba realizar mantenimiento en sistemas heredados, que pueden ser antiguos o carecer de metodología y documentación adecuadas, además de presentar una estructura poco modular. Estos sistemas heredados pueden ser desafiantes de mantener debido a su falta de claridad y organización.

BARRERA DE MANTENIMIENTO

Los costos relacionados con el mantenimiento pueden aumentar significativamente, ya que implica adaptaciones, mejoras y cambios diversos, además de la corrección de errores. En ocasiones, la complejidad del sistema puede aumentar a medida que se realizan cambios. Por esta razón, es esencial evaluar cuándo es apropiado cerrar el ciclo de vida de un sistema y considerar la posibilidad de reemplazarlo con uno nuevo.

La decisión de reemplazar un sistema existente con uno nuevo se toma considerando la estimación de los costos del ciclo de vida del proyecto actual en comparación con la estimación del nuevo proyecto. El punto en el que se decide abandonar el mantenimiento de un sistema existente y optar por uno nuevo se conoce como Barrera de Mantenimiento. Esta decisión se basa en un análisis cuidadoso de los costos y beneficios involucrados.

DESVENTAJAS

- La carga de trabajo de mantenimiento puede limitar la posibilidad de atender otros procesos de desarrollo.
- Tras un cambio pueden surgir efectos secundarios en el código, los datos y la documentación, lo que puede resultar en la generación de errores o en una disminución de la eficiencia.
- Las modificaciones realizadas durante el mantenimiento pueden dar lugar a una disminución en la calidad global del producto.
- Las tareas de mantenimiento a menudo requieren reiniciar las fases de análisis, diseño e implementación. No se trata simplemente de corregir un problema en un solo nivel, sino que el cambio debe propagarse.
- El mantenimiento puede representar entre el 40% y el 70% del costo total de desarrollo de un proyecto.
- Los errores en el software pueden provocar la insatisfacción del cliente.
- El trabajo de mantenimiento no siempre es atractivo, ya que no suele ser innovador ni creativo.
- En ocasiones, los cambios necesarios no fueron previstos en el diseño original.
- Puede ser complicado comprender el código de otros, especialmente si no cuenta con documentación adecuada o si carece de ella por completo.

TIPOS DE MANTENIMIENTO

1. **Mantenimiento perfectivo (60%):**
   - Realización de mejoras al sistema.
   - Adición de nuevas funcionalidades según las necesidades del cliente.
   - Optimización y aumento de la eficiencia del software.
   - Es el tipo de mantenimiento más comúnmente utilizado.
2. **Mantenimiento adaptativo (18%):**
   - Modificación del software para que se integre correctamente con su entorno.
   - Ajustes para que el software funcione en diferentes dominios, plataformas, hardware, o se adapte a cambios en el entorno.
   - Migración de datos y funcionalidades a nuevas versiones o entornos.
3. **Mantenimiento correctivo (17%):**
   - Diagnóstico y corrección de errores identificados en el software.
   - Solución de problemas de procesamiento, documentación, requisitos, rendimiento u otros aspectos.
4. **Mantenimiento preventivo (5%):**
   - Realización de modificaciones antes de que se presente una solicitud específica.
   - Orientado a facilitar futuros procesos de mantenimiento y evitar problemas.
   - Basado en el conocimiento acumulado sobre el producto y su funcionamiento.

ACTIVIDADES EN TORNO A UN CAMBIO

- Establecer el tipo de mantenimiento y gestionar el proceso de mantenimiento.
- Ingresar la solicitud del cambio, describiendo qué parte del sistema existente se quiere modificar.
- Analizar el impacto y alcance del cambio, evaluando cómo afectará a otras partes del sistema y a los usuarios.
- Generar una hoja de ruta (documentación) que detalle los pasos a seguir para realizar el cambio y rastrear su implementación.
- Comprender los cambios a realizar y determinar en qué partes del sistema influirán.
- Considerar la complejidad, modularidad, documentación y autodescripción del sistema al implementar el cambio.
- Realizar la implementación del cambio siguiendo la hoja de ruta.
- Evaluar la adaptabilidad del cambio a lo largo del sistema y verificar si se requieren ajustes adicionales.
- Considerar el efecto onda, es decir, cómo el cambio puede afectar otras partes del sistema o funcionalidades.
- Realizar pruebas exhaustivas en el software modificado para garantizar que funcione correctamente y no haya errores.
- Diseñar el cambio de manera que sea fácil de probar, verificar y que mantenga la integridad del sistema.
- Finalmente, el sistema con las modificaciones es liberado y puesto en uso.

CICLO DE MANTENIMIENTO

Para llevar a cabo los cambios de manera efectiva, es esencial utilizar un mecanismo que permita identificarlos, controlarlos, implementarlos e informarlos. El proceso de cambio se simplifica cuando en el desarrollo inicial se han considerado atributos de calidad como modularidad y documentación interna del código fuente y de soporte.

1. **Análisis**
   En esta fase se busca comprender el alcance y el efecto de la modificación necesaria. Se establecen principios generales, se crean planes temporales, se especifican controles de calidad, se identifican posibles mejoras y se estiman los recursos necesarios para el mantenimiento.

2. **Diseño**
   La etapa de diseño implica rediseñar el sistema para incorporar los cambios requeridos. Es importante que el diseño sea claro, modular y fácilmente modificable, además de utilizar notaciones estandarizadas.

3. **Diseño detallado**
   En esta fase se definen las notaciones para algoritmos y estructuras de datos, se especifican las interfaces, se manejan las excepciones y se consideran los efectos colaterales de la modificación.

4. **Implementación**
   Durante la implementación, se realiza la recodificación necesaria y se actualiza la documentación interna del código. Es fundamental seguir buenas prácticas de programación, como una correcta identación, comentarios de prólogo e internos, y una codificación clara y sencilla.

5. **Prueba**
   En esta etapa se llevan a cabo pruebas para revalidar el software después de los cambios. Se utilizan lotes de prueba y se comparan los resultados con los esperados.

6. **Actualización de la documentación de apoyo**
   Durante este paso, se debe revisar y actualizar cualquier documentación relacionada con el software que ha sido modificado. Esto puede incluir manuales de usuario, guías de instalación, documentación técnica, entre otros.

7. **Distribución e instalación de las nuevas versiones**
   Una vez que se han realizado los cambios y se ha asegurado su calidad, las nuevas versiones del software deben ser instaladas y distribuidas.

8. **Notificación de cambio**
   Es esencial comunicar de manera efectiva los cambios realizados en el software a los usuarios, clientes y otros involucrados. Esto puede incluir la elaboración de notas de versión, anuncios de cambios y cualquier otra forma de comunicación que informe sobre las modificaciones y cómo pueden afectar a los usuarios.

REJUVENECIMIENTO DE SOFTARE

El rejuvenecimiento del software es un desafío dentro del mantenimiento, que busca mejorar la calidad global de un sistema existente. Su objetivo es prevenir la degradación del sistema y evitar fallos relacionados con el envejecimiento, permitiendo aumentar su vida útil y continuar con el mantenimiento del software. Este enfoque retrospectivo considera los subproductos de un sistema para obtener información adicional o reformularlo de manera comprensible.

Re-Documentación

Consiste en analizar el código existente para generar documentación que revele su estructura, flujo y funcionamiento. Proporciona una comprensión más profunda del código.

Re-Estructuración

Implica modificar el software para hacerlo más comprensible y manejable. Se busca simplificar el código y reorganizarlo, a veces acompañado de una re-documentación.

Ingeniería Inversa

Este enfoque parte del código fuente sin documentación ni diseños y genera documentos de diseño y, en ocasiones, especificaciones y manuales de usuario. Se utiliza cuando no existe documentación para comprender el sistema.

Re-Ingeniería

La reingeniería es una extensión de la ingeniería inversa. Comienza con código desconocido y aplica la ingeniería inversa para obtener documentación, diseño y especificaciones adecuados. Luego, con esta información, se genera un nuevo código fuente mejor estructurado, mejorando la calidad sin alterar la funcionalidad.

AUDITORÍA

La auditoría es un análisis crítico que se lleva a cabo con el propósito de evaluar la eficiencia y eficacia de una sección u organismo, y para recomendar posibles cursos de acción que mejoren la organización y ayuden a alcanzar los objetivos establecidos. Este proceso no es simplemente mecánico; está dirigido por un auditor con juicio profesional, imparcialidad y objetividad.

La auditoría puede ser interna, externa o una combinación de ambas. La auditoría interna la realiza alguien dentro de la organización que está siendo evaluada, lo que puede afectar la objetividad, pero tiene un conocimiento profundo del lugar. La auditoría externa es realizada por un auditor externo, lo que brinda mayor objetividad y la capacidad de identificar aspectos que quizás no fueron detectados internamente. En muchos casos, se recomienda utilizar ambos enfoques.

DEFINICIÓN INFORMÁTICA

"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.

La auditoría en informática despliega una serie de funciones vitales. En primer lugar, contribuye a la prevención de delitos y problemas legales, generando estrategias para garantizar la seguridad y minimizar la vulnerabilidad del sistema ante ataques y errores. Además, verifica la adhesión a estándares de calidad y normativas específicas. El rol del auditor se limita a ofrecer sugerencias y recomendaciones, mientras que sus procedimientos varían según la filosofía y enfoque de cada entidad. La auditoría, en su carácter integral, evalúa minuciosamente todas las áreas del sistema bajo análisis, adoptando una perspectiva global y multidimensional.

La auditoría revisa y evalúa distintos aspectos:

- La organización involucrada en el procesamiento de información, incluyendo cada área.
- Los controles, sistemas y procedimientos informáticos, asegurándose de su solidez y cumplimiento.
- Los equipos de cómputo, abordando su eficiencia y seguridad, entre otros aspectos

OBJETIVOS

- Proteger los activos, incluyendo hardware, software y recursos humanos.
- Preservar la integridad de los datos, asegurando su consistencia, coherencia y ausencia de duplicados.
- Asegurar la efectividad de los sistemas, garantizando que cumplan con sus objetivos.
- Optimizar la eficiencia de los sistemas, utilizando la menor cantidad de recursos posible.
- Garantizar la seguridad y confidencialidad de la información.

INFLUENCIA DE LA AUDITORÍA EN INFORMÁTICA

Factores que pueden influir en la organización a través del control y la auditoría en informática:

- Control del uso adecuado de los recursos informáticos.
- Evaluación de pérdidas en capacidad de procesamiento de datos y sus causas.
- Mantenimiento de la privacidad individual y protección de datos.
- Prevención de la pérdida o mal uso de información.
- Apoyo en la toma de decisiones más acertadas.
- Resguardo de la privacidad de la organización.
- Impacto en otros aspectos relevantes.

CAMPO DE ACCIÓN

- Evaluación administrativa del área de informática.
- Evaluación de los sistemas y procedimientos, y de la eficiencia que se tiene en el uso de la información.
- Evaluación del proceso de datos, de los sistemas y de los equipos de cómputo (software, hardware, redes, bases de datos, comunicaciones).
- Seguridad y confidencialidad.
- Aspectos legales de los sistemas y de la información.