INGENIERIA SOFTWARE I

Unidad I: Ingeniería de software

¿Qué es el software?
Conjunto de todos los documentos asociados y configuración de datos que se necesitan para hacer que los programas operen de manera correcta.

¿Qué es la Ing. de software?
La ingeniería de software es la disciplina que se encarga del diseño, desarrollo, implementación, mantenimiento y gestión de software utilizando principios y métodos de la ingeniería para abordar problemas de manera organizada y sistemática. El objetivo es desarrollar sistemas de software de alta calidad y eficiencia que cumplan con los requisitos del usuario.

Proceso de resolución de problemas
Abarca todos los aspectos que van desde interpretar las necesidades del usuario hasta verificar que la respuesta brindada es correcta.
4 etapas:

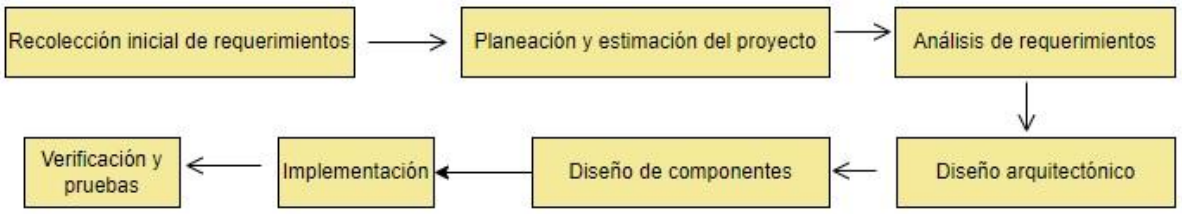
- **Análisis del problema**: Se analiza el problema en su contexto del mundo real. Deben obtenerse los requerimientos del usuario. El resultado de este análisis es un modelo del proceso del ambiente del problema y del objetivo a resolver.
- **Diseño de una solución**: El diseño de la resolución de problemas consiste en la identificación de las partes (subproblemas) que componen el problema y la manera en que se relacionan.
- **Implementación de la solución**: Se debe elegir los lenguajes y algoritmos adecuados para el desarrollo del programa.
- **Validación**: Se verifica que la ejecución del sistema en su totalidad, produce los resultados deseados, utilizando datos representativos del problema real.

Proceso de construcción del software
Conjunto estructurado de las actividades requeridas para construir un sistema. Se utiliza para mejorar la comprensión del problema a resolver, la comunicación entre los participantes del proyecto y el mantenimiento de un sistema complejo. 5 actividades fundamentales:

- **Especificación de requerimientos.**
- **Diseño.**
- **Implementación.**
- **Validación.**
- **Mantenimiento.**

3 macro etapas:

- **Definición** (¿Qué?) - Comprensión del dominio del problema recolección de los requerimientos, planeación y estimación del proyecto, y análisis de requerimientos.
- **Desarrollo** (¿Cómo?) - Diseño de arquitectura y de los componentes, codificación, verificación y prueba de los programas, y del sistema.
- **Mantenimiento** (Cambios) - Correcciones, adaptaciones, extensiones, mejoras, etc.



Proceso de software

Conjunto de actividades y resultados asociados que producen un producto de software.
4 actividades fundamentales:

1) **Especificación de software**: Se define el software a producir y las restricciones sobre su operación.
2) **Desarrollo del software**: Diseño y programación.
3) **Validación del software**: Se asegura que es lo que el cliente requiere.
4) **Evolución del software**: Modificación del software para adaptarlo a los cambios requeridos por el cliente y el mercado.

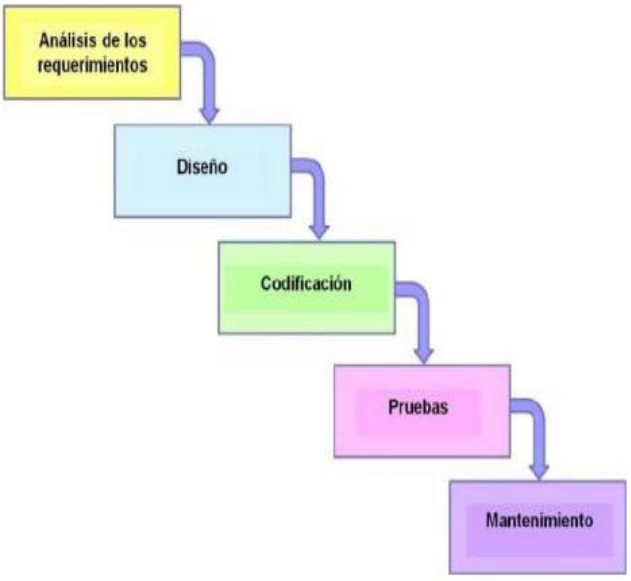
NOTA: **"Diferentes tipos de sistemas necesitan diferentes procesos de desarrollo".**

Representación de procesos de software: Descripción simplificada de un proceso del software que presenta una visión de ese proceso.

- **Flujo de trabajo**: Las actividades en este modelo representan acciones humanas (DER?).
- **Flujo de datos**: Muestra como la entrada en el proceso, se transforma en una salida, tal como un diseño (DFD?).
- **Relación**: Representa los roles de las personas involucradas en el proceso del software.

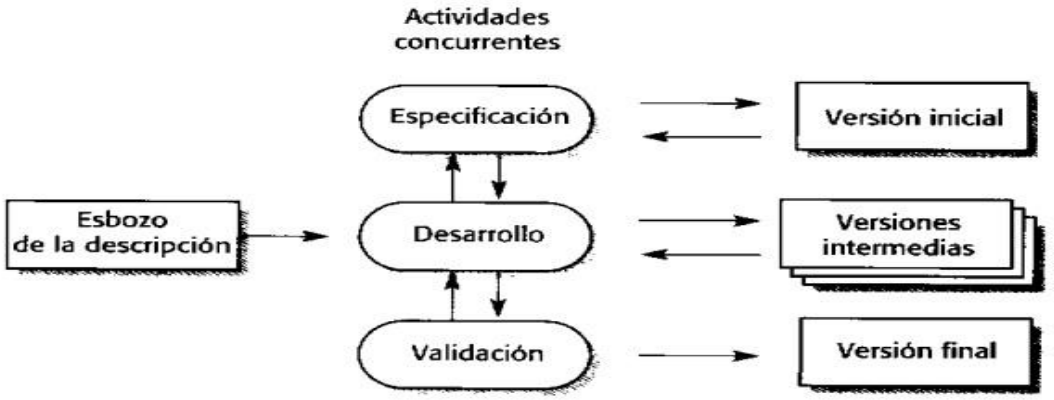
Modelos de procesos software
Los modelos a continuación no se excluyen y a menudo se utilizan juntos:

- **Modelo cascada**: Considera las actividades fundamentales del proceso de software: *especificación, desarrollo, validación y evolución*, y las representa como fases separadas del proceso.
  Las principales etapas de este modelo se transforman en actividades fundamentales del desarrollo:
  a) **Análisis y definición de requerimientos**: Son los servicios, restricciones y metas del sistema que se definen a partir de las consultas con los usuarios.
  b) **Diseño:** Identifica y describe las abstracciones fundamentales del sistema software y sus relaciones.
  c) **Implementación y pruebas de unidades:** El diseño del software se lleva a cabo como un conjunto o unidades de programas. La prueba implica verificar que cada una cumpla su especificación.
  d) **Integración y prueba de diseño:** Se integran las unidades y se prueba como sistema completo.
  e) **Funcionamiento y mantenimiento**: Puesta en uso y el mantenimiento, implica, corregir errores no descubiertos en las etapas anteriores.



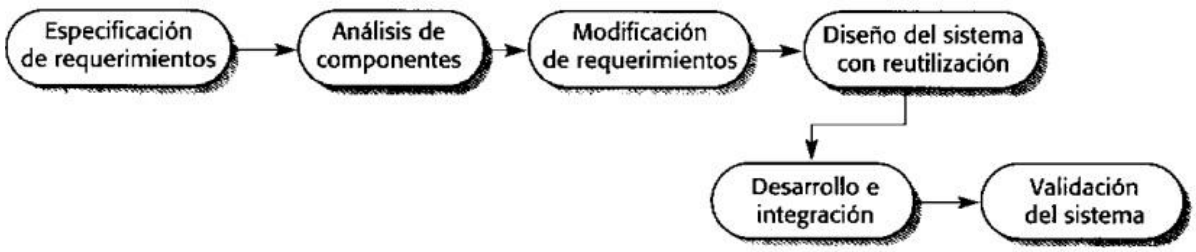
  El resultado de cada fase es uno o más documentos, y la siguiente no debe empezar hasta que la fase previa haya finalizado.

- **Desarrollo evolutivo**: Desarrolla una implementación inicial, exponiéndose a los comentarios del usuario y refinando a través de las diferentes versiones hasta que se desarrolla un sistema adecuado.
  2 Tipos de desarrollo evolutivo:
  1) **Desarrollo exploratorio:** Se trabaja con el cliente para explorar sus requerimientos y entregar un sistema final. Aquí se encuentran los siguientes ciclos de vida:
     - Incremental;
     - Espiral;
     - Iterativo;
     - Prototipado evolutivo.
  2) **Prototipos desechables:** El objetivo es comprender los requerimientos del cliente y entonces desarrollar una definición mejorada de los requerimientos para el sistema.



- **Ingeniería del software basada en componentes**: Se compone de una gran base de componentes de software *reutilizables* de algunos marcos de trabajo de integración para estos.
  4 Etapas:
  1) **Análisis de componentes**: Se buscan los componentes para implementar la especificación de requerimientos.
  2) **Modificación de requerimientos**: Los requerimientos se modifican para reflejar los componentes disponibles.
  3) **Diseño del sistema con reutilización**: Se diseña o se reutiliza un marco de trabajo para el sistema.
  4) **Desarrollo e integración**: El software que no se puede adquirir externamente se desarrolla y los componentes y los sistemas COTS* se integran.

***Componente comercial salido del estante** o **COTS** (del inglés Commercial Off-The-Shelf).*



Metodología

Modo sistemático de realizar, gestionar y administrar un proyecto para llevarlo a cabo con altas probabilidades de éxito.
En definitiva, son los procesos a seguir sistemáticamente para idear, implementar y mantener un producto software desde que surge la necesidad del producto hasta que cumplimos el objetivo por el cual fue.

5 Etapas:

- **Inicio**: Nacimiento de la idea. Se definen los objetivos del proyecto y los recursos necesarios para su ejecución.
- **Planificación**: Planeamiento detallado que guíe la gestión del proyecto, temporal y económica.
- **Implementación**: Conjunto de actividades que componen la realización del producto.
- **Control de producción**: Se controla el producto, analizando cómo el proceso difiere o no de los requerimientos originales e iniciando las acciones correctivas si fuesen necesarias.
- **Puesta en producción**: El proyecto entra en la etapa de definición, sabiendo que funciona correctamente y responde a los requerimientos solicitados en su momento.

Objetivos de cada etapa: Se pueden establecer una serie de objetivos, tareas y actividades que lo caracterizan.

Finalidad de una metodología
Se busca prolijidad, corrección y control de cada etapa del desarrollo de un programa.

Importancia de una metodología

Es importante ya que utilizar una metodología nos proporciona una gran herramienta para generar *eficiencia* a medida que se va utilizando. El uso de una metodología en la gestión de un proyecto persigue unos beneficios específicos: Organizar los tiempos de proyecto, proporcionar herramientas para estimar de forma correcta tiempos y costos.

Clasificación de metodologías:

- **Metodología estructurada:** La orientación de esta metodología se dirige hacia los procesos que intervienen en el sistema a desarrollar, es decir, cada función a realizar se descompone en pequeños módulos individuales.
- **Metodología orientada a objetos:** Arma módulos basados en componentes, esto permite la reutilización del código.

Ciclo de vida

Proceso que se sigue para construir, entregar y hacer funcionar el software, desde la concepción de la idea del sistema hasta su entrega y el desuso.

Etapas de un ciclo de vida

- **Expresión de necesidades**: Tiene como objetivo el armado de un documento en el cual se reflejan los requerimientos y funcionalidades que ofrecerá el usuario al sistema a implementar.
- **Especificación**: Se formalizan los requerimientos.
- **Análisis**: Se determinarán los elementos que intervienen en el sistema a desarrollar, su estructura, relaciones, evolución temporal, funcionalidades, etc.
- **Diseño**: Sabiendo que hacer, ahora se determinará el cómo hacerlo.
- **Implementación**: Empezamos a codificar algoritmos y estructuras de datos, definidos en las etapas anteriores, en el correspondiente lenguaje de programación o para un determinado sistema gestor de bases de datos. 2 formas de implementación:
    a) **Directa**.
    b) **Paralela**.
- **Debugging**: Se garantiza que nuestro programa no contiene errores de diseño o codificación; el objetivo es encontrar la mayor cantidad de errores.
- **Validación**: Se verifica que el sistema desarrollado cumple con los requerimientos expresados inicialmente por el cliente.
- **Evolución**: Se agregan, quitan o modifican funcionalidades (Evolución) y/o, la corrección de errores que surgen (Mantenimiento).

Modelos de ciclos de vida

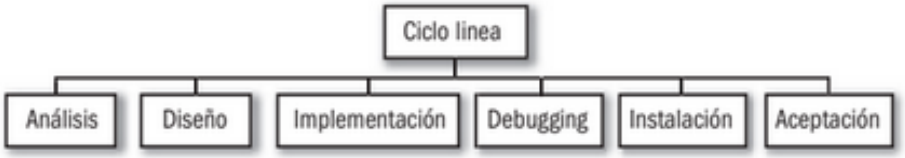
3 grandes visiones:

1) **Alcance del ciclo de vida:** Hasta donde deseamos llegar con el proyecto.
2) **Cualidad y cantidad de etapas**: En que dividiremos el ciclo de vida.
3) **La estructura y la sucesión de las etapas:** Si hay realimentación entre ellas, y si tenemos libertad de repetirlas.

¿Qué ciclos de vida conocemos?

- **Ciclo de vida lineal**: es el más simple de todos. Consiste en etapas separadas de manera lineal, donde cada etapa se realiza una sola vez y se pasa a la siguiente.
    - **Ventajas**:
        - Es sencillo y de fácil implementación, ideal para proyectos pequeños.
    - **Desventajas**:
        - No se puede empezar hasta tener todos los requerimientos bien definidos.

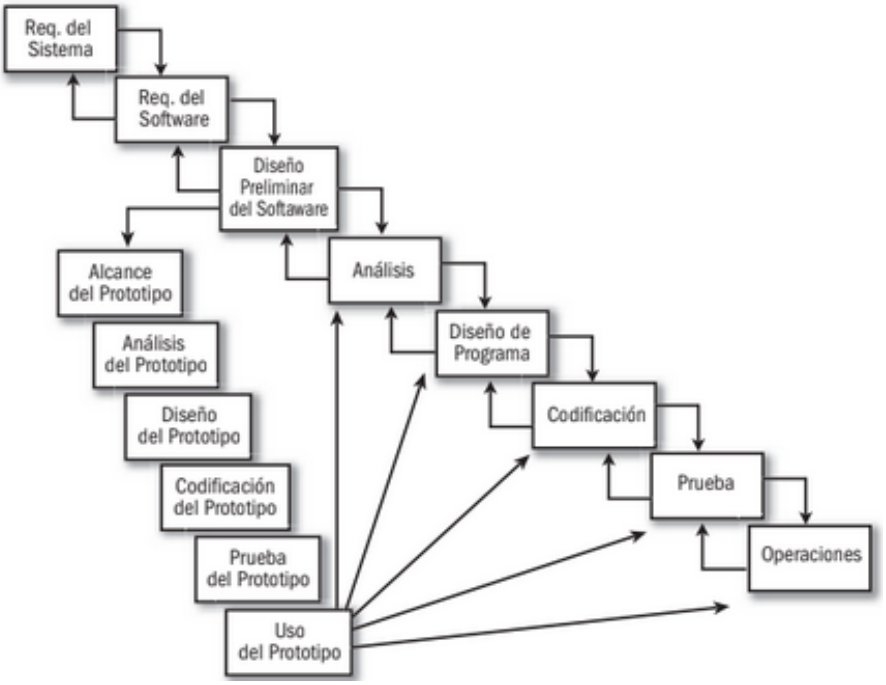
- No se puede volver atrás o es demasiado costoso volver.



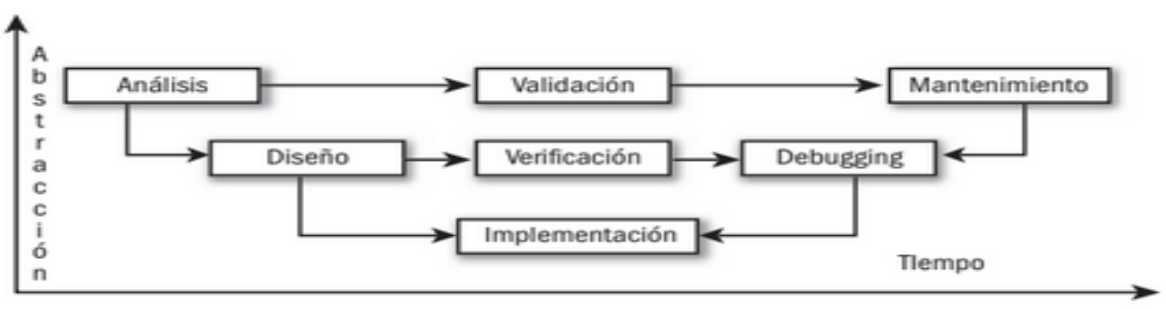
•**Ciclo de vida en cascada puro**: Es un ciclo de vida que admite iteraciones, pero aun así es rígido y con muchas restricciones.

- **Ventaja**:
  - Se puede realizar software de calidad sin contar con personal altamente calificado, ya que antes de pasar a la siguiente etapa se pueden hacer una o más revisiones.

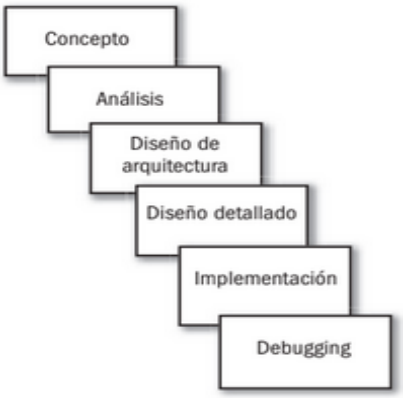
  Desventajas:
  - Contar con todos o la mayoría de los requerimientos al comenzar. Si se comete un error y no se detecta al pasar a la siguiente se vuelve costoso retroceder



- **Ciclo de vida en V**: variación del modelo en cascada con la diferencia de que a este se le agregaron dos etapas de retroalimentación entre las etapas de análisis y mantenimiento, y entre las de diseño y debugging.
  - **Ventajas**:
    - Facilita la localización de fallos.
    - Es sencillo y de fácil aprendizaje.
    - Hace explícita la parte de la iteración que hay que revisar.
    - Especifica bien los roles de los distintos tipos de pruebas a realizar.
  - **Desventajas**:
    - Es difícil que el cliente exponga explícitamente todos los requisitos.
    - El cliente debe ser paciente pues el producto se entrega al final del ciclo de vida.

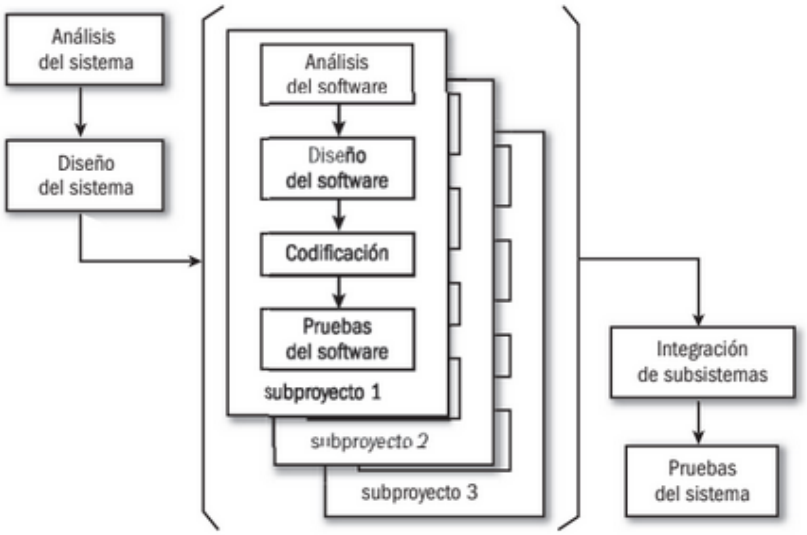


- **Ciclo de vida Sashimi**: parecido al ciclo de vida en cascada puro, con la diferencia de que en este las etapas se pueden solapar.

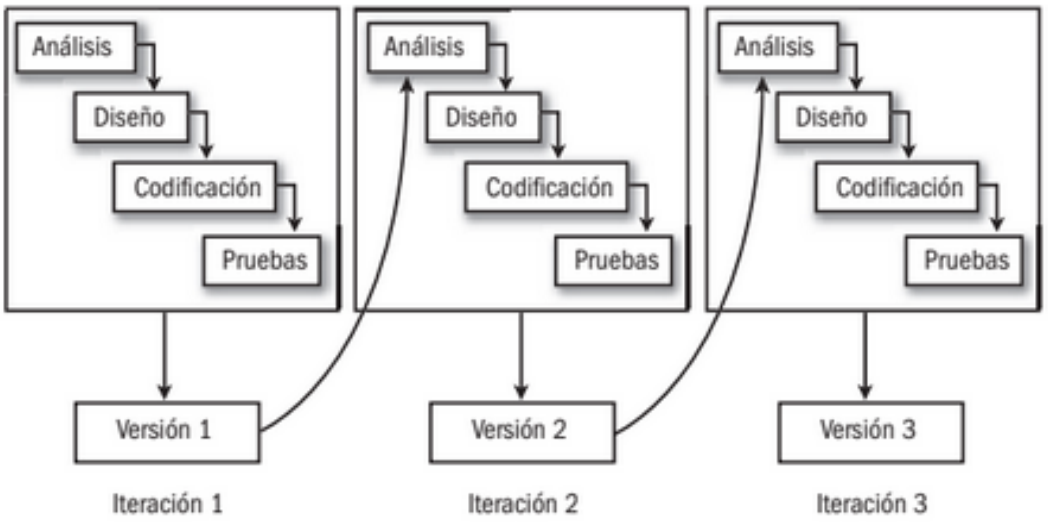


    - **Ventajas**:
        - Producto final de mayor calidad
        - Falta de necesidad de documentación detallada (el ahorro proviene por el solapado de las etapas)
    - **Desventajas**:
        - Es muy difícil gestionar el comienzo y el fin de cada etapa.
        - Posibles problemas de comunicación pueden generar inconsistencias en el proyecto.

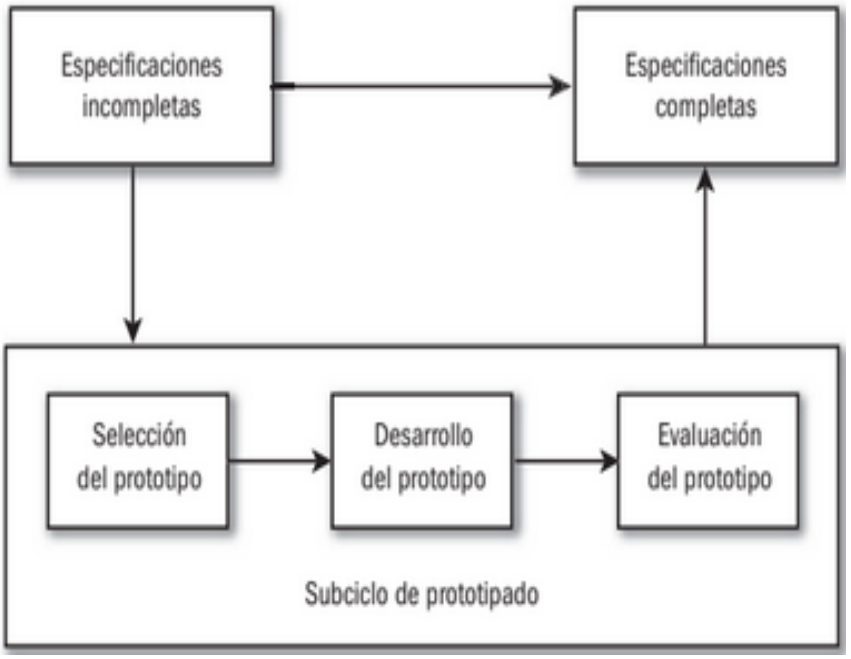
- **Ciclo de vida en cascada con subproyectos**: cada una de las cascadas se subdividen en etapas que se pueden desarrollar independientemente en paralelo.
    - **Ventajas**:
        - Se pueden tener a más personas trabajando al mismo tiempo.
    - **Desventajas**:
        - Pueden surgir dependencias entre las subetapas que requieran detener temporalmente el proyecto si no es gestionado de manera correcta.



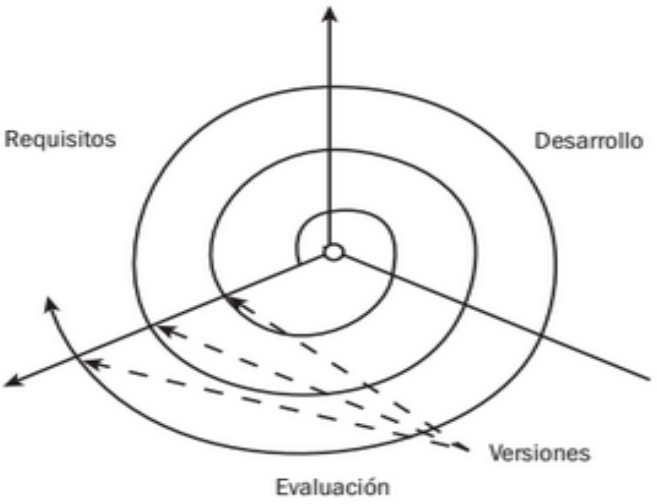
- **Ciclo de vida iterativo**: también derivado del ciclo de cascada puro, busca reducir el riesgo que surge entre las necesidades del usuario y el producto final por malos entendidos durante la solicitud de los requerimientos. Es la iteración de varios ciclos en cascada. Es el cliente quien luego de cada iteración evalúa el producto y propone cambios o mejoras.
    - **Ventajas**:
        - Es ideal en casos donde el cliente requiere entregas rápidas aunque el proyecto no esté terminado.
        - Se obtiene un aprendizaje luego de cada iteración.
    - **Desventajas**:
        - Requiere que el cliente esté involucrado durante todo el curso del proyecto.
        - El trato con el cliente debe ser de colaboración mutua.



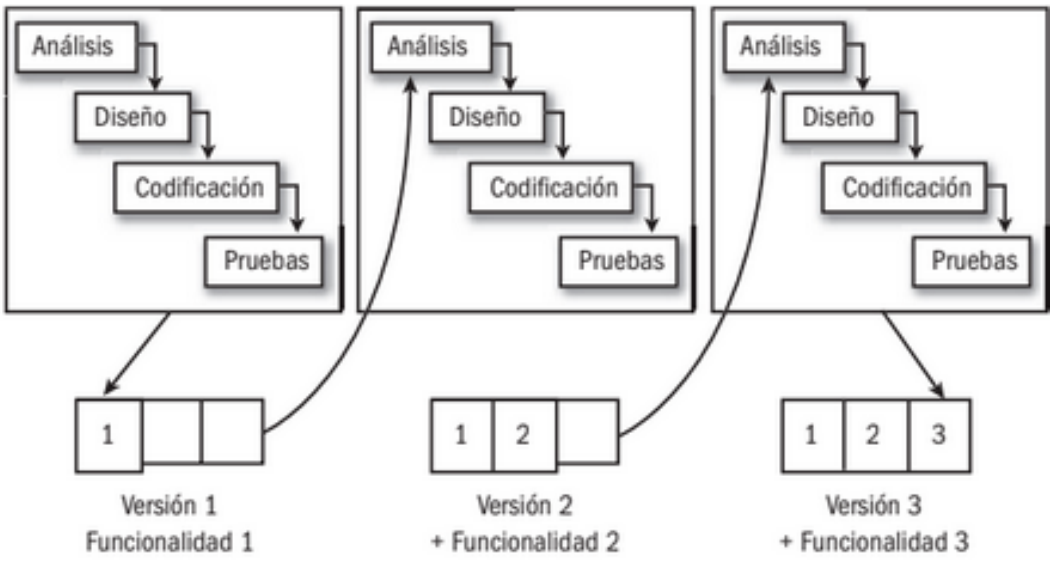
- **Ciclo de vida por prototipos**: se busca crear un producto provisional y parcial (un prototipo) si no se conocen las especificaciones de forma precisa. La idea es realizar un producto intermedio antes de de realizar un producto final
    - **Ventajas**:
        - Permite conocer cómo responderán las funcionalidades previstas para el producto final.
        - Es apto para desarrollos donde no se conoce a priori sus especificaciones ni la tecnología a utilizar.
    - **Desventajas**:
        - Se debe evaluar si el esfuerzo de realizar un prototipo vale realmente.
        - Tiende a ser altamente costoso y difícil para la administración temporal.



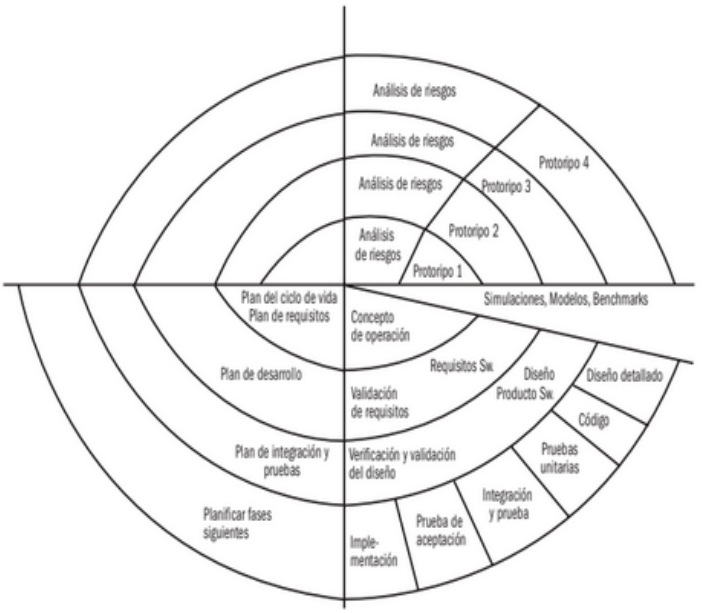
- **Ciclo de vida evolutivo**: este modelo acepta que los requerimientos del usuario puedan cambiar en cualquier momento. Debido a la dificultad a la que se enfrentan los usuarios de transmitir su idea como así también a los cambios constantes que sufren los requerimientos este modelo afronta dichos problemas mediante una iteración de ciclos **requerimientos-desarrollo-evaluación**.
    - **Ventajas**:
        - Se utiliza en desarrollos donde no se conocen a priori las especificaciones o la tecnología a utilizar.
        - Es bueno a la hora de realizar migraciones.
    - **Desventajas**:
        - Tiende a ser altamente costoso y difícil para la administración temporal.



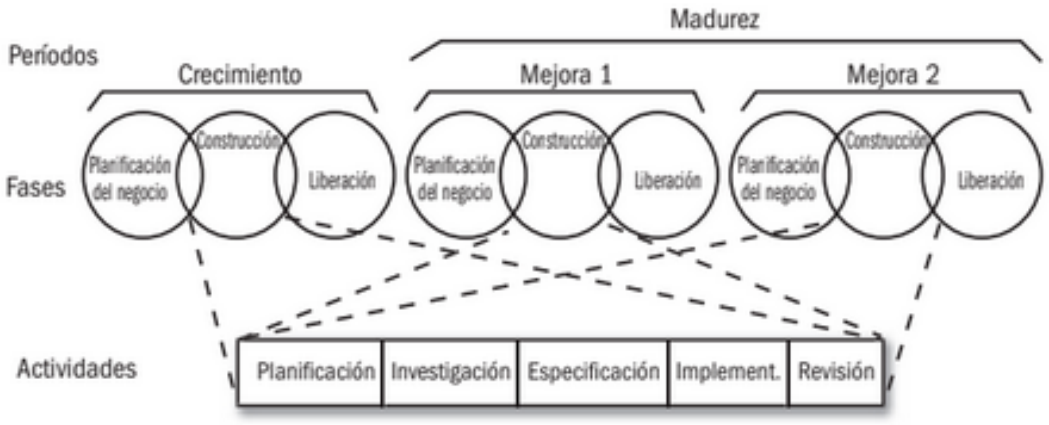
- **Ciclo de vida incremental**: se basa en la filosofía de construir incrementando las funcionalidades del programa. Se realiza construyendo por módulos que cumplen las diferentes funciones del sistema.
    - **Ventajas**:
        - Es menos riesgoso para proyectos pequeños.
        - Es más fácil relevar los requerimientos.
        - Si se detecta algún error grave, solo se desecha la última iteración.
        - No es necesario conocer todos los requerimientos.
        - Hace uso de la filosofía **divide y vencerás**.
    - **Desventajas**:
        - Es más riesgoso en proyectos grandes.oll;l



- **Ciclo de vida en espiral**: toma los beneficios de los ciclos de vida por prototipos e incremental pero se tiene más en cuenta el riesgo que surge debido a la incertidumbre proporcionada al principio del proyecto. A medida que el ciclo se cumple (el avance del espiral), se van obteniendo prototipos sucesivos que van ganando la satisfacción del cliente. Hay cuatro etapas que envuelven las actividades: **Planificación**, **Análisis de riesgo**, **Implementación** y **Evaluación**.
    - **Ventajas**:
        - Se puede empezar con un alto grado de incertidumbre.
        - Bajo riesgo de retraso en caso de detección de errores.
        - Adecuado para grandes proyectos dentro de una empresa.
    - **Desventajas**:
        - Costos temporales que suman cada vuelta del espiral.
        - Comunicación continua con el cliente o usuario.



- **Ciclo de vida orientado a objetos**: cada funcionalidad o requerimiento solicitado por el cliente es considerado un objeto, representado por **atributos** y **métodos**. La característica de esta metodología es la **abstracción** de los requerimientos del usuario, lo que los hace mucho más flexibles que los restantes. Utiliza las llamadas fichas **CRC (clases-responsabilidades-colaboración)**, las cuales facilitan la implementación de los requerimientos y ayudan a confeccionar los **casos de uso**.
    - **Ventajas**:
        - Se considera un modelo pleno a seguir, como así también una alternativa a los modelos anteriores.
        - Se puede usar cualquier lenguaje, independientemente de si este soporta POO o no, ya que es una metodología de desarrollo, no un paradigma de programación.



Unidad II: Ingeniería de Requerimientos

La ingeniería de requerimientos nos proporciona un mecanismo apropiado para entender lo que desea el cliente, analizar las necesidades, evaluar la factibilidad, negociar una solución razonable, especificar dicha solución sin ambigüedades, validar las especificaciones y administrar los requerimientos obtenidos con técnicas de recogida de información a medida que se transforman en un sistema funcional.

Posee 7 tareas:

1. **Concepción**: Se establece el entendimiento básico del problema.
2. **Indagación**: Preguntar cuáles son los objetivos para el sistema. Se identifican cierto número de problemas, cuales pueden ser:
    a. **Problema de alcance**: No se conocen los límites del problema.
    b. **Problema de entendimiento**: No se sabe lo que se necesita.
    c. **Problema de volatilidad**: Los requerimientos cambian con el tiempo.
3. **Elaboración**: Desarrolla un modelo refinado de los requerimientos, que identifique distintos aspectos de la función del software, su comportamiento e información.
4. **Negociación**: Se negocia y analiza con el cliente/usuario y otros participantes, el orden y prioridad de los requerimientos para luego con el empleo de un enfoque iterativo que da prioridad a los requerimientos, se evalúa su costo y riesgo, y se enfrentan los conflictos internos; algunos requerimientos se eliminan, se combinan o se modifican de modo que cada parte logre cierto grado de satisfacción.
5. **Especificación**: Puede ser un documento escrito, un conjunto de modelos gráficos, un modelo matemático formal, un conjunto de escenarios de usos, un prototipo o cualquier combinación de estos.
6. **Validación**: Se analiza la especificación a fin de garantizar que todos ellos han sido enunciados sin ambigüedades, que se detectaron y corrigieron las inconsistencias, las omisiones y los errores, y que los productos del trabajo se presentan conforme a los estándares establecidos para el proceso, el proyecto y el producto. *Se valida que los **requerimientos** sean:*
    ➔ Necesarios.
    ➔ Completos.
    ➔ Verificables
    ➔ No Ambiguos
    ➔ Concisos
    ➔ Consistentes
    ➔ Verificables.
7. **Administración de los requerimientos:** Conjunto de actividades que ayudan al equipo del proyecto a identificar, controlar y dar seguimiento a los requerimientos y a sus cambios en cualquier momento del desarrollo del proyecto.

<u>En conclusión:</u>

"La ingeniería de requerimientos es el proceso de descubrir, analizar, documentar y verificar estos servicios y restricciones que dispondrá el sistema".

¿Qué es un requerimiento?
Descripción de los servicios proporcionados por el sistema y sus restricciones operativas. Reflejar las necesidades de los clientes. 2 formas de requerimientos:

1) **Requerimiento del usuario**: A veces es una declaración abstracta de alto nivel, un servicio que debe proporcionar el sistema o una restricción de esto.
2) **Requerimiento del sistema**: Definición detallada y formal de una función del sistema.

Tipos de Requerimientos

Funcionales: Declaraciones de los *servicios o funcionalidades* que debe proporcionar el sistema, cómo debe reaccionar ante determinada entrada de datos y su comportamiento, también lo que el sistema no debe hacer.

"Lo que esperamos que deba hacer el sistema".

No Funcionales: Restricciones de los servicios o funciones ofrecidos por el sistema. Características no relacionadas con la funcionalidad que son importantes para el sistema, como la seguridad, la usabilidad, el rendimiento, la escalabilidad y la compatibilidad. A menudo se aplica al sistema en su totalidad. Existen 3 tipos de este:

a. Requerimiento del producto: Especifican el comportamiento del producto, (ej: Rendimiento).
b. Requerimiento organizacionales: Derivan de políticas y procedimientos existentes en la organización del cliente y en la del desarrollador. (ej: Estándares).
c. Requerimientos externos: Derivan de los factores externos al sistema y de su proceso de desarrollo. (ej: Legislaciones).

"Lo que define cómo debe ser el sistema".

Actividades del proceso IR

- **Comprensión del dominio**
- **Recolección de requisitos**
- **Clasificación**
- **Resolución de conflictos**
- **Priorización**
- **Verificación de requisitos**
- **Análisis**

Importancia del proceso IR

Gestionar necesidades en forma ordenada con proyectos más controlables, con disminución de costos y proyecciones factibles, también facilita la calidad del software, mejora la comunicación del equipo de trabajo y favorece la aceptación del producto.
En resumen, 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

Análisis de requerimientos

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.
El análisis cuenta con 4 actividades:

1. **Descubrimiento de requerimientos**: Proceso de interacción con los clientes/usuarios del sistema para recopilar sus necesidades.
2. **Clasificación y organización de requerimientos**: Organiza los requerimientos en grupos coherentes
3. **Organización por propiedades y negociación de requerimientos**: El objetivo es ordenar según las prioridades, y a encontrar y resolver los requerimientos en conflicto a través de la negociación.
4. **Documentación de requerimientos**: Producción de documentos formal o informal de los requerimientos.

OPCIONAL:

1. 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.
2. 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
3. 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.
4. 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

FIN OPCIONAL

Evaluación y negociación de requerimientos

- Descubrir requerimientos potenciales.
- Clasificar los requerimientos.
- Evaluar factibilidad y riesgos
- Especificación de requisitos de software

Identificación de necesidades

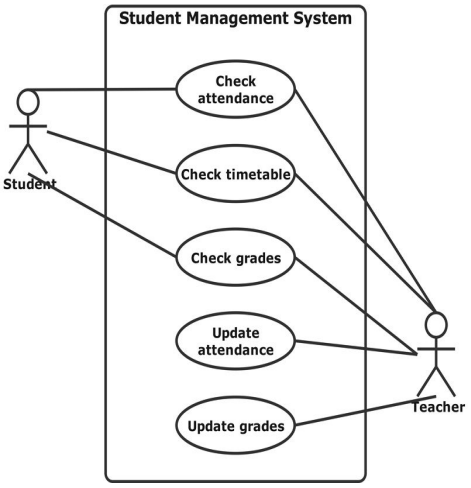
Tiene como objetivo principal la comprensión de los clientes y los usuarios que esperan que haga el sistema
En este proceso se identifican los aspectos claves que el sistema requiera y se descubra los aspectos irrelevantes del mismo.

Técnicas de recogida de información

- **Entrevistas o Cuestionarios**: se emplean para reunir información de personas o grupos. Las preguntas deben ser de alto nivel, para obtener información sobre aspectos globales del problema del usuario y soluciones potenciales.
- **JAD (Joint Application Design)**: consiste en un conjunto de reuniones entre los analistas y los usuarios. Se comienza con una documentación y se discuten los cambios agregados. Al final se obtiene la DOCUMENTACIÓN DE REQUISITOS aprobada.
- **Prototipo**: es una simulación del posible producto. Es una interfaz que permite ser evaluada por el usuario.
- **Tormenta de ideas**: se usa para generar ideas. En principio toda idea es válida y ninguna debe ser rechazada.

¿Qué son los Casos de Uso?

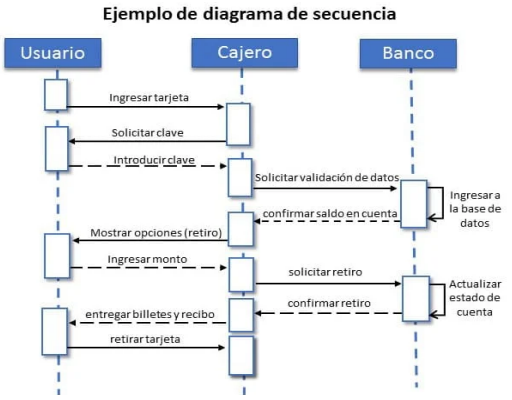
Son una **secuencia de interacciones** entre un sistema o una solución y el usuario u otro componente que utilice sus servicios. Pueden agrupar muchos eventos.



Técnica de modelado utilizada en el diseño de software para describir cómo los usuarios interactúan con el sistema y para identificar los requisitos funcionales del software. Se describen los pasos específicos que los usuarios realizan para lograr una tarea dentro del sistema, y también describen cómo el sistema responde a esas acciones del usuario.
Se representan típicamente en forma de *diagramas* de casos de uso, que muestran los diferentes actores (como usuarios o sistemas externos) y *los casos de uso* que estos actores pueden realizar en el sistema. También pueden incluir escenarios alternativos que describen lo que sucede cuando se producen excepciones o situaciones imprevistas.

¿Qué son los Eventos?

Describen el comportamiento del sistema y solo poseen **entrada/proceso/salida**. Esto puede hacerse mediante la creación de diagramas de secuencia, que muestran la secuencia de eventos y las acciones correspondientes que ocurren en el sistema en respuesta a cada evento.



¿Qué es la Especificación de Requisitos del Software?

Es un documento que contiene una descripción completa de las necesidades y funcionalidades del sistema que se está desarrollando. Describe el alcance del sistema, las funcionalidades que realizará, definiendo requisitos funcionales y no funcionales.

¿Que es un prototipo?

Es la primera versión de un nuevo tipo de producto, en la que se han incorporado solo algunas características del sistema final. Es un modelo o maqueta del sistema que se construye para comprender mejor el problema y sus posibles soluciones.
Características:

- Es de funcionalidad limitada.
- Poca fiabilidad.
- Características de operación pobres.

Se presenta al cliente un prototipo para su experimentación, ayuda al cliente a establecer claramente los requisitos y a los desarrolladores a mejorar el producto, examinar la utilidad de la aplicación y aprender sobre problemas que se presentarán durante el diseño e implementación de la misma

Estudio de factibilidad

Establece los requerimientos y restricciones básicas del negocio, asociados con el software que se va a construir, en este estudio tenemos 4 divisiones:

1. **Tecnología**: ¿Existe la tecnología para llevarlo a cabo?
2. **Finanzas**: ¿El software puede ser costeado por el cliente?
3. **Tiempo**: Vencer a la competencia.
4. **Recursos**: ¿La organización tiene los recursos para llevar a cabo el proyecto?

SRS : “Software requirements specification”

Documento que define e informa de manera completa, precisa, no ambigua, concisa y verificable los requisitos, el diseño y el comportamiento u otras características de un sistema o componente de un sistema.

Es el producto del análisis de requerimientos, que proporciona las pautas a seguir por los desarrolladores del sistema.

Características:

- **Correcta**
- **No ambigua**
- **Completa**
- **Consistente**
- **Con prioridades**
- **Verificable**
- **Modificable**
- **Rastreable**

Validación y Verificación de los requerimientos

Validación: El objetivo es detectar defectos en los requerimientos antes de invertir tiempo y dinero en las siguientes etapas del proceso de desarrollo.

En definitiva, comprueba que los requerimientos realmente definen el sistema que el cliente desea y que lo describen. Es lo que el cliente pretende ver en el producto final.

Posee 2 principales técnicas de validación, revisiones de requerimientos y construcción de prototipos

- **Revisión**: Se verifica que el documento de requerimientos no presenta anomalías ni omisiones.
- **Prototipos**: Se presenta un modelo ejecutable del sistema a los usuarios finales, este modelo puede no tener todas las funcionalidades completas.

La validación contesta la pregunta de si se está construyendo lo que el cliente pidió.

Verificación: Se planifica cómo verificar cada requerimiento. Si un requerimiento no se puede probar significa que el requerimiento no es correcto y es necesario y obligatorio replantearse.
La verificación contesta la pregunta de si lo que pidió el cliente funciona correctamente.

Gestión de los requerimientos

Proceso de organizar y llevar a cabo los cambios en los requerimientos con el objetivo de asegurar la consistencia entre los requerimientos y el sistema construido.

EXTRA
Requerimiento: Son las necesidades que provienen del cliente/usuario.

Requisito: Son las especificaciones puntuales sobre el sistema, que vienen a ser la solución a las necesidades del cliente.

Unidad V: El Proceso de Diseño

El diseño es un proceso en el que se determina la estructura del sistema de software y de sus datos antes de iniciar su codificación.
En la etapa de diseño nos preguntamos ¿Cómo?
La fase de diseño del sistema software parte de los resultados del *análisis de los requerimientos*, los cuales deben proporcionar los elementos claves para determinar la estructura del software.
Se debe concentrarse en cómo vamos a diseñar para ello existen 4 estilos:

- Diseñar para el cambio;
- Diseñar para la extensión;
- Diseñar para la eficiencia;
- Diseñar para la sencillez.

¿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.
Conceptos de diseño

- **Abstracción**: Se extraen características comunes a partir de ejemplos específicos. 2 tipos de abstracción:
    a) Abstracción de procedimiento: Secuencia de instrucciones que tienen una función específica y limitada.
    b) Abstracción de datos: Conjuntos de estos con nombre, que describen a un objeto de datos.

- **Refinamiento sucesivo:** Se desarrolla una jerarquía con la descomposición de un enunciado macroscópico de la función (abstracción del procedimiento) en forma escalonada hasta llegar a los comandos del lenguaje de programación.
  La abstracción y el refinamiento son conceptos complementarios. La primera permite especificar internamente el procedimiento y los datos. El refinamiento ayuda a revelar estos detalles a medida que avanza el diseño.
  IA : El refinamiento sucesivo en la ingeniería de software es una técnica que implica la elaboración gradual de un sistema o proceso a través de múltiples iteraciones, donde se refinan y agregan detalles adicionales en cada iteración para mejorar el diseño y la funcionalidad del sistema y reducir el riesgo de errores y defectos.

- **Modularidad:** El software se divide en componentes con nombres distintos y abordables por separado, en ocasiones llamados “módulos”, que se integran para satisfacer los requerimientos del cliente.

- **Arquitectura del software:** Estructura general de un software y a las formas que ésta da integridad conceptual al sistema. En su forma más sencilla, la arquitectura es la estructura de organización de los componentes de un programa (módulos), la forma en la que estos interactúan y la estructura de datos que utilizan.
  3 propiedades:
    a) Estructurales;
    b) Extrafuncionales;
    c) Familias de sistemas relacionados.

- **Ocultamiento de la información:** Implica que la modularidad efectiva se logra definiendo un conjunto de módulos independientes que intercambien sólo, aquella información necesaria para lograr la función del software.
- **Independencia funcional:** Resultado directo de la separación de problemas y de los conceptos de abstracción y ocultamiento de la información.
  Debe diseñarse software de manera que cada módulo resuelva un subconjunto específico de requerimientos y tenga una interfaz sencilla cuando se vea desde otras partes de la estructura del programa.
  La independencia funcional se evalúa con dos criterios cualitativos: cohesión y acoplamiento.

    - **Cohesión:** se refiere a la medida en que los elementos dentro de un módulo de software están relacionados entre sí y trabajan juntos para lograr una tarea común. Indica la fortaleza relativa funcional del módulo. Un módulo cohesivo ejecuta una sola tarea, por lo que requiere interactuar poco con otros componentes en otras partes del programa. Tiene distintos niveles:
        a) Funcional: Cuando un componente realiza un cálculo y luego devuelve un resultado
        b) De capa: Cuando una capa más alta accede a los servicios de otra más baja pero no a la inversa (Herencia de Clases)
        c) De comunicación: Todas las operaciones que acceden a los mismos datos
        d) Secuencial: Un módulo contiene acciones que han de realizarse en un orden particular sobre unos datos concretos.
        e) Procedural: Un módulo contiene operaciones que se realizan en un orden concreto aunque sean independientes.
        f) Lógica: Cuando un módulo contiene operaciones cuya ejecución depende de un parámetro, el flujo de control del módulo es lo único que une a las operaciones que lo forman.
        g) Casual: Es el peor tipo de cohesión. Los elementos no tienen ninguna relación conceptual ni de ningún otro tipo de los que hemos visto hasta ahora, tan solo están agrupados en la misma unidad de software.
        h) Temporal: Las operaciones se incluyen en un módulo porque han de realizarse al mismo tiempo; p.ej. inicialización

    - **Acoplamiento:** Independencia relativa entre módulos. Es un indicador de la interconexión entre módulos en una estructura de software y depende de:
        a) Complejidad de la interfaz entre módulos;
        b) Grado en el que se entra o se hace referencia a un módulo;
        c) Datos que pasan a través de la interfaz.

      El acoplamiento tiene distintos niveles:
        1) De contenido: Cuando un componente modifica datos internos en otros componentes.
        2) Común: Cierto número de componentes hacen uso de una misma variable global.
        3) Control: Cuando una operación **A()**, invoca a la operación **B()**, y pasa una bandera de control a B. La bandera entonces “dirige” el flujo de la lógica dentro de B.
        4) Molde: Cuando se declara a clase B como un tipo para un argumento de una operación de clase A (composición).
        5) Datos: Cuando las operaciones pasan cadenas largas de argumentos de datos.
        6) Rutina de llamada: Cuando una operación invoca a otra.
        7) Tipo de uso: Cuando el componente A usa un tipo de datos definidos en el componente B.
        8) Inclusión o importación: Cuando el componente A importa o incluye un paquete o el contenido del componente B.

9) Externo: Cuando un componente se comunica o colabora con componentes de infraestructura.

- **Jerarquía de control:** Consiste en subdividir el problema vertical y horizontalmente.
  Se comienza por descomponer el problema inicial en 4 o 5 módulos que cubren la totalidad de la solución, y posteriormente y en caso de ser necesario, estos se descomponen para especificar más detalles.
  a) Descomposición horizontal: La solución del problema se divide en varios subproblemas.
  b) Descomposición vertical: Se incrementan los detalles.

Diseño Arquitectónico

Proceso en el que se establece la *arquitectura del software*, es decir, la estructura y organización del sistema. Se identifican los componentes y su interconexión, se definen las interfaces, se establece el flujo de datos y se diseñan los algoritmos. Se intenta establecer una organizacion del sistema que sastifaga los requerimientos funcionales y no funcionales del propio sistema

En definitiva, define los módulos que componen al sistema y cómo estos interactúan y se relacionan entre sí.

Flujo de transformación: Cuando un nodo se ramifica o se conecta con varios nodos distintos.(Ej: encadenamiento de llamadas).

Flujo de transacción: Cuando un nodo se ramifica en diferentes secciones dependiendo ciertos valores.(ej Switch)

Proceso de diseño

El “proceso de diseño” va de una visión “panorámica” del software a otra más cercana que define el detalle requerido para implementar un sistema. El proceso comienza por:

- Centramiento de la arquitectura
- Definición de subsistemas
- Establecimiento de mecanismos de comunicación entre subsistemas
- Identificación de componentes
- Descripción detallada de cada componente
- Diseño de interfaces externas, internas y de usuario

“En definitiva es el proceso de aplicar distintas técnicas y principios con el propósito de definir un dispositivo, un proceso o un sistema con suficiente detalle como para permitir su realización física.”

Nota: Diseño alto nivel y detallado

Diseño alto nivel: Es el diseño Arquitectónico

Diseño Detallado

El diseño detallado de software implica una descripción detallada de cómo funcionará el software y cómo interactuarán sus componentes. En esta etapa se definen los detalles técnicos, se seleccionan los lenguajes de programación, se determinan las estructuras de datos y se establecen los procedimientos para la validación y verificación de los requisitos.
El diseño detallado de software se lleva a cabo siguiendo los siguientes pasos:

- Especificación de requisitos: Se establecen los requisitos detallados del software en términos de sus características y funcionalidades.
- Diseño de la arquitectura: Se establece la estructura general del sistema, se identifican los componentes principales y se definen las interfaces entre ellos.
- Diseño de los componentes: Se detalla la funcionalidad de cada componente, se establece su relación con otros componentes y se define cómo interactúa con ellos.
- Diseño de los algoritmos y estructuras de datos: Se definen los algoritmos que se utilizarán para implementar cada componente y se establecen las estructuras de datos necesarias.
- Validación y verificación: Se establecen los procedimientos de validación y verificación para asegurarse de que el diseño cumpla con los requisitos especificados.

El diseño detallado de software es una etapa crítica del proceso de desarrollo de software ya que cualquier error en esta etapa puede causar problemas en las etapas posteriores del proceso. Un diseño detallado preciso y bien documentado puede facilitar la implementación, el mantenimiento y la modificación del software a lo largo de su ciclo de vida.

Describe a detalle cada uno de los datos que comparten los módulos de la solución:

1. Detalle de datos:
    a. Entradas
    b. Salidas
    c. BD
2. Detalle de las interfaces:
    a. Del sistema con el usuario
    b. Entre los componentes del sistema
3. Detalle de los módulos:
    a. Definición de secuencias y condiciones
    b. Definición de clases y relaciones

**Arquitectónico: De la arquitectura o relacionado con ella*.

Modelos de diseño

- **Diseño de datos:** Transforma en modelo del dominio de la información el análisis en las estructuras de datos necesarias para la implementación
- **Diseño arquitectónico:** Estructura modular del programa/aplicación.
- **Diseño de interfaz:** Interfaces del software con otros sistemas y con los usuarios
- **Diseño procedimental:** Descripción procedimental de los componentes del sistema

Diseño Estructurado

El diseño estructurado de software es una técnica de diseño que se enfoca en la organización lógica del software a través de la estructuración de sus componentes en módulos o bloques lógicos, siguiendo principios de modularidad y cohesión. Este diseño se basa en el principio de descomposición jerárquica, que consiste en dividir el software en módulos que tienen una tarea específica y se organizan de manera jerárquica. Los módulos se diseñan para interactuar con otros módulos de manera predecible, siguiendo principios como la alta cohesión, el bajo acoplamiento y la modularidad jerárquica. El diseño estructurado de software tiene beneficios como la facilidad para entender y modificar el software, la modularidad y la reutilización de los módulos. Sin embargo, puede ser complicado si se utiliza en proyectos grandes y complejos.

La información fluye a través de las estructuras y de llamadas a funciones. La modularización se lleva a cabo por medio de la descomposición jerárquica del problema. En resumen:

- El sistema software es una jerarquía de módulos, con un módulo principal y una función de controlador
- El módulo principal transfiere el control a los módulos inmediatamente derivados de nodos que estos pueden ejecutar haya completado su tarea, devolverá nuevamente el control al módulo controlador
- La descomposición de un módulo en submódulos continuará hasta que se llegue a un punto en el que el módulo resultante tenga solo una tarea específica que ejecutar

EXTRA

Acoplamiento

En el contexto del diseño de software, el acoplamiento se refiere al grado de interdependencia que existe entre dos o más componentes de un sistema de software. En otras palabras, el acoplamiento se refiere a la medida en que los diferentes módulos o componentes de un sistema están interconectados y dependen unos de otros para su correcto funcionamiento.

- Un acoplamiento alto significa que los componentes están muy interconectados, lo que significa que un cambio en uno de ellos puede tener un impacto significativo en los otros. Por otro lado, un acoplamiento bajo significa que los componentes están relativamente independientes entre sí y un cambio en uno de ellos tendrá poco o ningún impacto en los demás.
- Un acoplamiento bajo se considera deseable porque permite que los componentes se puedan desarrollar y mantener de manera independiente. Esto a su vez mejora la flexibilidad y la escalabilidad del sistema, ya que permite la modificación o reemplazo de componentes sin afectar la funcionalidad del sistema en su conjunto.

Por otro lado, un acoplamiento alto puede dificultar el mantenimiento y la evolución del sistema, ya que cualquier cambio en un componente puede requerir cambios significativos en otros componentes. Por lo tanto, es importante que los diseñadores de software se esfuercen por reducir el acoplamiento en los sistemas de software siempre que sea posible.

Unidad VII: Prueba de Software

El objetivo principal de las pruebas es encontrar errores durante el proceso, a fin de que no se conviertan en defectos después de liberar el software. El beneficio de estas revisiones es el descubrimiento temprano de los errores de modo que no se propaguen a la siguiente etapa del proceso software.
En resumen, es el proceso de ejecutar un programa con el fin de encontrar errores.

Diferencia entre error, defecto y falla
Error: Es una acción incorrecta de un ser humano que produce un defecto en un artefacto.
Defecto: Es un elemento de un artefacto que es inconsistente con sus requerimientos y que eventualmente producirá fallas.
Falla: Discrepancia entre el comportamiento esperado y el comportamiento observado del software creado.

Objetivos:

- **Para demostrar al desarrollador y al cliente que el software satisface sus requerimientos.** Esto conduce a las pruebas de validación.
- **Para descubrir defectos en el software en que el comportamiento de este es incorrecto, no deseable o no cumple su especificación.** Esto conduce a la prueba de defecto, en los que los casos de prueba se diseñan para exponer los defectos.

"Las pruebas solo pueden demostrar la presencia de errores, no su ausencia" Dijkstra

Verificacion y validacion
Verificación: Se comprueba que el sistema cumple con su especificación de requerimientos. ¿Construimos el producto correctamente?
Validación: Se determina si un componente del sistema cumple con las condiciones que se impusieron para ese componente al inicio de la fase. ¿Construimos el producto correcto?
En definitiva, la verificación comprueba los requerimientos y la validación que satisfaga las expectativas del cliente.

Principios:

- El software siempre se testea.
- Un buen caso de testeo es el que muestra que el programa no anda, no que funciona correctamente.
- Nada es más difícil que saber cuándo dejar de testear.
- Uno no puede auto-testearse.
- Parte ineludible del test es determinar cuál es el resultado esperado.
- El test debe ser reproducible.
- Casos para condiciones válidas e inválidas.
- Inspeccionar todos los resultados de lo testeado.
- A medida que avanza el testeo crece la probabilidad de NO encontrar errores.
- El mejor programador debe realizar el testeo de caja blanca.
- El mejor usuario debe realizar el testeo de caja negra.
- El testeo es parte del proceso de desarrollo.
- Nunca se modifica un programa a efectos de ser más fácilmente testeado.

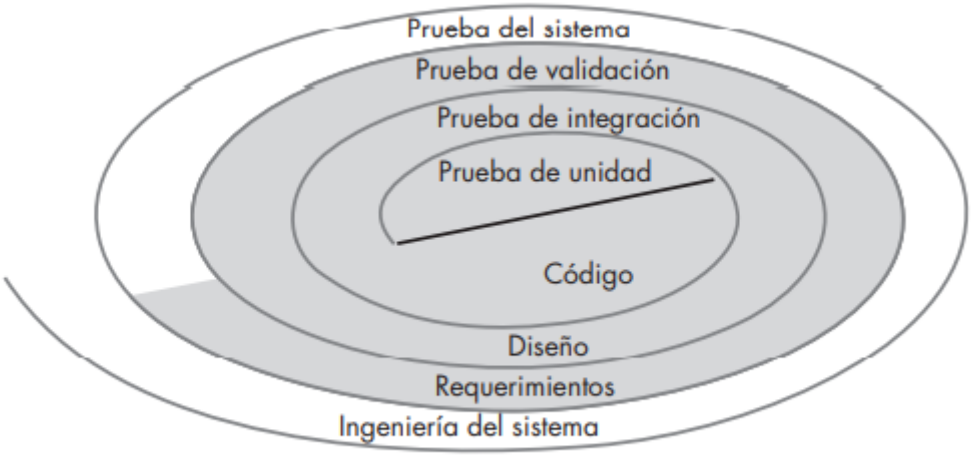
Criterios
¿Quien prueba? El desarrollador de software siempre es responsable de probar las unidades individuales del programa y de asegurarse de que cada una desempeña la función o muestra el comportamiento para el cual se diseñó. En muchos casos, el desarrollador también realiza pruebas de integración, una etapa en las pruebas que conduce a la construcción de la arquitectura completa del software, solo después de que la arquitectura de software está completa se involucra un grupo de pruebas independientes (GPI).
El desarrollador y el GPI trabajan conjuntamente.
¿Cuando se termina de testear? Nunca, los testeos continúan por parte del usuario.

Estrategia de prueba

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.

Al considerar el proceso desde un punto de vista procedural, las pruebas dentro del contexto de la ingeniería del software en realidad son una serie de 4 pasos que se implementan de manera secuencial:

- Prueba de unidad; (modulos)
- Prueba de integración; (conjunto de módulos)
- Prueba de Validación; (sistema completo)
- Prueba de Sistema.(sistema ya montado y se prueba todo el hardware y software)
- Adicionalmente: La prueba de aceptación. (Por parte del cliente).

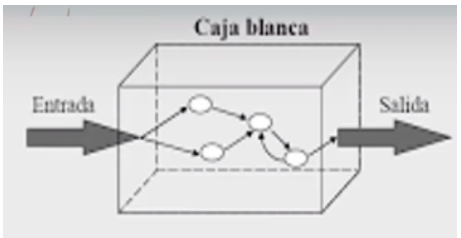


Planeamiento

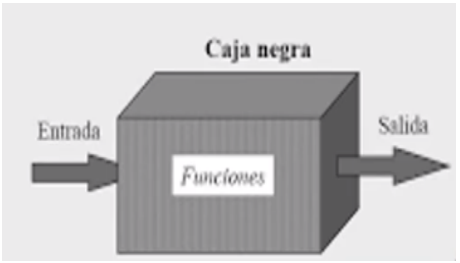
Se selecciona una característica del sistema o componente que se está probando. A continuación, se selecciona un conjunto de entradas que ejecutan dicha características, documenta las salidas esperadas o rangos de salida y, donde sea posible, se diseña una prueba automatizada.

Prueba de caja blanca y caja negra:

- **Caja blanca**: 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**.**

  El diseño de casos debe basarse en la elección de caminos importantes que ofrezcan una seguridad aceptable en descubrir defectos, y para ello se utilizan los llamados criterios de cobertura lógica.
  a) Cobertura de sentencias;
  b) Cobertura de decisiones;
  c) Cobertura de condiciones.
- **Caja negra:**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.

Tipos de prueba

- **Prueba de unidad:** 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:** 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. Observar que efecto onda produce.
- **Prueba de humo (Ad Hoc):** 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.
- **Prueba de estrés:** Las pruebas de estrés 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.
- **Prueba de volumen:** 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.
- **Prueba de carga:** 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).

Automatización de las pruebas

Cada prueba individual se implementa como un objeto y un ejecutor de pruebas ejecuta todas las pruebas. Las pruebas en sí mismas deben escribirse de forma que indiquen si el sistema probado funciona como se esperaba.

Unidad VIII: Implementación y Mantenimiento

Implementación

Empezamos a codificar los algoritmos y estructuras definidos durante el proceso de diseño .
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.

Mantenimiento

Es el conjunto de actividades que se llevan a cabo para hacer corrección de errores, mejoras de las capacidades, eliminación de funciones obsoletas y optimización en un sistema de software cuando ya está en operación. El propósito es preservar el valor del software sobre el tiempo.
El mantenimiento del software es una fase posterior al desarrollo y liberación del producto software y está encaminada principalmente a:

- **Incorporar nueva funcionalidad al producto software.**
- **Mejorar la funcionalidad presente en el producto software.**
- **Mejorar el rendimiento del producto software.**
- **Corregir defectos no detectados durante la fase de pruebas del software.**

¿Por qué es importante el mantenimiento del software?

Es importante ya que los desarrolladores no pueden darse el lujo de lanzar un producto y dejar que se ejecute, deben estar constantemente atentos a corregir y mejorar su software para seguir siendo competitivos y relevantes.

El uso de las técnicas y estrategias correctas de mantenimiento de software es una parte fundamental para mantener cualquier software en ejecución durante un largo período de tiempo y mantener contentos a los clientes y usuarios.

Además sin mantenimiento, cualquier software será obsoleto y esencialmente inútil con el tiempo.

Tipos de mantenimiento

Mantenimiento correctivo de software

El mantenimiento correctivo del software es la forma clásica y típica de mantenimiento (para el software y cualquier otra cosa). Es necesario cuando algo sale mal en una pieza de software, incluidos fallos y errores. Estos pueden tener un impacto generalizado en la funcionalidad del software en general y, por lo tanto, deben abordarse lo antes posible. Reconocer y solucionar las fallas antes de que los usuarios las descubran, es una ventaja adicional que hará que la empresa se vea confiable.

Mantenimiento preventivo de software

El mantenimiento preventivo de software está mirando hacia el futuro para que el software pueda seguir funcionando como se desee durante el mayor tiempo posible.
Esto incluye realizar los cambios necesarios, actualizaciones, adaptaciones y más. El mantenimiento preventivo del software puede abordar pequeños problemas que en un momento dado pueden carecer de importancia, pero pueden convertirse en problemas mayores en el futuro. Estos se denominan fallas latentes que deben detectarse y corregirse para asegurarse de que no se conviertan en fallas efectivas.

Mantenimiento perfectivo de software

El mantenimiento perfectivo de software tiene como objetivo ajustar el software agregando nuevas características según sea necesario y eliminando características que son irrelevantes o no efectivas en el software dado. Este proceso mantiene el software relevante a medida que el mercado y las necesidades del usuario cambian.

Mantenimiento adaptativo de software

Si es necesario cambiar el entorno en el que se utiliza la aplicación (que incluye el sistema operativo, la plataforma de hardware o, en el caso de las aplicaciones web, el navegador), puede ser necesario modificarla para mantener su plena funcionalidad en estas nuevas condiciones.

Mantenimiento evolutivo de software

Es un caso especial donde la adaptación es prácticamente obligatoria, ya que de lo contrario el programa quedaría obsoleto con el paso del tiempo.

Proceso de mantenimiento de software

El proceso de mantenimiento de software implica varias técnicas de mantenimiento de software que pueden cambiar según el tipo de mantenimiento y el plan de mantenimiento de software implementado.

Un sistema de software desarrollado atendiendo a los principios de reusabilidad y extensibilidad, garantizará un proceso de mantenimiento flexible. Comúnmente, el mantenimiento del software está orientado a la incorporación de mejoras a la funcionalidad del sistema software, y no tanto a la corrección de errores.

La mayoría de los modelos de procesos de mantenimiento de software incluyen los siguientes pasos:

1. **Identificación y rastreo**: el proceso de determinar qué parte del software necesita ser modificada (o mantenida). Esto puede ser generado por el usuario o identificado por el propio desarrollador de software según la situación y el fallo específico.
2. **Análisis**: el proceso de analizar la modificación sugerida, incluida la comprensión de los efectos potenciales de dicho cambio. Este paso generalmente incluye un análisis de costos para comprender si el cambio vale la pena desde el punto de vista financiero.
3. **Diseño**: diseño de los nuevos cambios utilizando especificaciones de requisitos.
4. **Implementación**: el proceso de implementación de los nuevos módulos por parte de los programadores.
5. **Prueba del sistema**: antes de iniciarlo, se debe poner a prueba el software y el sistema. Esto incluye el módulo en sí, el sistema y el módulo, y todo el sistema a la vez.

6. **Prueba de aceptación**: los usuarios ponen a prueba la modificación para su aceptación. Este es un paso importante ya que los usuarios pueden identificar problemas en curso y generar recomendaciones para una implementación y cambios más efectivos.
7. **Entrega**: actualizaciones de software o, en algunos casos, nueva instalación del software. Es cuando los cambios llegan a los clientes.

Estrategias para el mantenimiento del software

- **La documentación** es una estrategia importante en el desarrollo de software. Si la documentación del software no está actualizada, escalar puede resultar aparentemente imposible. La documentación debe incluir información sobre cómo funciona el código, soluciones a problemas potenciales, etc.

- **El control de calidad** también es una parte importante de un plan de mantenimiento de software. Si bien el control de calidad es importante antes del lanzamiento inicial del software, también se puede integrar mucho antes en el proceso (ya en la etapa de planificación) para asegurarse de que el software se desarrolle correctamente y para brindar información sobre cómo realizar cambios cuando sea necesario.

- **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.

REINGENIERÍA

La reingeniería de software es el proceso de modificar y mejorar un software existente para adaptarlo a las necesidades actuales y futuras del negocio, así como para hacerlo más fácil de mantener y actualizar. Este proceso implica una revisión completa del software, desde su diseño hasta su implementación, para identificar y corregir problemas y deficiencias.

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

La reingeniería de software se puede llevar a cabo por diversas razones, como la necesidad de actualizar el software para que funcione en nuevas plataformas o sistemas operativos, la necesidad de mejorar el rendimiento o la escalabilidad del software, o la necesidad de agregar nuevas funcionalidades.

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.
- Re-documentación, anotación, actualización de la información.
- Re-modularización.
- 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.

La ingeniería directa de software es un enfoque de desarrollo de software que implica la creación de un diseño detallado antes de escribir el código. En este enfoque, se sigue un proceso de desarrollo secuencial, en el que cada fase se completa antes de pasar a la siguiente.

INGENIERÍA INVERSA

Proceso mediante el cual se analiza el código fuente de un software existente para comprender su estructura, su funcionalidad y su comportamiento. Este proceso implica la extracción de información a partir del código fuente y su uso para crear modelos y diagramas que describan el software en términos de su arquitectura, componentes y relaciones, con el objetivo de crear representaciones del sistema en otra forma o a un nivel más alto de abstracción.

El proceso de ingeniería inversa de software generalmente se lleva a cabo por las siguientes razones:

- **Comprender el software existente**: El análisis del código fuente puede ayudar a entender cómo funciona el software existente, lo que puede ser útil para la integración con otros sistemas, la actualización o la mejora del software.
- **Documentar el software**: La ingeniería inversa de software también se puede utilizar para crear documentación detallada sobre el software, lo que puede ayudar a la comprensión y el mantenimiento del software.
- **Reutilización del código**: La ingeniería inversa de software también se puede utilizar para extraer el código fuente de un software existente y reutilizarlo en otros proyectos.

El proceso de ingeniería inversa de software consta de los siguientes pasos:

1. **Análisis del código fuente**: Se analiza el código fuente del software existente utilizando herramientas de análisis de software.
2. **Creación de modelos y diagramas**: A partir de la información obtenida, se crean modelos y diagramas que describen la arquitectura, componentes y relaciones del software.
3. **Validación de modelos**: Los modelos y diagramas creados se validan para asegurarse de que sean precisos y que representen correctamente el software existente.
4. **Documentación**: Se documenta el software utilizando los modelos y diagramas creados.

La ingeniería inversa de software puede ser un proceso valioso para comprender y documentar el software existente. Sin embargo, también puede ser un proceso costoso y requiere una comprensión profunda de la estructura y el comportamiento del software.

¿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

Costo de mantenimiento de software

El costo puede resultar elevado. Sin embargo, esto no niega la importancia del mantenimiento del software. En ciertos casos, el mantenimiento del software puede costar hasta dos tercios de todo el ciclo del proceso del software o más del 50 % de los procesos ciclo de vida del desarrollo de software.

Los costos involucrados en el mantenimiento del software se deben a múltiples factores y varían según la situación específica. Cuanto más antiguo sea el software, mayor será el costo de mantenimiento, ya que las tecnologías (y los lenguajes de codificación) cambian con el tiempo. Renovar un software antiguo para adaptarlo a la tecnología actual puede ser un proceso excepcionalmente caro en determinadas situaciones.

Además, es posible que los ingenieros no siempre puedan enfocarse en los problemas exactos cuando buscan actualizar o mantener un software específico. Esto hace que utilicen un método de prueba y error, que puede resultar en muchas horas de trabajo.

Hay ciertas formas de intentar reducir los costos de mantenimiento del software. Estas incluyen la optimización de la parte superior de la programación utilizada en el software, la escritura fuerte y la programación funcional.