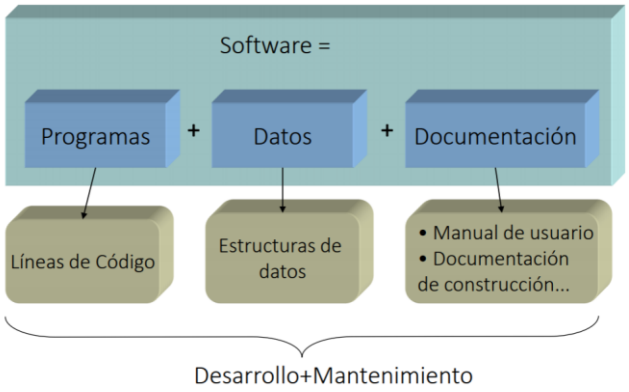
# RESUMEN INGS

## Parcial 1

## Unidad 1: Introducción a la Ingeniería del Software




Objetivos:

Reducir el coste y mejorar la calidad del software, explotar y aprovechar el potencial proporcionado por el hardware, desarrollar y mantener software asegurando calidad, fiabilidad, facilidad de uso, imposibilidad de mal uso.

Actividades de Desarrollo:

- ❑ Decidir qué hacer (Análisis)
- ❑ Decidir cómo hacerlo (Diseño)
- ❑ Hacerlo (Codificación)
- ❑ Probar el producto (Pruebas)
- ❑ Usar el producto (Entrega/Instalación)
- ❑ Mantener el producto (Mantenimiento)

Actividades de Control: se ocupan de evaluar y asegurar la calidad del software.

- ❑ Métricas
- ❑ Aseguramiento de la calidad
- ❑ Gestión de configuraciones

Actividades de Gestión:

- ❑ Planificación y estimación
- ❑ Seguimiento de los proyectos
- ❑ Administración de proyectos
- ❑ Dirección de proyectos

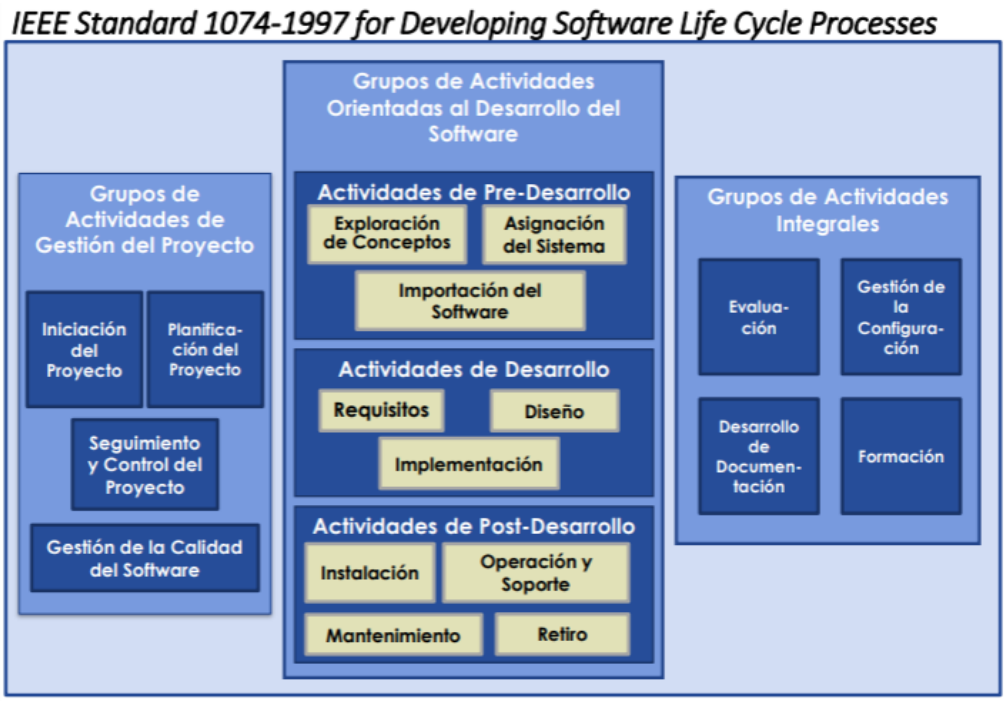
Actividades de Operación:

- ❑ Entrega (e instalación)
- ❑ Puesta en marcha
- ❑ Formación a los usuarios

Producto: rentable, fiable, de calidad, eficiente, usable, que cumpla con los requisitos, etc.

Proceso: con éxito, eficiente, de calidad, controlado, etc.

# Unidad 2: Metodologías, Ciclos de Vida y Proceso Software




Metodología: El conjunto de métodos que se utilizan en una determinada actividad con el fin de formalizarla y optimizarla. Determina los pasos a seguir y cómo realizarlos para finalizar una tarea. Define el qué, el cómo y el cuándo además de proporcionar mecanismos para determinar la consecución de objetivos.

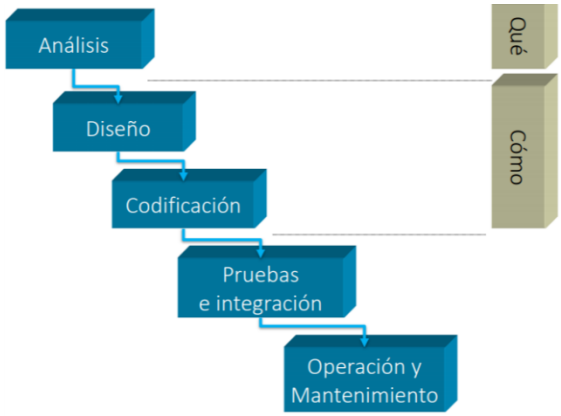
- Tradicionales: Fases bien definidas, entregas al final, requisitos y planificación bien definidos.
- Ágiles: Continua interacción con el cliente, muchas entregas parciales y ciclo iterativos más cortos.
- Modelos centrados en el usuario: Enfoque e interacción continua con el usuario. Usabilidad y Accesibilidad como características de calidad prioritarias.

Ciclo de vida: Conjunto de fases por las que pasa el sistema que se está desarrollando desde que nace la idea inicial hasta que el software es retirado o reemplazado. Determina el orden de las fases del proceso software definiendo las entradas y salidas de cada fase.

Proceso software: Es el conjunto de actividades que se realizan para la gestión, desarrollo, mantenimiento y soporte de los sistemas software. Incluye los actores que ejecutan las actividades, sus roles y los artefactos producidos.

1 Ciclos de vida de metodologías tradicionales:

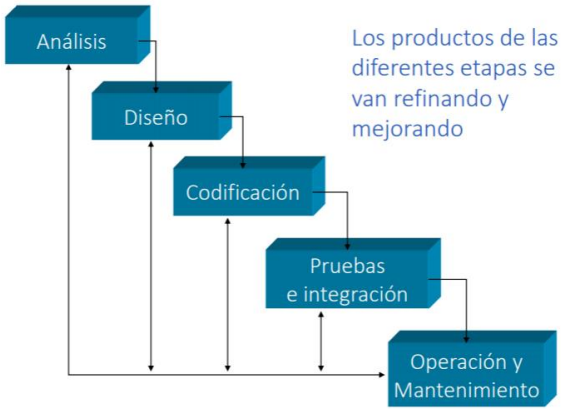
1.1 Modelo de cascada clásico:



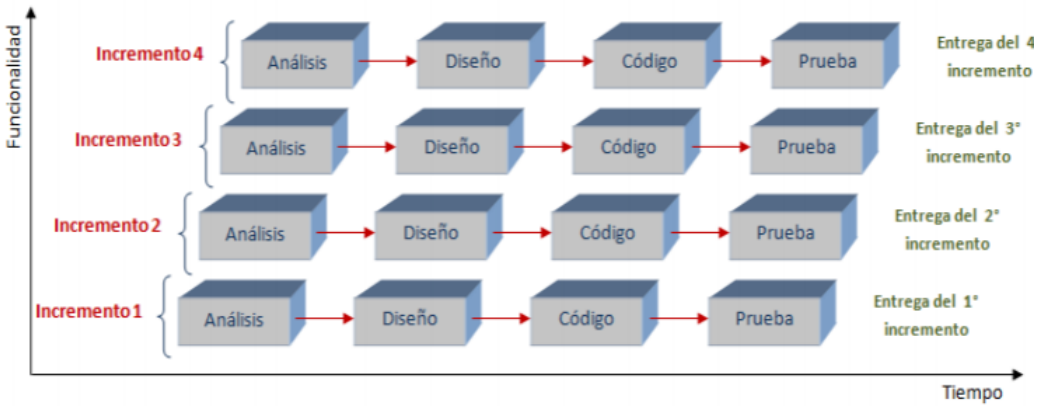

- Sucesión de etapas que producen productos intermedios. Para que el proyecto tenga éxito deben desarrollarse todas las fases.
- Las fases continúan hasta que los objetivos se han cumplido.
- Si se cambia el orden de las fases, el producto final será de inferior calidad

- No se permiten las iteraciones.
- Los requisitos se congelan al principio del proyecto.
- No existe un proyecto "enseñable" hasta el final del proyecto

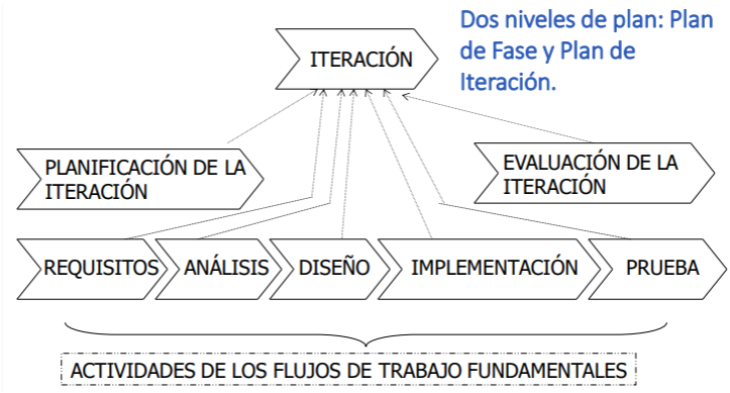
1.2 Modelo de refinamiento por pasos:




1.3 Modelo incremental iterativo:




- Se analiza primero el problema y se divide en subproblemas que se van desarrollando en cada iteración. La división de los requisitos que se implementarán en cada incremento se realiza al principio del proyecto.
- El producto software se desarrolla por partes. En cada incremento se agrega más funcionalidad al sistema. Se integran a medida que se completan.



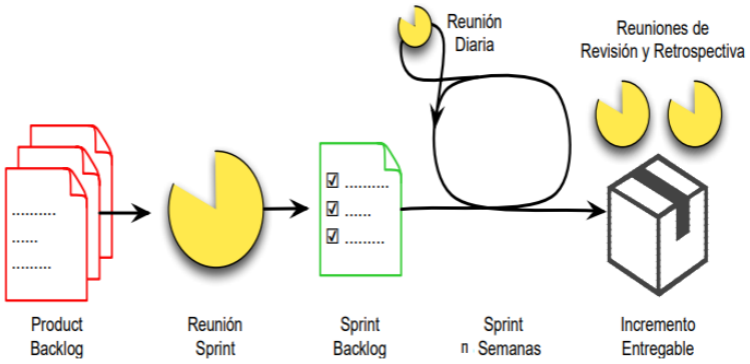

• Refinamiento por pasos, pero, normalmente, se iteran todas las fases cada vez.
• En cada ciclo, iteración, se revisa y mejora la calidad del producto.

- Permite construir una implementación parcial del sistema global y posteriormente ir aumentando la funcionalidad del sistema (incorporar nuevos requisitos).
- El producto provisional de cada desarrollo será usado en el entorno operativo. ν Produce un sistema operacional rápidamente.

2 Ciclos de vida de metodologías ágiles:

La máxima prioridad es la satisfacción del cliente a través de entregas continuas de evaluables. Los cambios en los requisitos son bienvenidos en cualquier fase del desarrollo. Las entregas de software son frecuentes, desde un par de semanas hasta un par de meses, con cierta preferencia por los plazos más cortos y son la principal medida de progreso. Interesados y desarrolladores deben trabajar juntos diariamente durante el proyecto. Los equipos, además de auto-organizarse, deben reflexionar a intervalos regulares y frecuentes sobre cómo ser más eficaces y mejorar la calidad técnica y deben ajustar su comportamiento en consecuencia.

2.1 SCRUM




Se basa en sprints: iteraciones de desarrollo cuya duración recomendada es 2-4 semanas. El sprint es por tanto el núcleo central que proporciona la base de desarrollo iterativo e incremental.

- Durante cada sprint, el equipo crea un incremento de software potencialmente entregable (utilizable).
- El equipo determina la cantidad de elementos del Product Backlog que puede comprometerse a completar durante el siguiente Sprint.

Se centra en entregables pequeños y concretos trabajados por orden de prioridad. Al finalizar cada sprint se entrega uno o varios productos funcionales completos.

Ventajas: Flexibilidad a cambios , Reducción del Time to Market, Optimiza el proceso, Alcance viable, Mayor productividad del equipo, Maximiza el retorno de la inversión (ROI), Incremental, Reducción de riesgos.

Desventajas: Equipo auto-organizado (puede ser ventaja pero tiene riesgos), Fuerte compromiso por parte del equipo, Fuerte compromiso por parte del cliente , Restricciones en el tamaño de los proyectos, Posible falta de documentación.

Usabilidad: Medida en la que un producto se puede usar por determinados usuarios para conseguir objetivos específicos con efectividad, eficiencia y satisfacción en un contexto de uso especificado. Se mide el número de errores cometidos por los sujetos de prueba y si estos fueron recuperables o no al usar los datos o procedimientos adecuados, el tiempo, y el recuerdo y respuesta emocional del usuario. Accesibilidad -> Caso específico de Usabilidad.

# Unidad 3: Actividades tempranas en el desarrollo del SW

## Actividades de Pre-Desarrollo.

- ❑ Estimación y planificación
- ❑ Definición del ciclo de vida.
- ❑ Definición de estándares, métodos, técnicas y herramientas.
- ❑ Definición de objetivos, entradas y salidas.
- ❑ Definición de criterios de calidad y validación.
- ❑ Definición de hitos. ✗
- ❑ Creación del equipo de desarrollo inicial.
- ❑ Definición de mecanismos de seguimiento y control.

* Definir productos correspondientes a cada hito. (Productos intermedios “enseñables”). Definir Qué, Quién, Cuándo y Cómo se va a evaluar. Se consigue un hito cuando se ha revisado la calidad de uno o más productos y se han aceptado. Tras cada hito, se debería generar un informe de progreso del proyecto. Coincidiendo con el final de una fase (al menos).

# Unidad 4: Análisis (¿Qué hay que hacer?)

Análisis de requisitos: Análisis del problema y especificación completa del comportamiento externo que se espera del sistema software que se va a construir, así como de los flujos de información y control.

- Requisitos de usuario: Declaraciones de las funciones o acciones que los distintos usuarios pueden realizar con la aplicación y bajo qué restricciones
- Requisitos software: Especifican de una manera completa, consistente y detallada qué debe hacer y cómo debe comportarse el software para cumplir

con los objetivos de la aplicación. Sirven de base a los desarrolladores para diseñar el sistema.

Tareas:

- Análisis del problema
- Análisis de Usuarios y de las Tareas
  - Identificar potenciales usuarios del sistema, su jerarquía y las tareas a realizar.
- Educción de requisitos.
  - Identificar los requisitos que se obtienen de los usuarios y clientes.
- Análisis de los requisitos.
  - Razonar sobre los requisitos educidos, combinar requisitos relacionados, establecer prioridades entre ellos, determinar su viabilidad, etc.
- Representación (modelización).
  - Registrar los requisitos de alguna forma, incluyendo lenguaje natural, lenguajes formales, modelos, prototipos, maquetas, UML, etc.
- Validación.
  - Examinar inconsistencias entre requisitos, determinar la corrección, ambigüedad, etc. Establecer criterios para asegurar que el software reúna los requisitos cuando se haya producido. El cliente, usuario y desarrollador se deben poner de acuerdo.

Requisitos:

- Requisitos funcionales.

  Acciones fundamentales que tienen que tener lugar en la ejecución del software.

  Son acciones elementales necesarias para el correcto comportamiento de nuestro sistema

  Ejemplo: "El usuario podrá dar de alta un elemento"

- Requisitos no funcionales.

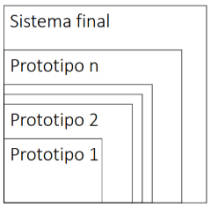
  Representan características o cualidades generales que se esperan del software para conseguir su propósito.

  - Requisitos de interfaz y usabilidad.
    - Menús, Ventanas, Mensajes de error, Formatos de Pantalla, etc.
  - Requisitos operacionales.
    - Modos de operación, back-ups, funciones de recuperación, etc.
  - Requisitos de documentación.
    - Idiomas, tipos de usuario, ayuda on-line, web, tutoriales, etc.
  - Requisitos de seguridad.
    - Diferentes niveles de acceso al sistema, protección, mantenimiento de históricos, claves, etc.
  - Requisitos de mantenibilidad y portabilidad.
    - Grado en que debe ser fácil cambiar el software o portarlo.
  - Requisitos de recursos.
    - Tanto hardware como software. Ej: limitaciones sobre memoria, almacenamiento.
  - Requisitos de rendimiento.
    - Tiempo de respuesta, nº de usuarios, terminales soportadas, consumo de memoria, etc.
  - Requisitos de comportamiento.
  - Requisitos de disponibilidad.
    - Especialmente para aquellos sistemas informáticos que necesiten estar conectados a la red.
  - Requisitos de soporte.
    - Incluyen la facilidad de instalación, facilidad de mantenimiento, facilidad de actualización y facilidad de portabilidad.
  - Requisitos de verificación y fiabilidad.
    - Recuperación ante situaciones anómalas o de error.
  - Requisitos legales.
    - Características que deben cumplir el sistema para cumplir con la legislación vigente.

- Los requisitos deben ser:
  - Completos.
    Todo lo que el software tiene que hacer está recogido en el conjunto de requisitos.
  - No ambiguos.
    Cada requisito debe tener una sola interpretación.
  - Relevantes.
    Importancia para el sistema software a implementar.
  - Traceables
    Cada acción de diseño debe corresponderse con algún requisito, y debe poder comprobarse.
  - Correctos.
    Cada requisito establecido debe representar algo requerido por el usuario para el sistema que se construye.
  - Consistentes.
    Ningún requisito puede estar en conflicto con otro. Tipos de inconsistencias:
    - Términos conflictivos: Si dos términos se usan en contextos diferentes para la misma cosa.
    - Características en conflicto: Si en dos partes de la ERS se pide que el producto muestre comportamientos contradictorios.
    - Inconsistencia temporal: Si dos partes de la ERS piden que el producto obedezca restricciones de tiempo contradictorias.

Modelos de desarrollo:

- Maquetas: Cuando los requisitos no están claros. Cuando el alcance del proyecto no está bien definido. Cuando los usuarios no se muestran colaboradores.
  Permiten adaptar el modelo mental del usuario con el del ingeniero del software, permitiendo una mejor educción de requisitos software; validar los requisitos del usuario; comprobar la aceptación del usuario.
- Prototipos: Cuando no se conoce la complejidad total del desarrollo. Cuando el ingeniero del software no tiene muy clara la solución informática más adecuada.




Permite verificar los requisitos principales del usuario. Verificar la viabilidad del diseño informático del sistema. Facilitar el diseño de la Interacción Persona-Ordenador y de las Interfaces de Usuario del sistema. El prototipo evoluciona iterativamente.

Un prototipo es un modelo que aún está en fase de experimentación y que se ha creado para probarlo, estudiarlo y mostrar su funcionamiento, mientras que una maqueta es una recreación a escala o (en mock-up) de algún objeto funcional ya existente.

Principales errores del análisis: Alto riesgo de mal entendimiento entre cliente/usuario e ingeniero del software (Requisitos ambiguos). ν Tendencia a acortar y minimizar la importancia de esta etapa. ν No establecer los criterios de aceptación del Software. ν Pobre estudio del problema: El ingeniero del software debe conocer bien el dominio del problema. ν No revisión de la especificación de requisitos por el cliente/usuario o el ingeniero del software. ν Permitir el continuo cambio de requisitos por parte del usuario.

Al final del análisis se generará un documento final: ERS que proporcionará los medios de comunicación entre todas las partes: clientes, usuarios, analistas y diseñadores, y ayudará al control de la evolución del sistema

# Unidad 5: Diseño (¿Cómo hay que hacerlo?)

Es el proceso de definición de la arquitectura, componentes, módulos, interfaces, procedimientos de prueba y datos de un sistema software para satisfacer unos requisitos especificados.

- Diseño de la arquitectura: definición de la colección de componentes del sistema y sus interfaces. Descomponer el sistema en módulos. Determinar las estructuras de datos. Diseñar las interfaces.
- Diseño detallado: descripción detallada de la lógica de cada uno de los módulos, de las estructuras de datos que utilizan y de los flujos de control. Descripción detallada de los módulos: estructuras y algoritmos.

Principios básicos: Abstracción: manejo de conceptos generales y no de instancias particulares. Refinamiento: seguir una estrategia de diseño descendente. Modularidad.

Métodos de diseño:

- Diseño estructurado: Se considera que el sistema es un conjunto de módulos o unidades, cada uno con una función bien definida, organizados de forma jerárquica.
- Diseño orientado a objetos: Se considera que el sistema es una colección de objetos que interactúan a través de mensajes. Cada objeto tiene su propio estado y conjunto de operaciones asociadas.

Métricas de diseño:

- Cohesión: medida de la relación (funcional) de los elementos de un módulo. Es mejor un alto grado de cohesión: Menor coste de programación y mayor calidad del producto.
- Acoplamiento: Es una medida de la interconexión entre los módulos de un programa. Es mejor un bajo acoplamiento: Minimizar el efecto onda (propagación de errores), minimizar el riesgo al cambiar un módulo por otro, facilitar el entendimiento.

Buen diseño: Estructura flexible, extensible, portable y reusable.

Principales errores del diseño: Falta de detalle en las especificaciones de diseño. ν No documentar el diseño. ν No tener en cuenta el entorno físico.

Al final del análisis se generará un documento final.

# Unidad 6: Codificación, Pruebas, Entrega y Finalización

Codificación: Traducir las especificaciones de diseño en un lenguaje de programación.

- Salidas: Programas. Documento de diseño modificado. Manual técnico. Manual de usuario.

Prueba: Proceso de ejecutar un software con el fin de encontrar errores.

Definiciones incorrectas: Probar es demostrar que no hay errores en el programa. Probar es mostrar que el programa funciona correctamente.

El objetivo es descubrir los errores cometidos durante las fases anteriores de desarrollo de producto, no demostrar que el programa funciona. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces. La prueba tiene éxito si descubre un error no detectado hasta entonces. Tipos:

- Verificación: ¿Se ha construido el sistema correctamente? Comprueba el funcionamiento del software, que implemente correctamente una función específica.
- Validación: ¿Se ha construido el sistema correcto? Comprueba si los requisitos de usuario se cumplen y los resultados obtenidos son los previstos.

Los casos de prueba deben ser escritos tanto para condiciones de entrada válidas y esperadas como para condiciones inválidas e inesperadas. Examinar un programa para comprobar si hace o no lo que se supone que debe hacer es sólo la mitad, también hay que ver si hace o no lo que se supone que no debe hacer.

Se deben documentar los casos de prueba, esto permite re-ejecutarlos en el futuro (pruebas de regresión).

Pruebas en las diferentes fases:

- Análisis de requisitos: ¿Existe algún requisito no identificado? θ ¿Hay requisitos redundantes o contradictorios?
- Diseño. ¿Abarca el diseño todos los requisitos descritos en la especificación? ¿Es lo más simple posible la solución elegida?
- Codificación. Inspecciones de código. Suelen realizarse por un equipo que incluye personas ajenas al proyecto, participantes en otras fases del proyecto y personal del grupo de calidad.

Técnicas:

- **Pruebas de caja negra**: Conducida por los datos de Entrada/Salida. Considera el software como una caja negra sin tener en cuenta los detalles procedimentales de los programas. Los datos de entrada deben generar una salida en concordancia con las especificaciones.

- Partición de equivalencia: dividir el dominio de entrada de un programa en un número finito de clases de equivalencia de las que se puedan derivar casos de prueba con el objetivo de reducirlos. Pasos:
  1. Identificar las clases de equivalencia: Conjunto de datos que definen entradas Válidas y No Válidas al sistema.
  2. Identificar los casos de prueba.
- Análisis de Valores Límite (AVL). Seleccionar casos de prueba que ejerciten los valores límite de cada clase de equivalencia.

- Si la entrada es un rango, se define una clase de equivalencia válida y dos inválidas.
  - Ej. Un contador va de 01 a 59

| Clases válidas | Clases no válidas |
|---|---|
| [01, 59] | n < 1<br>n > 59 |

- Si la entrada es un valor específico, se define una clase de equivalencia válida y dos inválidas.
  - Ej. Nº propietarios de un coche

| Clases válidas | Clases no válidas |
|---|---|
| 1 <= nº prop. <= 5 | n = 0<br>n > 5 |

- Si la entrada en un miembro de un conjunto, se define una clase de equivalencia válida y otra inválida.
  - Ej. Días de la semana

| Clases válidas | Clases no válidas |
|---|---|
| {L,M,X,J,V,S,D} | ≠ {L,M,X,J,V,S,D} |

- Se la entrada es un valor lógico, se define una clase de equivalencia válida y otra inválida.
  - Ej. ID que empieza por letra

| Clases válidas | Clases no válidas |
|---|---|
| 1er dígito ID € [A, Z] | 1er dígito ID ≠ [A, Z] |

- Los siguientes son los datos de entrada para un formulario:
  - Edad es un número positivo entre 18 y 65.
  - Nombre es un campo alfanumérico de 25 caracteres exactamente.
  - Día libre de la semana es un campo que puede tomar los valores (lunes, martes, miércoles, jueves, viernes, sábado, domingo).
  - Antigüedad en la empresa en meses es un campo de dos dígitos que puede tomar el valor 0.
- El programa asigna 4 ayudas diferentes en función de los días libres y de la antigüedad en la empresa de la siguiente forma:
  - A1: Lunes y personas con antigüedad inferior a 1 año.
  - A2: Martes o miércoles y personas con antigüedad inferior a 3 años.
  - A3: Jueves o viernes y personas con antigüedad inferior a 8 años.
  - A4: Sábado o domingo y personas con antigüedad superior o igual a 8 años.
  - Error

| Atributo | Válida | Inválida |
|---|---|---|
| Edad | 1. [18, 65] | 2. < 18<br>3. > 65<br>4. No número |
| Nombre | 5. Alfanumérico 25 caracteres | 6. < 25 caracteres<br>7. > 25 caracteres<br>8. Algún carácter no alfanumérico |
| Día Libre | 9. {Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo} | 10. Cualquier otra cadena |
| Antigüedad | 11. [0, 99] | 12. > 99 caracteres<br>13. No número |
| Salida | 14. {A1, A2, A3, A4, Error} | 15. Cualquier otra salida |

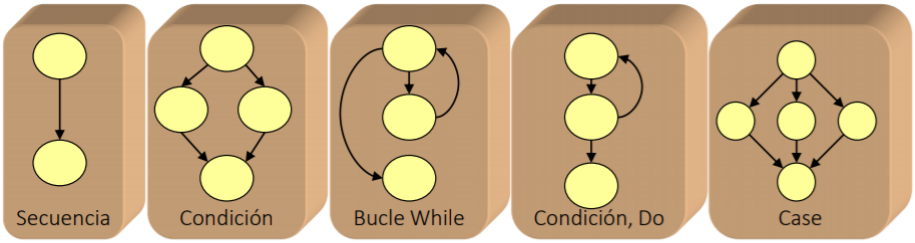
| Edad | Nombre | Día Libre | Antigüedad | Salida |
|---|---|---|---|---|
| 25 | "Antonio …" | Lunes | 10 | A1 |
| 25 | "Antonio …" | Martes | 16 | A2 |
| … | … | … | … | … |
| 18 (LI) | "Antonio …" | Martes | 10 | A1 |
| 17 (LI-1) | "Antonio …" | Martes | 10 | Error |

# Examen Final

- **Pruebas de caja blanca**: Probar el comportamiento interno y la estructura del programa examinando la lógica interna intentando provocar situaciones extremas.
  - Se ejecutan todas las sentencias (al menos una vez)
  - Se recorren todos los caminos independientes de cada módulo.
  - Se comprueban todas las decisiones lógicas.
  - Se comprueban todos los bucles.

  *Técnicas:*

  - Pruebas de interfaz: Analiza el flujo de datos que pasa a través de la interfaz (tanto interna como externa): argumentos de funciones, consistencia y manejo de ficheros correcta, etc.
  - Prueba de estructuras de datos locales: Aseguran la integridad de los datos durante todos los pasos de la ejecución del módulo: Referencias, declaración, cálculos.
  - Prueba del camino básico: Definen un conjunto básico de caminos de ejecución usando la complejidad ciclomática (grafos de flujo).




  - Número de caminos básicos a probar:

$$V(G)=Aristas-Nodos+2$$
$$V(G)=Nº\ regiones\ cerradas+1$$

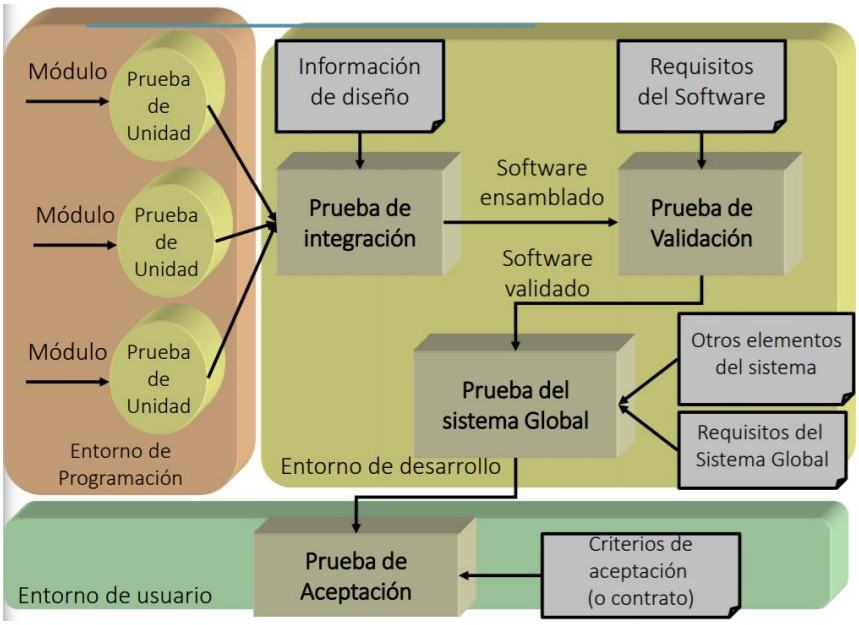
  Un módulo con complejidad ciclomática mayor de 10 debe ser examinado para posibles simplificaciones o partirlo en varias subrutinas.
  Ejemplo diapo 36

  - Pruebas de condición: de condición simple incluye variables lógicas True/False y expresiones relacionales >,>=, etc. De condición compuesta incluye condiciones simples y operadores lógicos AND, NOT, etc.

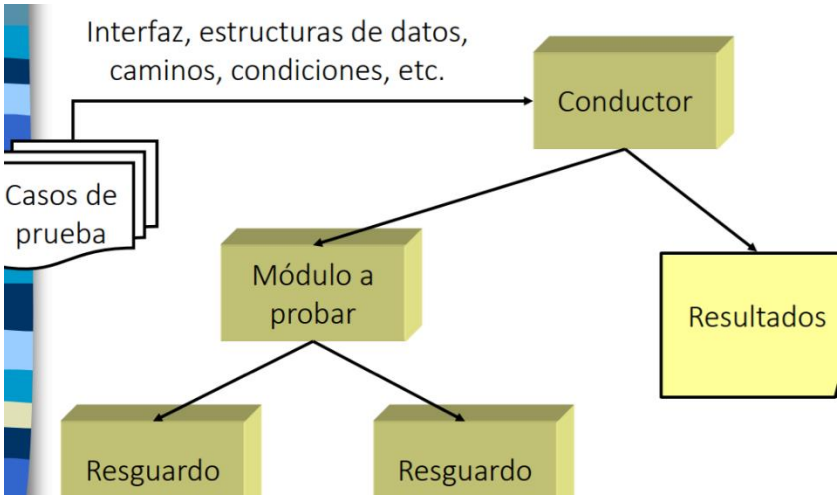
  - Pruebas de condición y decisión: Rama Verdadera y rama Falsa. Ejemplo diapo 45.

*Diseño de los casos de prueba:* Crear el subconjunto de todos los posibles casos de prueba que tiene la mayor probabilidad de detectar el mayor número posible de errores, usando técnicas de caja negra y después completar creando casos suplementarios que examinen la lógica del programa (caja blanca).

*Estrategia de pruebas:*




- Pruebas unitarias: comprueba la lógica, funcionalidad cada módulo. La realiza el programador en su entorno de trabajo.
  Enfoque de caja blanca, se examinan interfaces y funcionalidad. La prueba de unidad se simplifica cuando se ha diseñado un módulo con alto grado de cohesión.
  Ventajas: El error está más localizado. Se pueden probar simultáneamente varios módulos.
  Se crean módulos específicamente creados para la prueba:
  1. Conductor: llama al módulo a probar.
  2. Resguardo: es llamado por el módulo a probar.




- Pruebas de integración: comprueba la agrupación de módulos y su flujo de información. Enfoque de caja negra.
  ¿Qué módulo integrar? Módulos críticos, si cumple alguna de estas características: Está dirigido a varios requisitos del software. Tiene un nivel de control alto. Es complejo o contiene un algoritmo nuevo. Es propenso a errores. Tiene unos requisitos de rendimiento muy definidos o muy estrictos. Es un módulo de E/S.

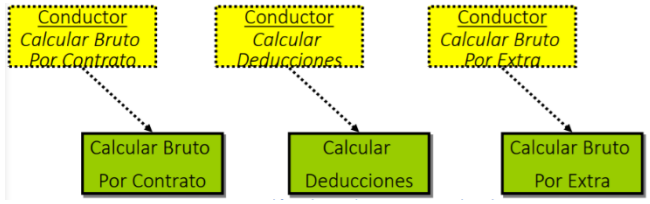
Hay que determinar la manera en la que se combinan los distintos módulos con pruebas incrementales y no incrementales:

*Incrementales:*

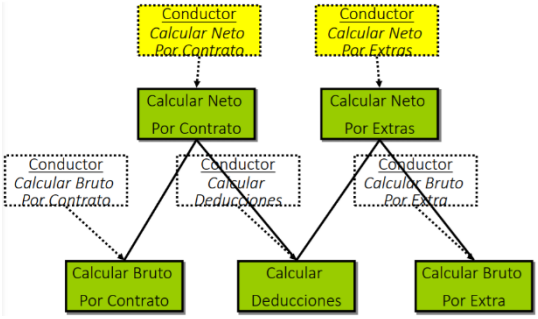
- Ascendentes: Comienza con los módulos terminales y se integra de abajo hacia arriba. Se utilizan módulos conductores.

1. Se combinan módulos del nivel más bajo en grupos.
2. Se construye un conductor para coordinar la Entrada y Salida de los casos de prueba.
3. Se prueba el grupo.
4. Se eliminan los conductores y se combinan los grupos moviéndose hacia arriba por la estructura del programa.

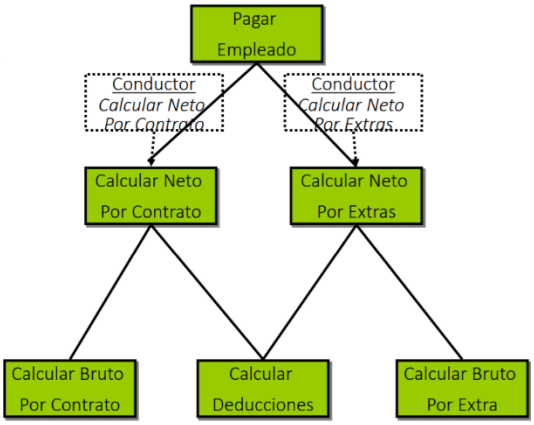
1




2

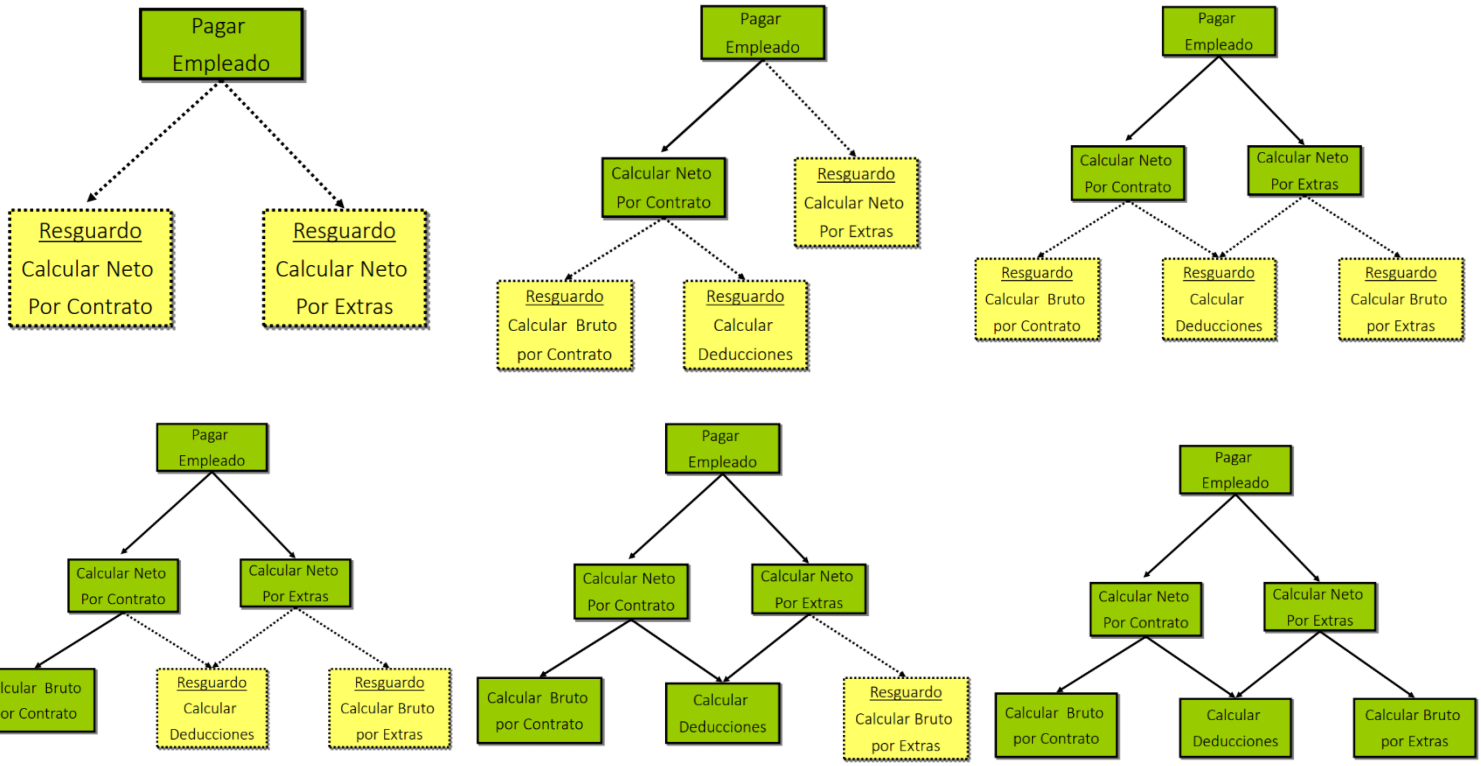



3




- Descendentes: Comienza con al módulo superior y se continúa hacia abajo por la jerarquía de control (en profundidad o en anchura). Se utilizan módulos resguardo.

1. Se usa el módulo de control principal como conductor de pruebas, construyéndoles resguardos para los módulos inmediatamente subordinados.
2. Se sustituyen uno a uno los resguardos por los módulos reales.
3. Se prueba cada vez que se integra un módulo.
4. Al acabar las pruebas, se reemplaza otro resguardo por el módulo real.
5. Se hacen pruebas de regresión: Repetir ciertos casos de prueba para asegurar que no se introducen nuevos errores.




- Sandwich: Combina las aproximaciones ascendentes y descendentes. Se aplica la integración ascendente en los niveles inferiores de la jerarquía de módulos. Paralelamente, se aplica la integración descendente en los niveles superiores de la jerarquía de módulos. La integración termina cuando ambas aproximaciones se encuentran en un punto intermedio de la jerarquía de módulos.

- Pruebas del sistema: Se integra con su entorno hardware y software: Pruebas de volumen, de funcionamiento, de recuperación, de seguridad, de resistencia (en condiciones de sobrecarga o límites), de rendimiento/comportamiento, etc.

- Pruebas de validación: comprueba la concordancia respecto a los requisitos software. La técnica utilizada es la de la caja negra. Consta de dos partes:
  - La validación por parte del usuario para comprobar si los resultados producidos son correctos.
  - La utilidad, facilidad de uso y ergonomía de la interfaz de usuario.

Dos tipos de prueba:

- Pruebas alfa: Las lleva a cabo el cliente en el lugar de desarrollo.
- Pruebas beta: Las lleva a cabo el cliente en su entorno. El desarrollador no está presente normalmente.

- Pruebas de aceptación: comprueba que el producto se ajusta a los requisitos del usuario. Consiste en la aceptación por parte del cliente del software desarrollado. Comprueba que el sistema está listo para su uso operativo. El usuario aporta los casos de prueba.

- Pruebas de verificación y validación de unidades, integración y sistema llevadas a cabo por el equipo de ingenieros de software.
- Pruebas de validación y evaluación del sistema llevadas a cabo por el usuario final.

Prevención de errores: Introducir precisión en el proceso de desarrollo software. Establecer una verificación al final de cada fase antes de comenzar la siguiente. Establecer diferentes procesos de pruebas.

Recomendaciones: Documentar los casos de prueba. Enfoque basado en el riesgo, concentrándose en los módulos más complejos y en los que podrían hacer el mayor daño si fallan. Minimizar la complejidad de los módulos. Diseñar las pruebas de unidad de forma sistemática. Recordar que también se pueden cometer errores en las pruebas.

Resumen: En la fase de pruebas hay que probar que el sistema hace lo que tiene que hacer y no hace lo que no tiene que hacer. La estrategia de pruebas a seguir es siempre de dentro hacia fuera. La construcción de casos de pruebas se realiza combinando técnicas de caja negra con técnicas de caja blanca. Cuando se termina el proceso de pruebas aún no se ha acabado el sistema, falta entregarlo y el resto de movidas post entrega.

## Unidad 7: Mantenimiento

Conjunto de actividades que se realizan sobre el software una vez que éste está operativo (después de la entrega) con el fin de corregir defectos, mejorar el rendimiento u otros atributos, o adaptarlo a un cambio en su entorno. Cubre la vida de un sistema software desde que se instala hasta que se reemplaza o da de baja. Llevar a cabo el mantenimiento de forma planificada facilita el mantenimiento y reduce su coste.

*Tipos de mantenimiento:*

- Correctivo (~20%):
  Modificar el sistema tras la detección de defectos, ambigüedades o errores. Incluye: Diagnóstico de errores y corrección de errores. Tipos: De emergencia y planificado.
- Adaptativo (~25%):
  Modificar el sistema para acomodarlo a cambios físicos del entorno. Incluye: Actividades para ajustar el software a un entorno nuevo (hw/sw). Actividades para añadir nuevos periféricos. Actividades para ajustar el software a cambios en fuentes externas (p.ej.: leyes).

- Perfectivo (~50%):

  Mejorar el sistema para cumplir con las nuevas necesidades/requerimientos de los usuarios/negocio. Incluye: Mejoras en el software para aumentar la eficiencia, rendimiento, etc. Actividades necesarias para cumplir nuevos requisitos relativos a fuentes externas (p.ej.: nuevo formato de informes).

- Preventivo (~5%):

  Modificar el sistema con los cambios necesarios para mantener la eficiencia y fiabilidad del software. Incluye: Revisión periódica de equipos, periféricos y protocolos asociados. Pruebas y revisiones periódicas del software:

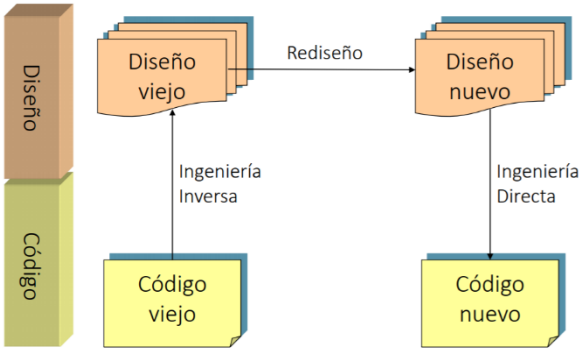
  - Monitorizar el rendimiento, incremento en el tráfico de datos, backups y recuperación de errores, crecimiento del tamaño de los ficheros/bases de datos.

  Aumentar el tamaño previsto de los ficheros/bases de datos.

  - Estructural: Modificar la arquitectura interna del sistema con el objetivo de mejorar su mantenibilidad. Incluye: Mejorar la documentación existente. Reestructurar el código para mejorar la legibilidad. Efectuar reingeniería del software para incrementar su modularidad.

    Reingeniería: Es el proceso de examinar un sistema software existente y modificarlo (reconstruirlo) con la ayuda de herramientas automáticas para: mejorar su mantenibilidad y compresión, incrementar su calidad y aumentar su vida. Normalmente se cambia la forma del software (ej. hacer el código más estructurado y legible o un diseño más completo), no la funcionalidad. Es un proceso de reconstrucción. Tipos:

    - Re-estructuración: Proceso de cambiar el código modificando su forma, pero no su funcionalidad.
    - Ingeniería inversa: Proceso de recobrar una descripción de más alto nivel de un sistema software a partir de una descripción de más bajo nivel (ej. diseño a partir de código o análisis a partir de diseño). La ingeniería inversa no cambia lo que hace el SW. Transforma la representación del SW a una forma más fácil de entender, más clara y más completa.




    - Migración: Proceso de convertir un sistema software de un lenguaje a otro o portarlo de una plataforma a otra.

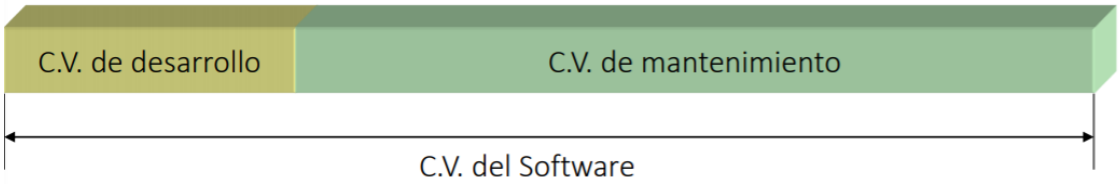
    ¿Cuándo aplicar reingeniería? Fallos frecuentes (difíciles de encontrar). Problemas de eficiencia o rendimiento. Difícil de

modificar. Difícil de probar. Frecuente mantenimiento. Caro de mantener, etc.

*Estrategias de mantenimiento:*

- Mantenimiento estructurado/planificado: Se acumulan los cambios (mejoras principalmente) y se realizan todos a la vez cada cierto tiempo en fechas planificadas.
- Mantenimiento no estructurado: Bajo petición. Cada vez que hay una demanda, se lleva a cabo el mantenimiento pedido.
- Combinado: Estructurado y cuando hay emergencias (especialmente el correctivo). Suele ser la más beneficiosa para la empresa desarrolladora.

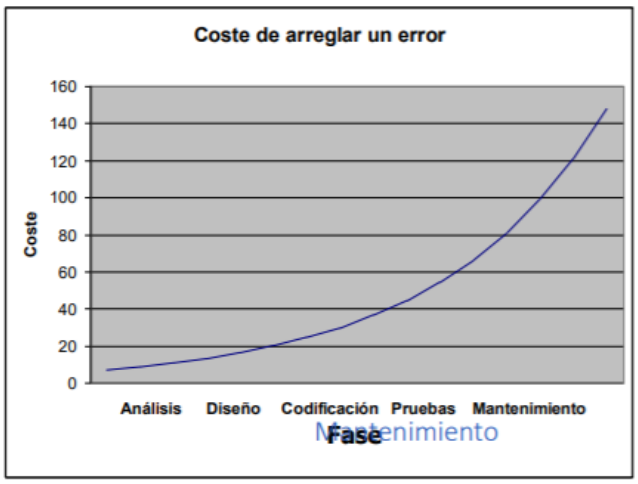
*Ciclo de vida del mantenimiento:*




Está incluido dentro del ciclo de vida del software, pero también debe tenerse en cuenta en las actividades de desarrollo.

Actividades realizadas: Actualizar la documentación. Controlar las versiones (Gestión de configuraciones). Aseguramiento de calidad.

*Coste del mantenimiento:*




Entre el 80% y el 90% del presupuesto de muchas empresas de informática se gasta en mantenimiento. El mantenimiento de una aplicación grande consume de 2 a 4 veces más recursos que el desarrollo de la misma. Los costes de mantenimiento son difíciles de estimar.

*Problemas del mantenimiento:*

Dificultad en el seguimiento de los cambios. Dificultad en conocer el proceso seguido en la construcción del software. Efectos secundarios en los cambios. Mala documentación. Falta de previsión de futuros cambios durante el diseño del software.

*Mantenibilidad:*

Es la facilidad de mantener un software. Típicamente del 10 al 15% de las líneas de código se cambian anualmente, por lo que siempre es necesaria.

*Documentos:*

- Contrato de mantenimiento: Documento firmado por ambas partes donde se recogen las condiciones del mantenimiento.
- Plan de mantenimiento.
- Formulario de petición de mantenimiento.
- Informe de cambios del software.
- Documentación interna al programa.

## Unidad 8: Gestión de la Configuración del Software

Configuración del software: Es el estado actual del sistema software y las interrelaciones entre sus componentes constitutivos (código fuente, datos y documentación).

Gestión de Configuración del Software (GCS): Es una disciplina de gestión que permite controlar formalmente la evolución y los cambios del software, garantizando la visibilidad en el desarrollo y en el producto, y la trazabilidad en el producto durante todo su ciclo de vida. Su misión es identificar, controlar y organizar la evolución de un sistema software.

Todos los sistemas software cambian a lo largo de su vida (diferente versión, plataforma, etc.) Hay que asegurar que los cambios producidos ocasionen el mínimo coste. La GCS asegura:

- Coherencia entre versiones.
- Seguridad ante pérdidas de software o de personal.
- Reutilización del software.
- Poder recuperar cualquier versión realizada por cualquier desarrollador en cualquier momento.
- Calidad: se aceptan únicamente los elementos formalmente revisados y aprobados.

Objetivo: Establecer y mantener la integridad de los productos generados durante un proyecto de desarrollo de software y a lo largo de todo el ciclo de vida. Actúa sobre programas, documentos y datos.

Elemento de Configuración del Software (ECS): Cada uno de los componentes básicos de un producto software sobre los que se realizará un control. Tiene un nombre y puede evolucionar. Cumple dos condiciones: θ Evoluciona en el tiempo. θ Nos interesa controlar esa evolución.

Línea Base: Es una configuración de referencia en el proceso de desarrollo del software a partir de la cual las revisiones de los elementos de configuración del software se han de realizar de manera formal, para controlar los cambios en el software, sin impedir llevar a cabo aquellos que sean justificados.

*Tipos:* (las más habituales, pero pueden variar)

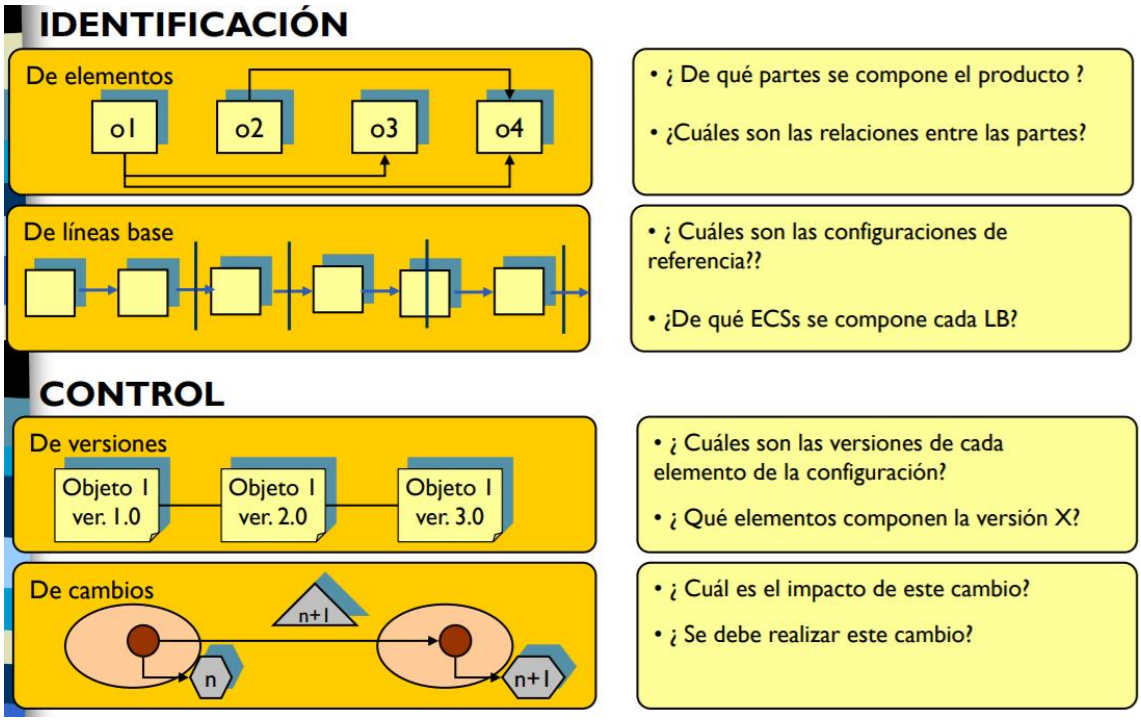
- Línea base Funcional: después de la fase de análisis.
- Línea base de Diseño: después de la fase de diseño.
- Línea base de Producto: en las pruebas cuando tengamos un producto final estable (ej. después de las pruebas de integración).
- Línea base Operativa: después de entregar el sistema final.

- Objetivos secundarios:
  - Identificar los resultados de las tareas realizadas en cada fase.
  - Asegurar que se ha completado la fase.
  - Servir como punto de partida para los desarrollos posteriores.
  - Servir como punto de partida para las peticiones de cambio.

Una vez que se ha desarrollado y revisado un elemento de configuración, pasa a formar parte de la siguiente línea base planificada del proyecto. Esto es lo mismo que decir que el elemento se convierte en línea base.

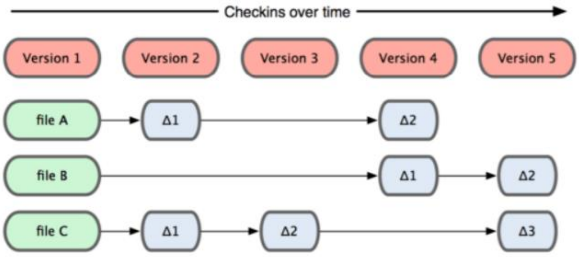
Cuando un ECS se convierte en una línea base se introduce en una Base de Datos del proyecto. A partir de aquí el elemento se debe modificar siguiendo un procedimiento establecido. El objetivo es que se puedan hacer los cambios necesarios, pero de manera controlada y previa autorización. Los cambios sobre un elemento de una Línea Base producen la creación de una nueva versión del elemento.

*Actividades:*

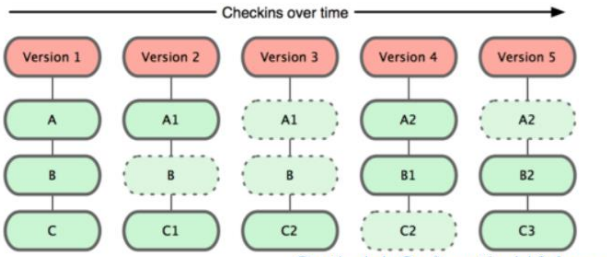


*Herramientas:*

La GCS se realiza con herramientas automáticas como SVN y GIT.




SVN almacena los datos como cambios de cada archivo respecto a una versión base.




Git almacena la información como instantáneas del proyecto a lo largo del tiempo



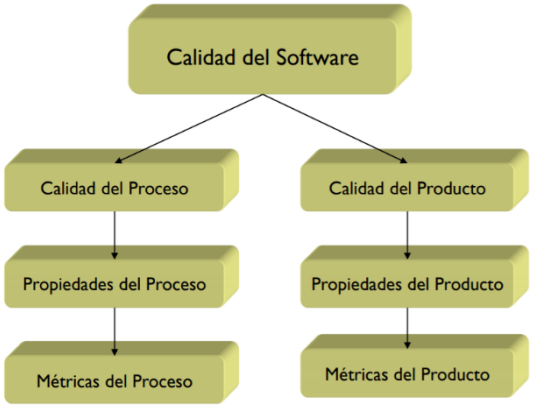
*Documentación:*

- Plan de gestión de configuración: Define los planes, normas y procedimientos asociados a la implantación de un sistema GCS. Se realiza en la fase de planificación.
- Dossier de gestión de cambios: Contiene toda la información necesaria para asegurar el control de los cambios.

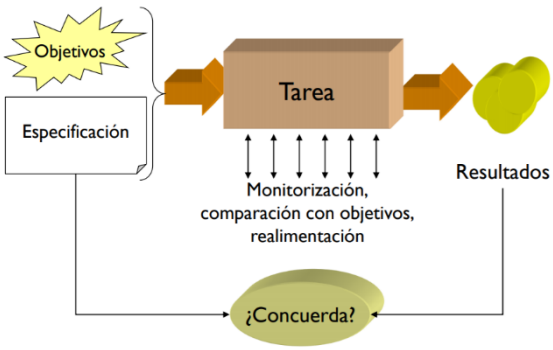
## Unidad 9: Aseguramiento de Calidad del Software

- V&V (Verificación y Validación): Actividades técnicas. Trata errores técnicos del software.
    - Error <- Acción Humana
- QA (Quality Assurance): Actividades de gestión. Trata fallos y defectos de calidad del software.
    - Defecto <- Falta en Req, Diseño, Código.
    - Fallo <- Manifestación de una falta en Sistema

"Calidad del software es el grado en el que un sistema, componente o proceso cumple los requisitos especificados y las necesidades o expectativas del cliente o usuario"




Métricas del producto y proceso:



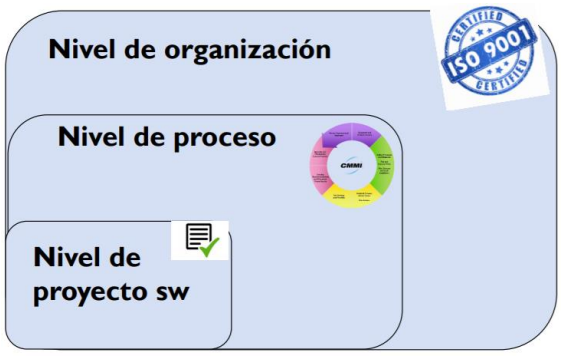
*Actividades:*


*Midiendo la calidad del SW:*

Factores que pueden ser medidos directamente: Errores reportados/mes, nº de documentos generados, etc.

Factores que no pueden ser medidos directamente: Facilidad de uso, facilidad de mantenimiento, etc.

*Niveles de calidad:*




*Medidas Analíticas de Software Quality Assurance:*

- Medidas estáticas: Analizan el objeto a medir sin necesidad de ejecutarlo:
    - Auditorías: Realización de una investigación para determinar: (i) el grado de cumplimiento de estándares, requisitos, procedimientos y métodos definidos; (ii) la efectividad del proceso llevado a cabo. Pueden ser del producto, del proceso o del sistema de aseguramiento de calidad.
    - Revisiones formales: Reunión formal donde se analizan de forma estructurada los resultados (parciales o finales) de un proyecto software para evaluar el proceso y producto software y detectar desviaciones respecto a las especificaciones de calidad. Se realizan en todas las fases del proceso software, pero son especialmente efectivas en las primeras.

- Medidas dinámicas: Requieren la ejecución del objeto que está siendo medido/probado.
    - Técnicas. Aplicar técnicas de ingeniería del software para mejorar la calidad de los productos (principalmente) software.

- Organizativas. Aplicación de planes para proporcionar una mejor calidad a los procesos software.
- Humanas. Dotar de formación al equipo de desarrollo para asegurar su eficiencia y eficacia.

*Sistemas SW de alta calidad:*

Debería:

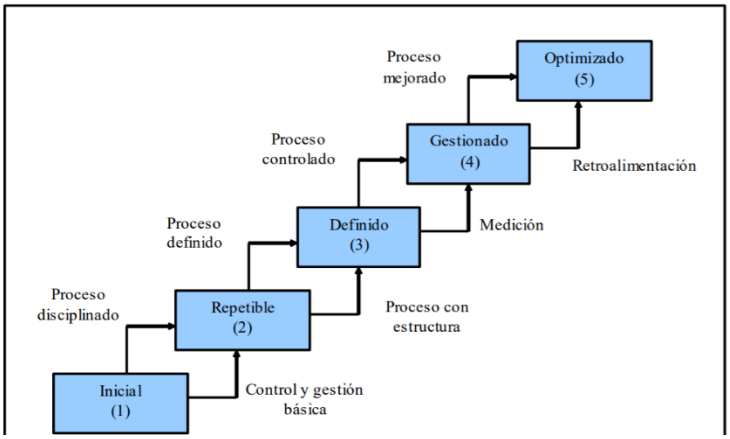
- ❑ Cumplir con un conjunto de requisitos funcionales claramente especificados.
- ❑ Cumplir con un conjunto de factores de calidad del software medibles acordados previamente entre el usuario y desarrollador.
- ❑ Cumplir plazos y presupuesto.
- ❑ Mantenerse rentable durante el desarrollo y mantenimiento.
- ❑ Ser útil para el usuario, *mantenible* y adaptable durante todo su ciclo de vida.

*Documentación:*

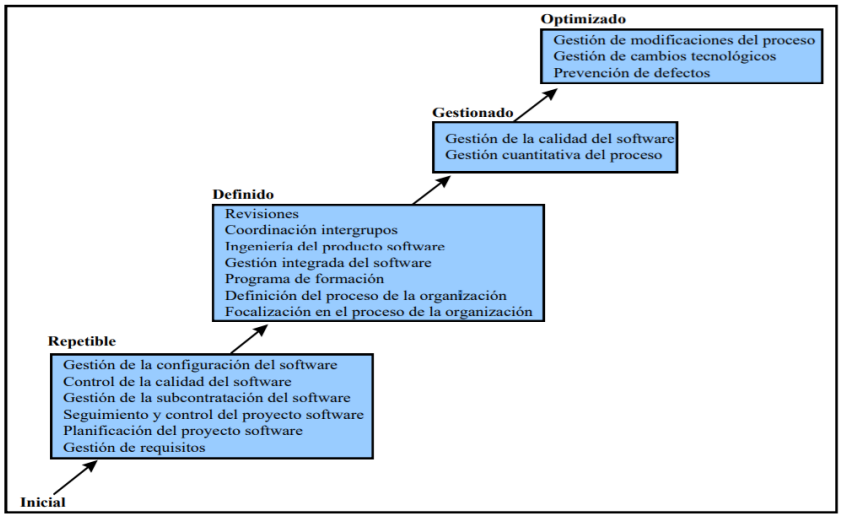
- Plan de aseguramiento de la calidad: Planificación de cómo se va a construir la calidad en el software y cómo se va a evaluar. Este documento se debe producir en una fase muy temprana del proceso de desarrollo.

*Modelos de mejora de procesos, CMMI (Capability Maturity Model Integration)*

**Niveles de Madurez**




**Modelo CMM – Áreas de Proceso Clave**




**Elementos básicos de cada Nivel**