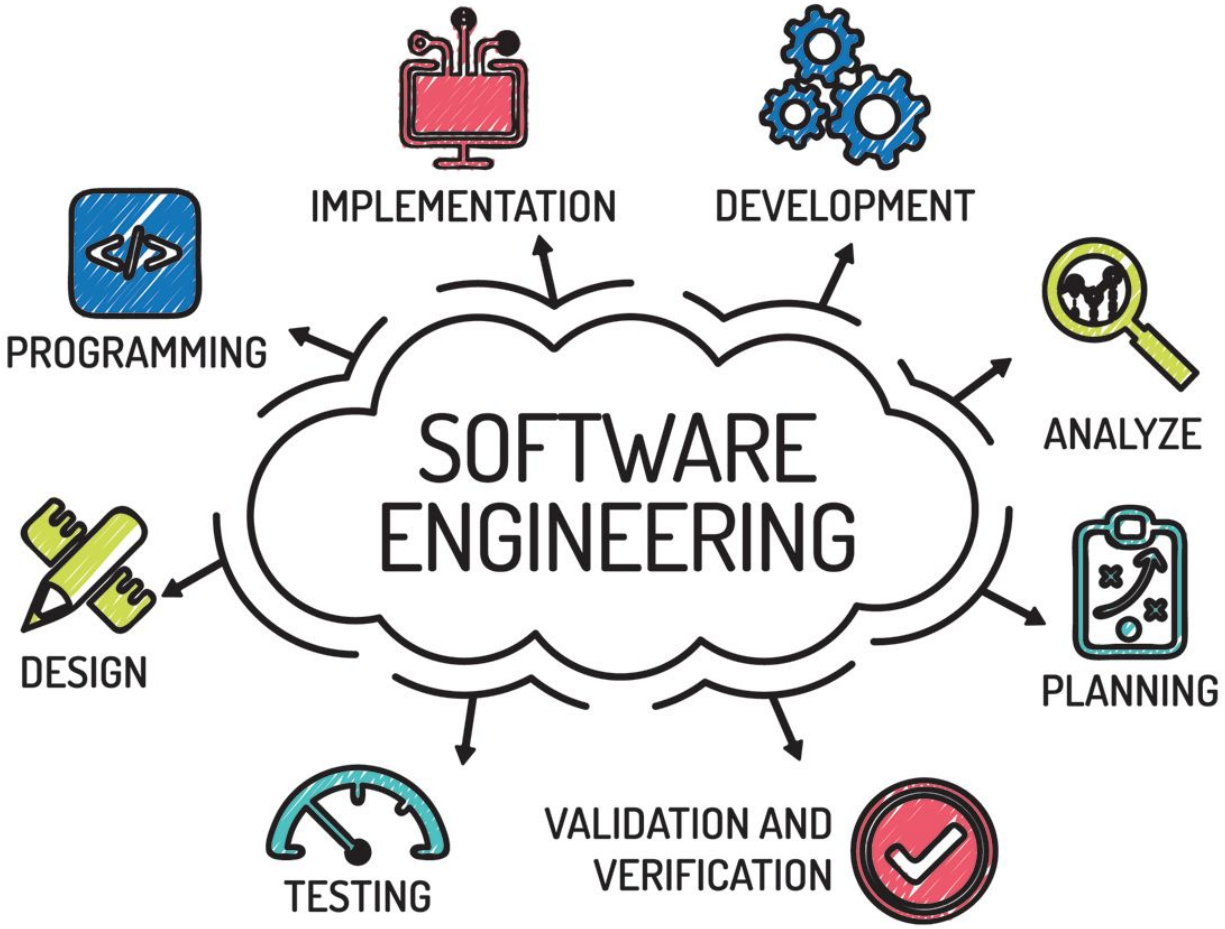


GAREIS, Leonel

Abril de 2019

INTRODUCCIÓN

¿Que es el Software?
El software se forma con:

(1) Las **instrucciones** (programas) que al ejecutarse proporcionan las características, funciones y el grado de desempeño deseados

(2) Las **estructuras de datos** que permiten que los programas manipulen información de forma adecuada

(3) Los **documentos** que describen la operación y el uso de los programas

¿Que es la Ingeniería de Software?
El IEEE ha elaborado una definición comprensible:
"La ingeniería de software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software."

Pressman caracteriza a la Ingeniería de Software como "una tecnología multicapa"

- Cualquier disciplina de ingeniería debe descansar sobre un esfuerzo de organización de **calidad**.
- El **proceso** define un marco de trabajo para un conjunto de áreas clave, las cuales forman la base del control de gestión de proyectos de software y establecen el contexto en el cual se aplican los métodos técnicos, se producen resultados de trabajo, se establecen hitos, etc.
- Los **métodos** indican cómo construir técnicamente el software.
- Las **herramientas** proporcionan un soporte automático o semiautomático para el proceso y los métodos, a estas se las conoce **herramientas CASE** (*Computer-Aided Software Engineering*)

Dado lo anterior, podemos decir que *el objetivo de la ingeniería de software es lograr productos de calidad, mediante un proceso apoyado por métodos y herramientas.*

VISIÓN GENERAL DEL PROCESO DEL SOFTWARE

Cuando se trabaja para construir un producto o sistema es importante seguir una serie de pasos predecibles, una especie de mapa de carreteras que ayude a crear un resultado de alta calidad y a tiempo. Este mapa se llama **proceso de software**.
Un proceso de software se define como ***"un marco de trabajo para las tareas que se requieren en la construcción de software de alta calidad"***.
Los procesos son importantes porque **ofrecen estabilidad, control y organización a una actividad** que puede volverse caótica si no se controla

Sin importar el tamaño del area de aplicacion , tamaño o complejidad del proyecto, el desarrollo de cualquier sistema se encontrará al menos en uno de los siguientes “procesos” genéricos:

- **Definición** (*análisis del sistema o software*)
- **Desarrollo** (*incluye diseño, codificación y prueba*)
- **Mantenimiento**

- **<u>Definición</u>**
  - En este proceso, determinamos **QUÉ** debe hacer el sistema
    - funcionalidad
    - información que va a manejar
    - necesidades de rendimiento
    - restricciones de diseño
    - interfaces del sistema con los usuarios y otros sistemas
    - criterios de validación
  - Desarrollar los documentos de requisitos del sistema (***SyRS***) y del software (***SRS***)

- **<u>Desarrollo</u>**
  - Se diseñan las **estructuras de datos** y los **programas**
  - Se escriben y documentan los programas
  - Por último, se prueba el software construido

- **<u>Mantenimiento</u>**
  - Este proceso comienza una vez construido el sistema
  - Se centra en el **cambio**

SISTEMAS DE INFORMACIÓN

¿QUE ES UN SISTEMA?
Según la RAE, un sistema es un conjunto de cosas que ordenadamente relacionadas entre sí, contribuyen a un determinado objetivo.
También es válido decir que es un modelo formado por una serie de elementos interrelacionados entre sí, que opera en un entorno cambiante y con unos determinados objetivos

A partir de esta definición podemos identificar los elementos que forman parte de un sistema:

- Los **componentes**
- Las **relaciones** entre ellos (que determinan la **estructura** del sistema)
- El **objetivo** del sistema

En todos los sistemas también podemos identificar otros elementos importantes para comprender cómo son y cómo funcionan:

- El **entorno**: aquello que lo rodea, dentro del cual está ubicado
- Los **límites**: la frontera entre lo que es el sistema y lo que constituye su entorno

INFORMACIÓN Y DATOS

Los datos están constituidos por los registros de los hechos, acontecimientos, transacciones, etc.
La información implica que los datos estén procesados y puestos en contexto, de tal manera que resulten útiles o significativos para el receptor de los mismos
Podemos decir entonces, que los datos son la materia prima, y la información es un producto.

¿QUE ES UN SISTEMA DE INFORMACIÓN?

Según Andreu, un sistema de información es :
"*Un conjunto formal de procesos que, operando sobre una colección de datos estructurada según las necesidades de la empresa, recopilan, elaboran y distribuyen la información (o parte de ella) necesaria para las operaciones de dicha empresa y para las actividades de dirección y control correspondientes (decisiones) para desempeñar su actividad de acuerdo a su estrategia de negocio*."

En pocas palabras:
Un **sistema de información** (**SI**) es un conjunto de elementos orientados al tratamiento y administración de datos e información, organizados y listos para su uso posterior, generados para cubrir una necesidad o un objetivo. Estos sistemas toman datos para luego transformarlos en información.
Para Le Moigne, los sistemas están formados por tres subsistemas interrelacionados:

- El **subsistema físico** transforma un flujo físico de entradas en un flujo físico de salidas.
- El **subsistema de decisión** procede a la regulación y control del sistema físico, decidiendo su comportamiento en función de los objetivos marcados.
- En interconexión entre el sistema físico y el sistema de decisión se encuentra el **subsistema de información**, encargado de almacenar y tratar la información relativa al sistema físico para ponerla a disposición del sistema de decisión.

INGENIERÍA DE REQUERIMIENTOS

¿Que es un Requisito?
En este caso, es una condición que debe cumplir un sistema para satisfacer un contrato, una norma o una especificación.

¿Como debe ser un requisito?

- **Necesario**: Su falta provoca deficiencia en el sistema a construir. No puede ser reemplazado
- **Conciso**: Es fácil de leer y entender, su redacción es simple

- **Completo**: No necesita ampliar detalles en su redacción. Proporciona la información suficiente
- **Consistente**: No es contradictorio con otro requerimiento
- **No ambiguo**: Tiene una sola interpretación. No debe causar confusiones su interpretación
- **Verificable**: Se puede verificar mediante un proceso que el software cumple con ese requisito. Si no lo puedo verificar... como se que se cumplió con él o no?

¿Y el Análisis de Requisitos?
Es un **proceso de estudio** de las necesidades de los usuarios para llegar a una definición de los requisitos del sistema, de hardware o de software.

¿Porque es tan importante la ingeniería de requisitos?
Porque nos va a permitir gestionar el proyecto de forma estructurada (ya que consiste en una serie de pasos bien definidos), nos va a permitir plantear cronogramas de proyectos confiables (estimar costos, recursos, tiempos), también nos va a ayudar a disminuir los costos y retrasos de proyectos (errores encontrados a tiempos ahorran costos y tiempo) y por supuesto a construir un software de mejor calidad

REQUISITOS FUNCIONALES Y NO FUNCIONALES

- **Requisitos Funcionales**:
    - describen la funcionalidad o los servicios que se espera que el sistema proveerá, sus entradas y salidas, excepciones, etc.
        - Por ejemplo:
        - “El sistema enviará un correo electrónico cuando se registre alguna de las siguientes transacciones: pedido de venta de cliente, despacho de mercancía al cliente, emisión de factura a cliente y registro de pago de cliente.”
        - “El sistema deberá ofrecer un explorador (browser) para que el usuario lea documentos en el almacén de documentos.”
- **Requisitos No Funcionales**:
    - se refieren a las propiedades emergentes del sistema como la fiabilidad, el tiempo de respuesta, la capacidad de almacenamiento, la capacidad de los dispositivos de entrada/salida, y la representación de datos que se utiliza en las interfaces del sistema.
        - Por ejemplo:
        - “Todas las comunicaciones externas entre servidores de datos, aplicación y cliente del sistema deben estar encriptadas utilizando el algoritmo RSA.”
        - “El sistema debe contar con un módulo de ayuda en línea.”

Actividades generales en el Análisis de Requisitos
- **Extracción de requisitos:** se descubren, revelan, articulan y comprenden los requisitos, usando técnicas de recogida de información. Se debe trabajar junto al

cliente para descubrir el problema a recibir, los servicios que debe prestar el sistema, las restricciones que debe presentar, etc.

- **Análisis de requisitos:** Se enfoca en descubrir problemas con los requerimientos hallados en la etapa anterior. Se leen los requerimientos, se conceptúan, investigan. Se resaltan los problemas
- **Especificación de requisitos:** se documentan los requerimientos acordados con el cliente. Se puede decir que la especificación es "pasar en limpio" el análisis realizado previamente, aplicando técnicas y estándares.
- **Validación de los requisitos:** es la actividad de la IR que permite demostrar que los requerimientos definidos en el sistema son los que realmente quiere el cliente. Además tenemos que garantizar que todos los requerimientos especificados sigan los estándares de calidad

TÉCNICAS DE RECOGIDA DE INFORMACIÓN

- **Entrevistas o Cuestionarios**: Se emplean para reunir información proveniente de personas o grupos. Las preguntas deben ser de alto nivel, para obtener información sobre aspectos globales del problema del usuario y soluciones potenciales. Debemos usar preguntas abiertas, para descubrir sentimientos, opiniones, experiencias o explorar un proceso o problema. Con este método podemos extraer una gran cantidad de requerimientos.
- **JAD (Joint Application Design)**: es el Desarrollo Conjunto de Aplicaciones, y consiste en un conjunto de reuniones entre los analistas y los usuarios.
  Se comienza con una documentación y se discuten los cambios o agregados. Al final del JAD se obtiene la DOCUMENTACIÓN DE REQUISITOS aprobada.
- **Prototipos**: Es una técnica usada para validar requerimientos hallados. Son simulaciones de un posible producto que nos permiten una importante retroalimentación en cuanto si los requerimientos que hemos identificado son los que le permiten al cliente realizar su trabajo eficazmente. Puede ser usado como un medio para explorar nuevos requerimientos
- **Tormenta de ideas:** Se usa para generar ideas. La intención es la de generar la máxima cantidad posible de requerimientos para el sistema. Los participantes deben pertenecer a muchas disciplinas y preferentemente deben tener mucha experiencia. Cuantas más ideas se sugieren, habrá mejor resultados. En un principio toda idea es válida y ninguna debe ser rechazada. Se debe hacer en un ambiente relajado.

Especificación de Requisitos de software (SRS)

Es un documento que contiene una descripción completa de las necesidades y funcionalidades del sistema que será desarrollado. Describe el alcance del sistema, las funciones que realizará, definiendo requisitos funcionales y no funcionales. Es el resultado final del análisis y validación (o evaluación) de requerimientos. Los clientes y usuarios usan la SRS para comparar si lo que se está proponiendo coincide con las necesidades de la empresa. Los analistas y programadores utilizan el SRS para determinar el producto que debe desarrollarse. Las pruebas se van a elaborar también en base a este documento. La SRS, al igual que los requerimientos, debe

ser: completa, precisa, consistente, no ambigua, verificable, concisa. Las características de la SRS deben ser validadas.

EL ESTUDIO DE FACTIBILIDAD

El estudio de la factibilidad de un sistema, tiene como objetivo el análisis de la necesidad que se tiene para proponer una solución a corto plazo, que tenga restricciones económicas, técnicas, legales y operativas. Un estudio de factibilidad debería ser relativamente barato y rápido. **El resultado debe informar la decisión respecto a si se continúa o no con un análisis más detallado**.

Se trata de recopilar suficiente información para que los directivos, a su vez, tengan información necesaria para decidir si se debe proceder a realizar un estudio de sistema. Los datos para los estudios de la viabilidad se pueden recopilar de entrevistas.

Se analizan 3 áreas de Viabilidad:

- **Viabilidad Técnicas:** el analista debe averiguar si es posible actualizar o incrementar los recursos técnicos actuales de manera que se puedan satisfacer los requerimientos considerados.
- **Viabilidad Económica:** se debe analizar el tiempo que el equipo necesita, los costos de realizar un estudio completo del sistema, costos de los empleados de la empresa, costos estimado del hardware y software
- **Viabilidad Operativa:** implica determinar si el sistema funcionara una vez instalado. Si los usuarios están conformes con el sistema actual y por lo cual no necesita uno nuevo. De lo contrario hay más posibilidad de que el nuevo sistema solicitado sea utilizado.

DISEÑO DEL SOFTWARE

¿Que es el DISEÑO?

Diseñar significa planear la forma y el método de una solución. Es el proceso que determina las características principales del sistema final, establece los límites en performance y calidad. Se caracteriza por un gran número de decisiones técnicas.

¿POR QUE HAY QUE DISEÑAR?

Porque en el diseño es donde se toman decisiones que afectarán finalmente al éxito de la implementación del programa, al igual que la facilidad de mantenimiento que tendrá el mismo (corrección de errores, evolución, etc.). El diseño es el proceso en el que se asienta la calidad del desarrollo del software. Sin diseño nos arriesgamos a un sistema inestable, que falle ante pequeños cambios, que pueda ser difícil de probar, de mantener.

DISEÑO ESTRUCTURADO

Es el es el proceso de definir **QUÉ** componentes existirán y la interconexión entre los mismos para solucionar un problema bien especificado.

El objetivo del diseño estructurado es la de desarrollar la estructura modular del programa y definir las relaciones entre módulos.
La ingeniería de diseño comienza cuando finaliza la de requisitos.
Su partida de inicio son los DFDs y las estrategias aplicadas son:

- Análisis de Transacción
- Análisis de Transformación

Se dispone de:

- Las **entradas** que suministran al sistema las entidades externas.
- Las **salidas** aportadas por el sistema a dichas entidades externas.
- Las **funciones descompuestas** que se han de realizar en ese sistema.
- El **esquema lógico de datos** del sistema.

¿Que se hace en el diseño estructurado?

- Determinar qué módulos implementarán los procesos terminales (primitivos) obtenidos en el análisis.
- Organizar la estructura de estos módulos y definir las conexiones entre los mismos.
- Describir el pseudocódigo para cada módulo.

Se basa en:

- **abstracción:** la abstracción es la selección de ciertos aspectos de un problema. El objetivo es suprimir aquellos aspectos que no son relevantes para el proposito que se este haciendo.
- **modularidad:** es la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.
- **encapsulamiento y ocultamiento de información:** el principio de ocultamiento sugiere que los modulos se caractericen por ocultar unos a otros. Es decir que los datos y procedimientos dentro de cada modulo solo pueda ser accedido desde otro modulo a traves de una interfaz establecida

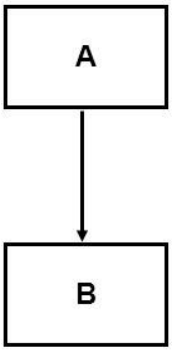
<u>Diseño arquitectónico del sistema</u>
El diseño arquitectónico representa la estructura de datos y los componentes del programa necesarios para construir un sistema computacional. Asume el estilo arquitectónico que tomara el sistema, la estructura y las propiedades de los componentes que constituyen el sistema y las interrelaciones entre todos los componentes arquitectónicos de un sistema.

¿Por qué es importante?
Nadie trataría de construir una casa sin un plano. Tampoco empezará a trazar planos bosquejando la distribución de la fontanería. Necesitaria un panorama general (la propia casa) antes de preocuparse por los detalles. Eso es lo que hace el diseño arquitectónico: proporciona una vista general y asegura que se obtenga lo que se desea.

MÉTRICAS DE CALIDAD ESTRUCTURAL
Mayor calidad estructural, indicaría un mantenimiento más fácil y, un aumento de extensibilidad y de reutilización.
Dos métricas utilizadas son:

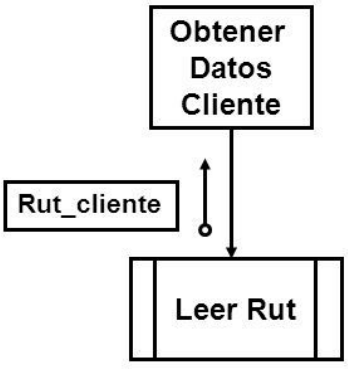
- **Acoplamiento**: es el grado de interdependencia entre módulos. Depende de la forma de interactuar entre los módulos
- **Cohesión**: es la medida de la relación funcional entre los elementos de un módulo, dicho de otra forma, es la medida cualitativa de cuán estrechamente están relacionados los elementos internos de un módulo.

NIVELES DE ACOPLAMIENTO



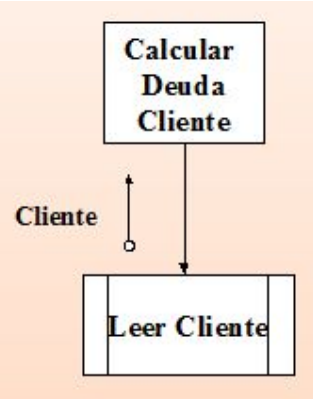
- **Acoplamiento normal:**
  - A y B normalmente acoplados si:
    - 1) A llama a B
    - 2) B retorna el control a A

"No se pasan parámetros entre ellos, solo existe la llamada del uno al otro"



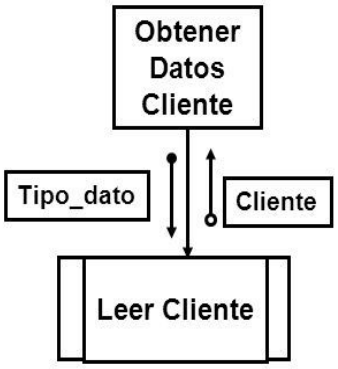
- **Acoplamiento de Datos:**
  - los módulos se comunican mediante parámetros, y cada parámetro constituye una unidad elemental de datos (todas las entradas y salidas se pasan como argumentos y todos los argumentos son elementos de datos y no de control)

"El módulo <<Leer Rut>> pasa al modulo <<Obtener Datos Cliente>> solamente los datos necesarios para el proceso, que en este caso es <<Rut_cliente>>"



- **Acoplamiento por Estampado:**
  - dos módulos hacen referencia a la misma estructura de Datos, siempre que esta no se una estructura de datos global

"*El módulo <<Leer Cliente>> pasa al módulo <<Calcular Deuda Cliente>> el registro completo de cliente, que está formado por: DNI, Nombre, Apellido, Deuda, Dirección, etc.*"



- **Acoplamiento de Control:**
    - dos módulos están acoplados por control cuando uno de ellos pasa al otro modulo elementos de control como argumentos .

"*El módulo <<Obtener Datos Cliente>> le pasa al módulo <<Leer Cliente>> un flag <<Tipo_dato>>, que indica que dato del cliente quiere leer*"

- **Acoplamiento común**:
    - Dos módulos o más comparten una estructura global de datos (entorno común).
      "*Una estructura de Datos, por ejemplo, datos del cliente, se define como global en el sistema y puede acceder a ella más de un módulo.*"

- **Acoplamiento de contenido**:
    - Ocurre cuando un módulo hace una referencia al interior de otro

NIVELES DE COHESIÓN

- **Funcional**: existe cohesion funcional cuando un modulo realiza una sola función. Todos los elementos que componen el modulo estan relacionados en el desarrollo de una unica funcion.
    - **Ejemplos** de módulos con cohesión funcional son los que realizan el cálculo del coseno de un ángulo, la función obtener un número aleatoriamente, escribir un registro en un archivo, borrar un registro, etc.

- **Secuencial:** existe cohesión secuencial cuando el módulo representa el empaquetamiento físico de varios módulos con cohesión funcional. Se usa cuando varios módulos con cohesión funcional trabajan sobre la misma estructura de datos.

En ella los datos de salida de un elemento sirven como datos de entrada al siguiente elemento de procesamiento.
    - **Por ejemplo** consideremos un módulo llamado <<formatear registro>>, que hace las actividades de leer el registro y formatearlo. En este caso se observa que la salida de la primera actividad, el registro, es la entrada para la segunda actividad, formatear.

- **Comunicacional:** un modulo con cohesion comunicacional es aquel cuyos elementos o actividades utilizan los mismos datos de entrada y salida. La mayor diferencia entre estas la secuencial y la comunicacional, es que las actividades de la **secuencial** se realizan en un orden especifico, mientras que la **comunicacional** el orden en que se ejecutan sus actividades no importa.
    - **Por ejemplo**, consideremos un módulo, llamado <<leer registro cliente>>, que realiza las actividades de leer el nombre y leer la dirección del cliente. En este caso se observa que las diferentes actividades utilizan los mismos datos de entrada y de salida, que serán los de cliente. Además es indiferente el orden de ejecución de las diferentes actividades.

- **Procedural:** Este tipo de cohesion se da cuando el modulo tiene una serie de elementos (funciones) relacionados por un procedimiento efecutado por el codigo. Los modulos tienden a estar compuestos por funciones que tienen poca relacion unas con otras.
    - Un ejemplo seria un módulo que realiza las siguientes actividades: limpiar cubiertos, preparar pollo, llamar por teléfono, picar verdura, poner la mesa. Estas actividades no tienen ninguna relación entre sí, y lo único que ocurre en el módulo es que el control de ejecución se va pasando de una actividad a la siguiente. Estas actividades constituyen acciones que una persona puede hacer un cierto dia, pero que es muy difícil que vuelva a repetirlas y, además, en el mismo orden. Por esto, la reutilización de este módulo sería muy difícil.

- **Temporal:** es aquel cuyos elementos están implicados en actividades que están relacionadas en el tiempo. Uno de los ejemplos más claros de este tipo de cohesión son los módulos de iniciación y finalización, y todos aquellos que representan unas acciones que deban ejecutarse en un momento determinado del tiempo sin que sean necesarios parámetros de control.
    - **Por ejemplo**, un módulo que realice las siguientes actividades: sacar la basura, apagar la televisión y limpiarse los dientes. Se observa que estas actividades están relacionadas en el tiempo (se ejecutan antes de que una persona se acueste) y su orden de ejecución es indiferente.

- **Lógica:** Los elementos de un módulos están lógicamente asociados si pueden pensarse en ellos como pertenecientes a la misma clase lógica de funciones. Ejemplo: Se pueden combinar en un modulo simple todos los elementos de procesamiento que caen en la clase “entradas”, que abarca todas las operaciones de entrada. Un modulo lógicamente cohesivo no realiza una función específica, sino que realiza una serie de funciones

    - **Un ejemplo**, podría ser un modelo que en función del valor del flag de entrada que reciba, realiza diferentes actividades que están relacionadas de forma lógica. Un caso es un módulo que realiza las actividades correspondientes a un viaje teniendo en cuenta el medio de locomoción (barco, avión, etc.).

- **Coincidental o casual:** se dice que en un modulo existe cohesion coincidemental cuando entre los elementos que lo componen no existe ninguna relacion con sentido. Las actividades en un modulo asi, no estan relacionadas, ni mediante flujos de datos ni de control. Este tipo de cohesion es muy perjudicial ya que el objetivo de la modularizacion es crear modulos con sentido y no simplemente romper en trozos el codigo.

CALIDAD DEL SOFTWARE

¿Que es la gestión de la calidad de un software?

La gestión de la calidad de un software es una actividad protectora que incorpora tanto aseguramiento como control de la calidad. Se aplica a cada paso de la ingeniería del software.
Cuando hablamos hablamos de calidad, hablamos de dos tipos: *calidad del diseño* y *calidad de concordancia*.

- La **calidad del diseño** se refiere a las características que los diseñadores especifican para un elemento. Esta incluye requisitos, especificaciones y diseño del sistema.
- La **calidad de concordancia** es el grado en el que las especificaciones de diseño se aplican durante la implementación. Si esta sigue el diseño y el sistema resultante satisface los requisitos y metas de desempeño, la calidad de concordancia es alta.

Robert Glass argumenta que es conveniente una relación más “intuitiva”.

satisfacción del usuario = producto manejable + buena calidad + entrega dentro del presupuesto y tiempo

¿Qué tareas realiza la SQA?

Abarca procedimientos para la aplicación eficaz de métodos y herramientas, revisiones técnicas formales, estrategias, métodos de prueba, procedimientos para garantizar la concordancia con los estándares y mecanismos de medición y reporte

Factores de calidad del software

Sirven para descomponer el concepto genérico de “calidad” en otros más sencillos
La división es subjetiva pero se la puede agrupar en 3 perspectivas: operativa, de mantenimiento y evolutiva
Operativa:

- Corrección: el software cumple con las especificaciones
- Fiabilidad: grado de confianza del software
- Eficiencia: aprovechamiento de recursos de software

- Seguridad: grado en el que puede controlarse el acceso a sus datos
- Facilidad de uso: grado de facilidad para los usuarios en su uso

Mantenimiento:
- Flexibilidad: esfuerzo necesario para modificar un programa
- Facilidad de prueba: para realizar pruebas del sistema
- Facilidad de mantenimiento: grado de esfuerzo para localizar y reparar errores

Evolutivos:
- Portabilidad: facilidad de migrar el software de un entorno a otro
- Capacidad de reutilización: grado en que un programa o parte de él puede usarse para otro
- Capacidad de interoperación: esfuerzo necesario para que un software opere conjuntamente con otros sistemas

---

PRUEBAS DEL SOFTWARE

El objetivo de la prueba del software es descubrir errores; se cumple planeando y ejecutando una serie de pasos (pruebas de unidad, integración, validación y sistema)
Un buen caso de prueba es aquel que tiene una alta probabilidad de descubrir un error no encontrado hasta entonces.
La actividad llevada a cabo es la de **V&V** (Validacion y Verificacion), donde se prueba el código, la documentación y la ayuda del software.

Según **Myers**, ante todo, el objetivo de las pruebas es la detección de defectos en el software y que descubrir un defecto debería considerarse el éxito de una prueba.
Sus recomendaciones para las pruebas son:
- Cada caso de prueba debe definir el resultado de salida esperado.
- El programador debe evitar probar sus propios programas
- Se debe inspeccionar a conciencia el resultado de cada prueba
- Al generar casos de prueba, se deben incluir tantos datos de entrada válidos y esperados como no válidos e inesperados
- Las pruebas deben centrarse en dos objetivos
    - Probar si el software no hace lo que debe hacer
    - Probar si el software hace lo que debe hacer
- Se deben evitar los casos desechables, es decir, los no documentados ni diseñados cuidadosamente, ya que suele ser necesario probar una y otra vez el software hasta que esta libre de defectos
- NO deben hacerse planes de prueba suponiendo que no existen errores, y por lo tanto asignando pocos recursos.

EL PROCESO DE PRUEBA

Comienza con la generación de un plan de pruebas en base a la documentación sobre el proyecto y la documentación sobre el software a probar. A partir de dicho plan, se comienzan a diseñar en detalle las pruebas específicas basándose en la documentación del software a probar.
Una vez detalladas las pruebas, se toma la configuración del software que se va a probar para ejecutar sobre ella los casos.
A partir de los resultados de salida, se pasa a su evaluación mediante comparación con la salida esperada. A partir de esta, se pueden realizar dos actividades

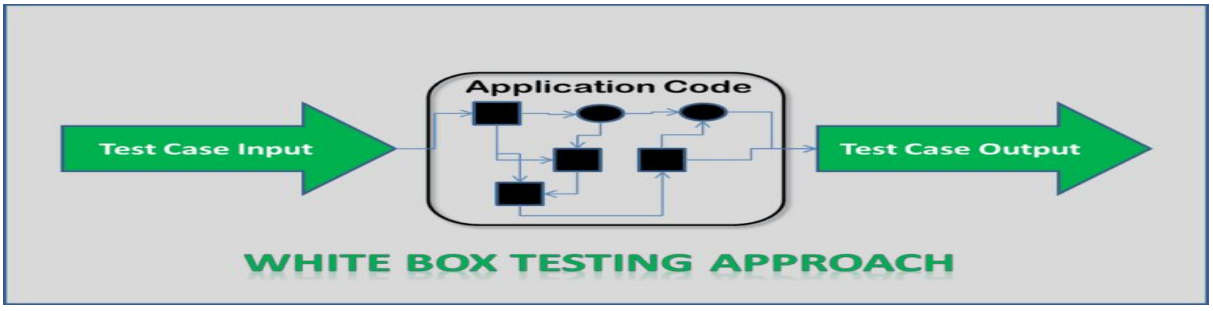
- La depuración (localizacion y correccion de defectos)
- El análisis de la estadística de errores.

La depuración puede corregir o no los defectos. Si no consigue localizarlos, puede ser necesario realizar pruebas adicionales para obtener más información. Si se corrige un defecto, se debe volver a probar el software para comprobar que el problema esté resuelto.

Por su parte, el análisis de errores puede servir para realizar predicciones de la fiabilidad del software y para detectar las causas más habituales de error y mejorar los procesos de desarrollo.

MÉTODOS DE DISEÑO DE CASOS DE PRUEBA
Existen distintos enfoque para diseñar casos de prueba

Las Pruebas de caja blanca (White-Box Testing).



Consiste en centrarse en la estructura interna (implementación) del programa para elegir los casos de prueba. En este caso, la prueba ideal (exhaustiva) del software consistiría en probar todos los posibles caminos de ejecución.
Conociendo el código y siguiendo su estructura lógica, se pueden diseñar pruebas destinadas a comprobar que el código hace correctamente lo que el diseño de bajo nivel indica y otras que demuestren que no se comporta adecuadamente ante determinadas situaciones. **Ejemplos típicos de ello son las pruebas unitarias**.

El diseño de casos debe basarse en la elección de caminos importantes que ofrezcan una seguridad aceptable en descubrir defecto, y para ello se utilizan los llamados **criterios de cobertura lógica**.
Una posible clasificación de **criterios de cobertura lógica**, es la que se ofrece más abajo.

1. **Cobertura de Sentencias**. Se trata de generar los casos de prueba necesarios para que cada sentencia o instrucción del programa se ejecute al menos una vez.
2. **Cobertura de Decisiones.** Consiste en escribir casos suficientes para que cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, uno falso.
3. **Cobertura de Condiciones.** Se trata de diseñar tantos casos como sea necesario para que cada condición de cada decisión adopte el valor verdadero al menos una vez y el valor falso al menos una vez.

Las pruebas de caja negra (Black-Box Testing)

Son pruebas funcionales. Se parte de los requisitos funcionales, a muy alto nivel, para diseñar pruebas que se aplican sobre el sistema sin necesidad de conocer cómo está construido por dentro (Caja negra). Las pruebas se aplican sobre el sistema empleando un determinado conjunto de datos de entrada y observando las salidas que se producen para determinar si la función se está desempeñando correctamente por el sistema bajo prueba. Las herramientas básicas son observar la funcionalidad y contrastar con la especificación.

Estrategias de prueba del software

¿En qué consiste una estrategia de prueba de un software?
La prueba es un conjunto de actividades que se planean con anticipación y se realizan de manera sistemática.
Cualquier estrategia de prueba debe incorporar la planeación de pruebas, el diseño de caso de pruebas, la ejecución de las pruebas y la recolección y evaluación de los datos resultantes.

- **Verificación y Validación**
  - **Verificación** es el conjunto de actividades que aseguran que el software implemente correctamente una función específica.
  - **Validación** es un conjunto diferente de actividades que aseguran que el software construido corresponde con los requisitos del cliente

Niveles de prueba:

1. **Prueba de unidad:** es la prueba de cada módulo, que normalmente realiza el propio personal de desarrollo en su entorno
2. **Prueba de integración:** con el esquema del diseño del software, los módulos probados se integran para comprobar sus interfaces en el trabajo.
3. **Prueba de validación:** el software totalmente ensamblado se prueba como un todo para comprobar si cumple los requisitos funcionales y de rendimiento, facilidad de mantenimiento, recuperación de errores, etc. Coincide con la de sistema cuando el sw. no forma parte de un sistema mayor.
4. **Prueba del sistema:** el sw. ya validado se integra con el resto del sistema (rendimiento, seguridad, recuperación y resistencia)
5. **Prueba de aceptación:** el usuario comprueba en su propio entorno de explotación si lo acepta como está o no

¿Quien prueba?
El desarrollador de software siempre será el responsable de probar las unidades individuales del programa y asegurar que cada una realice su función para la cual se la diseñó. En muchos casos el desarrollador también aplica la prueba de integración. Solo después de que la arquitectura de software está completa participará un grupo independiente de prueba. El papel de este grupo consiste en eliminar los problemas propios de dejar que el constructor pruebe lo que él mismo ha construido; es decir, elimina el conflicto de intereses que pudiera haber

TIPOS DE PRUEBAS DE SOFTWARE

Pasaremos a detallar algunas de las pruebas más utilizadas:

- **Prueba Unitaria**
    - Objetivo: Se focaliza en ejecutar cada módulo (o unidad mínima a ser probada, ej = una clase) lo que provee un mejor modo de manejar la integración de las unidades en componentes mayores.
- **Prueba de Regresión**
    - Objetivo: Determinar si los cambios recientes en una parte de la aplicación tienen efecto adverso en otras partes. En esta prueba se vuelve a probar el sistema a la luz de los cambios realizados durante el debugging, mantenimiento o desarrollo de la nueva versión del sistema buscando efectos adversos en otras partes.
- **Pruebas de Humo (Smoke Testing o Ad Hoc)**
    - Objetivo: Toma este nombre debido a que su objetivo es probar el sistema constantemente buscando que saque “humo” o falle. Permite detectar problemas que por lo regular no son detectados en las pruebas normales. Algunas veces, si las Pruebas ocurren tarde en el ciclo de desarrollo está será una forma de garantizar el buen desarrollo.
- **Pruebas de Stress**
    - Objetivo: Las pruebas de stress identifican la carga máxima que el sistema puede manejar.

El objetivo de esta prueba es investigar el comportamiento del sistema bajo condiciones que sobrecargan sus recursos.
Si se detectan errores durante estas condiciones “imposibles”, la prueba es valiosa porque es de esperar que los mismos errores puedan presentarse en situaciones reales, algo menos exigentes.

- **Pruebas de Carga**
    - Objetivo: Verificar el tiempo de respuesta del sistema para transacciones o casos de uso de negocios, bajo diferentes condiciones de carga. La meta de las pruebas de carga es determinar y asegurar que el sistema funciona apropiadamente aún más allá de la carga de trabajo máxima esperada. Adicionalmente, las pruebas de carga evalúan las características de desempeño (tiempos de respuesta, tasas de transacciones y otros aspectos sensibles al tiempo).
- **Pruebas de Volumen**
    - Objetivo: El objetivo de esta prueba es someter al sistema a grandes volúmenes de datos para determinar si el mismo puede manejar el volumen de datos especificado en sus requisitos.
- **Pruebas de Recuperación y Tolerancia a fallas**
    - Objetivo: Estas pruebas aseguran que una aplicación o sistema se recupere de una variedad de anomalías de hardware, software o red con pérdidas de datos o fallas de integridad. Las pruebas de tolerancia a fallas aseguran que, para aquellos sistemas que deben mantenerse corriendo, cuando una condición de falla ocurre, los sistemas alternos o de respaldo pueden tomar control del sistema sin pérdida de datos o transacciones.
- **Pruebas de Seguridad y Control de Acceso**
    - Objetivo: garantizan que, con base en la seguridad deseada, los usuarios están restringidos a funciones específicas o su acceso está limitado únicamente a los datos que está autorizado a acceder. Por ejemplo, cada usuario puede estar autorizado a crear nuevas cuentas, pero sólo los administradores pueden borrarlas. Si existe seguridad a nivel de datos, la prueba garantiza que un usuario “típico” 1 puede ver toda la información de clientes, incluyendo datos financieros; sin embargo, el usuario 2 solamente puede ver los datos institucionales del mismo cliente.

EL PROCESO DEL SOFTWARE

¿Que es un Proceso de Software?
Es un conjunto de actividades y resultados asociados que producen un producto de software. Es uno de los componentes de un método de desarrollo de software.
Existen 4 actividades fundamentales de proceso, comunes para todos los procesos de software:

- Especificación del software
- Desarrollo del software
- Validación del software

- Evolución del software

Concepto de ciclo de vida
Es una descripción de un proceso de software que se presenta desde una perspectiva particular. Una abstracción de un proceso real.
Existe una gran variedad de modelos diferentes “genéricos” o paradigmas de desarrollo de software.

¿Que es una metodología?
La metodología para el desarrollo del software es un modo sistemático de realizar, gestionar y administrar un proyecto para llevarlo a cabo con altas probabilidades de éxito. Son los procesos a seguir sistemáticamente para idear, implementar y mantener un software desde que nace la necesidad del producto hasta que cumplimos el objetivo para el cual fue creado

¿Qué etapas tiene un ciclo de vida?
- **Expresión de necesidades**: Armado de un documento donde se reflejan los requerimientos y funcionalidades que tendrá el producto.
- **Especificaciones**: Formalizamos los requerimientos obtenidos en el paso anterior
- **Análisis**: Ya determinamos qué elementos van a intervenir en el sistema que vamos a desarrollar. Su estructura, relaciones, funciones. Sabemos QUE producto vamos a construir
- **Diseño**: Ya sabemos qué hacer. Ahora en esta etapa determinamos cómo hacerlo. Definimos en detalle entidades y relaciones de la BD, el lenguaje que usaremos y demás
- **Implementación**: Empezamos a codificar los algoritmos y estructuras definidos anteriormente.
    - Dentro de las implementaciones del software encontramos dos principales
        - **Método directo:** Se abandona el sistema antiguo y se adopta inmediatamente el nuevo. Esto puede ser sumamente riesgoso porque si algo marcha mal, es imposible volver al sistema anterior, las correcciones deberán hacerse bajo la marcha. Regularmente con un sistema nuevo suelen surgir problemas de pequeña y gran escala. Si se trata de grandes sistemas, un problema puede significar una catástrofe, perjudicando o retrasando el desempeño entero de la organización.
        - **Método paralelo:** Los sistemas de información antiguo y nuevo operan juntos hasta que el nuevo demuestra ser confiable. Este método es de bajo riesgo. Si el sistema nuevo falla, la organización puede mantener sus actividades con el sistema antiguo. Pero puede representar un alto costo al requerir contar con personal y equipo para laborar con los dos sistemas, por lo que este método se reserva específicamente para casos en los que el costo de una falla sería considerable.

- **Debugging/Prueba**: El objetivo de esta etapa es garantizar que nuestro software no tiene errores de diseño o codificación. Deseamos encontrar la mayor cantidad de errores.
- **Validación**: Verificamos si el software desarrollado cumple con los requerimientos expresados inicialmente por el cliente
- **Evolución y Mantenimiento**: Se contempla en esta etapa la corrección de errores que pudiesen surgir (mantenimiento) como así también la incorporación de nuevas funcionalidades a nuestro software (evolución)

Codificar y corregir (Code-and-Fix)

Antes el programador realizaba un relevamiento de las solicitudes de quien necesitaba cierto programa o producto de software, y con los requerimientos bajo el brazo empezaba a programar. Esta tarea no estaba administrada, supervisada o gestionada de ningún modo. Los errores se iban corrigiendo a medida que surgían. Los programas crecieron en complejidad y esta vieja técnica de code & fix quedó obsoleta. Al no seguir normas para el proyecto, el cliente solo impartía especificaciones muy generales del producto que necesitaba. Se programa y corregía sobre la marcha.

Este es el modelo básico utilizado en los inicios del desarrollo de software. Contiene dos pasos:

- Escribir código.
- Corregir problemas en el código.

Este modelo tiene tres problemas principales:

- Después de un número de correcciones, el código puede tener una muy mala estructura, hace que los arreglos sean muy costosos.
- Frecuentemente, aún el software bien diseñado, no se ajusta a las necesidades del usuario, por lo que es rechazado o su reconstrucción es muy cara.
- El código es difícil de reparar por su pobre preparación para probar y modificar.

Modelo en cascada

El primer modelo de desarrollo de software que se publicó se derivó de otros procesos de ingeniería. Éste toma las actividades fundamentales del proceso de especificación, desarrollo, validación y evolución y las representa como fases separadas del proceso.

El modelo en cascada consta de las siguientes fases:

1. **Definición de los requisitos:** Los servicios, restricciones y objetivos son establecidos con los usuarios del sistema. Se busca hacer esta definición en detalle.
2. **Diseño de software:** Se particiona el sistema en sistemas de software o hardware. Se establece la arquitectura total del sistema. Se identifican y describen las abstracciones y relaciones de los componentes del sistema.
3. **Implementación y pruebas unitarias:** Construcción de los módulos y unidades de software. Se realizan pruebas de cada unidad.
4. **Integración y pruebas del sistema:** Se integran todas las unidades. Se prueban en conjunto. Se entrega el conjunto probado al cliente.

5. **Operación y mantenimiento:** Generalmente es la fase más larga. El sistema es puesto en marcha y se realiza la corrección de errores descubiertos. Se realizan mejoras de implementación. Se identifican nuevos requisitos.

Cada fase tiene como resultado documentos que deben ser aprobados por el usuario. Una fase no comienza hasta que termine la fase anterior y generalmente se incluye la corrección de los problemas encontrados en fases previas.

Tipos de ciclo de vida que conocemos:

- **Ciclo de vida lineal:** Es el más simple de los ciclos de vida. Consiste en etapas separadas de manera lineal. Cada etapa se realiza una sola vez, y se pasa a la etapa siguiente.
    - **Ventajas**
        - Es sencillo y de facil implementacion, ideal para proyectos pequeños
    - **Desventajas**
        - Necesitamos contar con todos los requerimientos para empezar
        - No podemos volver atrás, o es demasiado costoso hacerlo.

- **Ciclo de vida en cascada puro**: Es un ciclo de vida que admite iteraciones. De todas formas es un modelo rígido, con muchas restricciones.
    - **Ventaja**
        - podemos decir que se puede entregar un software de calidad sin contar con personal altamente calificado, ya que antes de pasar a la etapa siguiente se realiza una o varias revisiones.
    - **Desventaja**
        - contar con todos (o la mayoría) de los requisitos antes de empezar. Si se comete un error y no se detecta en la etapa siguiente es muy difícil y costoso volver atrás

- **Ciclo de vida en V** : El modelo en V es una variación del modelo en cascada que muestra cómo se relacionan las actividades de prueba con el análisis y el diseño. el modelo en V hace más explícita parte de las iteraciones y repeticiones de trabajo que están ocultas en el modelo en cascada. Mientras el foco del modelo en cascada se sitúa en los documentos y productos desarrollados, el modelo en V se centra en las actividades y la corrección.
    - **Ventajas:**
        - La relación entre las etapas de desarrollo y los distintos tipos de pruebas facilitan la localización de fallos.
        - Es un modelo sencillo y de fácil aprendizaje
        - Hace explícito parte de la iteración y trabajo que hay que revisar
        - Especifica bien los roles de los distintos tipos de pruebas a realizar
        - Involucra al usuario en las pruebas
    - **Desventajas**
        - Es difícil que el cliente exponga explícitamente todos los requisitos
        - El cliente debe tener paciencia pues obtendrá el producto al final del ciclo de vida

        - Las pruebas pueden ser caras y, a veces, no lo suficientemente efectivas
        - El producto final obtenido puede que no refleje todos los requisitos del usuario

- **Ciclo de vida tipo Sashimi**: Parecido al ciclo de vida en cascada puro, con la diferencia que en este las etapas están solapadas.
    - **Ventaja**
        - Aumenta su eficiencia porque las etapas se retroalimentan implícitamente en el modelo, lo que también ayuda a realizar un software de calidad.
    - **Desventajas**
        - Es muy difícil gestionar el inicio y el fin de cada etapa. Es una opción válida para desarrollar una aplicación que compartirá recursos con otras aplicaciones.

- **Ciclo de vida en cascada con subproyectos**: Cada una de las cascadas se divide en subproyectos o subetapas independientes que se pueden desarrollar en paralelo, lo que permite tener más gente trabajando al mismo tiempo.
    - **Ventaja**
        - Permite la entrega parcial de software
    - **Desventaja**
        - Pueden surgir dependencias entre las distintas subetapas o subproyectos. Se debe administrar muy bien.

- **Ciclo de vida incremental e iterativo**: Construye incrementando funcionalidades del sistema. Se realiza construyendo por módulos que cumplen las diferentes funciones del sistema, lo que permite ir aumentando gradualmente las capacidades del programa. Se aplica el ciclo de vida en cascada en cada funcionalidad del programa a construir. Se entrega al final de cada cascada (ciclo), una versión al cliente que contiene una nueva funcionalidad. Es una táctica de divide y conquistaras, donde si producimos pequeños sistemas es menos riesgoso y más simple que hacer uno grande. Además, no necesitamos disponer de todos los requerimientos al inicio del proyecto. Se usa cuando el usuario necesita entregas rápidas aunque sean parciales.
    - **Ventajas**
        - En el desarrollo de este modelo se da la retroalimentación muy temprano a los usuarios.
        - Permite separar la complejidad del proyecto, gracias a su desarrollo por parte de cada iteración o bloque.
        - El producto es consistente y puntual en el desarrollo.
        - Los productos desarrollados con este modelo tienen una menor probabilidad de fallar.
        - Se obtiene un aprendizaje en cada iteración que es aplicado en el desarrollo del producto y aumenta las experiencias para próximos proyectos

    - **Desventajas**
        - La entrega temprana de los proyectos produce la creación de sistemas demasiados simples que a veces se ven un poco monótonos a los ojos del personal que lo recibe.
        - La mayoría de los incrementos se harán en base de las necesidades de los usuarios.
        - Requiere de un cliente involucrado durante todo el curso del proyecto. Hay clientes que simplemente no estarán dispuestos a invertir el tiempo necesario.
        - El trato con el cliente debe basarse en principios éticos y colaboración mutua, más que trabajar cada parte independientemente, defendiendo sólo su propio beneficio.

- **Ciclo de vida en espiral**: Se basa en una serie de ciclos repetitivos para ir ganando madurez con el producto final. Se tiene en cuenta el riesgo (debido a la incertidumbre sobre los requerimientos). A medida que el ciclo se cumple (el espiral avanza) se van obteniendo prototipos sucesivos que van ganando la satisfacción del cliente.
    - **Ventajas**
        - Se puede comenzarse un proyecto con un alto grado de incertidumbre
        - Hay bajo riesgo en el retraso de la detección de errores (ya que deberían solucionarse en la próxima rama del espiral).
    - **Desventajas**
        - Costo temporal que da cada vuelta del espiral
        - Dificultad para evaluar riesgos
        - Necesidad de una comunicación continua con el cliente.
    - Se usa cuando necesitamos relevamientos exhaustivos en sistemas realmente grandes que utilizara mucha gente. *Ejemplo: sistema que administre reclamos, pedidos e incidentes*

Conclusión: ¿Cómo elegir un ciclo?
Va a depender de la **complejidad** del problema, el **tiempo** que tengamos para hacer una entrega final, si el cliente quiere o no **entregas parciales**, la **comunicación** que tenemos con el cliente, y la certeza o incertidumbre que tenemos sobre los **requerimientos** (son completos y correctos?)

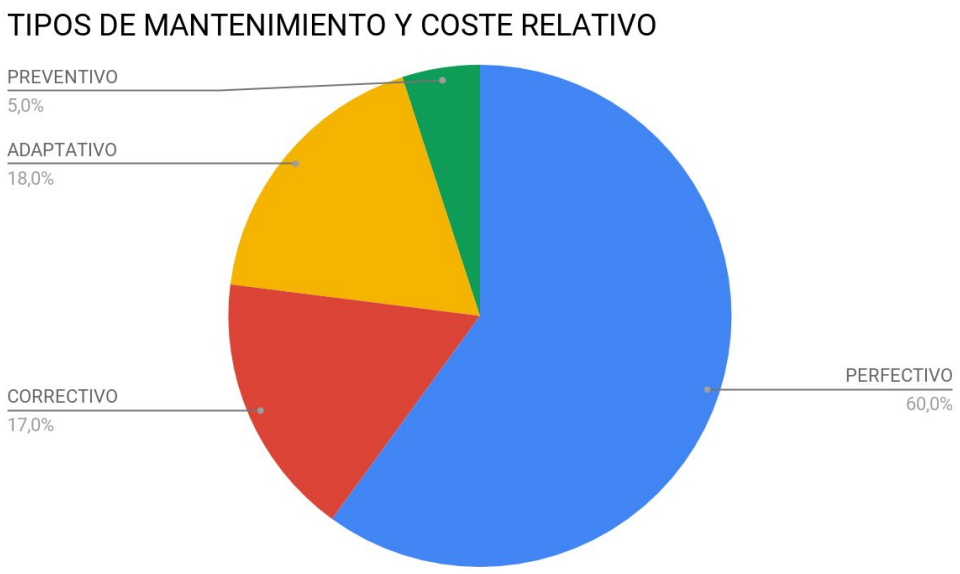
MANTENIMIENTO DEL SOFTWARE

¿QUE ES EL MANTENIMIENTO DEL SOFTWARE?

El mantenimiento del sw. es la modificación de un producto sw. después de su entrega al cliente o usuario para corregir defectos, para mejorar el rendimiento u otras propiedades deseables, o para adaptarlo a un cambio de entorno

TIPOS DE MANTENIMIENTO

- **CORRECTIVO**: es el conjunto de actividades dedicadas a corregir defectos en el hardware o software detectados por los usuarios durante la explotación del sistema.
- **PERFECTIVO**: es el conjunto de actividades para mejorar o añadir nuevas funcionalidades requeridas por el usuario. Otras posibles mejoras pueden consistir en mejorar el rendimiento de un programa, aumentar la facilidad para mantenerlo, etc.
- **ADAPTATIVO**: es el conjunto de actividades que se realizan para adaptar el sistema a los cambios en su entorno tecnológico
- **PREVENTIVO**: conjunto de actividades para facilitar el mantenimiento futuro del sistema (por ejemplo, incluir sentencias que comprueben la validez de los datos de entrada o reestructurar programas para mejorar su legibilidad)

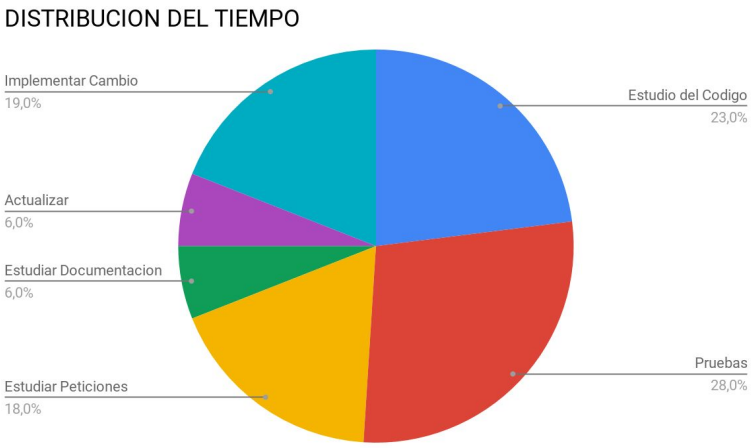


La mayoría de los estudios han identificado el **mantenimiento perfectivo** como la actividad dominante en los centros de procesos de datos

Si analizamos las actividades que deben realizar los programadores, podemos fijarnos en las siguientes:

- Estudiar peticiones
- Estudiar la documentación
- Estudiar el código
- Implementar el cambio
- Realizar pruebas
- Actualizar la documentación del programa

En la siguiente figura podemos observar la proporción del tiempo empleado por los programadores para realizar dichas actividades

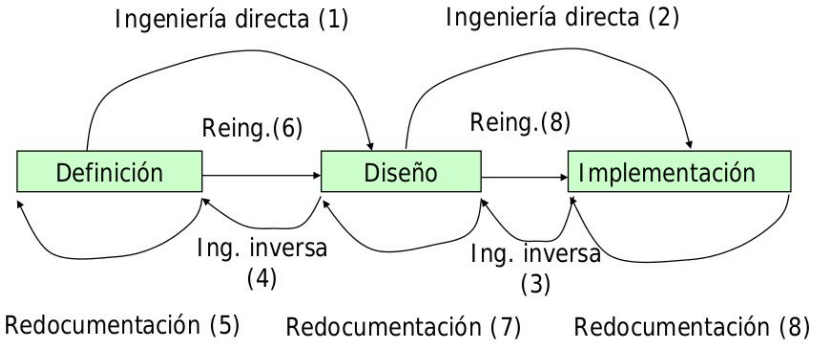


Dichas actividades pueden agruparse en funciones más genéricas como:

- **Comprensión del software y de los cambios que hay que realizar:** un programador que quiera cambiar el software necesita comprenderlo. En caso contrario, existe un gran riesgo de introducir nuevos defectos que, ademas, son los mas costosos de reparar
- **Modificación del software incorporando los cambios:** implica la creación y modificación de estructuras de datos, lógica de procesos, interfaces, etc. así como la actualización de la documentación. Los programadores se deben asegurar de que los cambios no afectarán a la coherencia del programa y que no incrementan su complejidad.
- **Reutilización de pruebas para validar los cambios:** se debe comprobar la corrección del software después de realizar cualquier cambio mediante la realización de pruebas selectivas. Pensar que un pequeño cambio no necesita la realización de pruebas puede acarrear graves problemas de calidad y de fiabilidad.

Los métodos utilizados para el mantenimiento del Software son:

- **Reingeniería**
- **Ingeniería Directa**
- **Ingeniería inversa**
- **Reestructuración del software:** consiste en la modificación del software para hacerlo más fácil de entender y cambiarlo para hacerlo menos susceptible de incluir errores en cambios posteriores.
- **Transformación de programas:** técnica formal de transformación de programas



REINGENIERÍA

Consiste en la modificación de un producto software, o de ciertos componentes, usando para el análisis del sistema, existentes tecnicas de ingenieria inversa y, para la etapa de reconstrucción, herramientas de ingeniería directa.

El objetivo de la REINGENIERÍA es la de proveer métodos para reconstruir el software.

Las actividades se pueden dividir en tres grupos:

Mejora del software

- Reestructuración
- Redocumentacion, anotación, actualización de la información
- remodularizacion
- Reingeniería de procesos
- Reingeniería de datos
- Análisis de facilidad de mantenimiento, análisis económico

Comprensión del Software

- Visualización
- Análisis, mediciones
- Ingeniería inversa, recuperacion de diseño

Captura, conservación y extensión del conocimiento sobre el software

- Descomposición
- Ingeniería Inversa y recuperacion de diseño
- Recuperación de Objetos
- Comprensión de Programas
- Transformaciones y bases de conocimiento

INGENIERÍA DIRECTA

Corresponde al desarrollo de software adicional, el objetivo “directa” se incluye exhaustivamente para diferenciar este concepto de la ingeniería inversa

INGENIERÍA INVERSA

Según *Chikofsky*, es el proceso de análisis de un sistema para identificar sus componentes e interrelaciones y crear representaciones del sistema en otra forma o a un nivel más alto de abstracción.

Considera dos áreas incluida en la ingeniería inversa:

- **REDOCUMENTACION:** es la creación o revisión de una representación equivalente semánticamente dentro del mismo nivel de abstracción relativo. Es la creación de información correcta y actualizada del sistema
- **RECUPERACIÓN DEL DISEÑO:** es un subconjunto de la ingeniería inversa, en el cual aparte de las observaciones del sistema, se añaden conocimientos sobre su dominio de aplicación, información externa, y procesos deductivos con el objetivo de identificar abstracciones significativas a un mayor nivel.

¿QUÉ BENEFICIOS OBTENGO DE ESTE MÉTODO?

- Reducir la complejidad del sistema
- Generar vistas alternativas
- Recuperar la información perdida (cambios que no se documentaron en su momento)
- Detectar efectos laterales
- Facilitar la reutilización