

Resumen

IDS I

Introducción

Desafíos de la IDS:

- Escala
Los métodos usados deben poder aplicarse tanto en problemas grandes como chicos.
- Calidad
La calidad es un factor diferencial del Software industrial. Además, no es fácilmente cuantificable.
- Productividad
Se busca maximizar la cantidad de líneas de código escritas en un determinado tiempo (en general KLOC/PM).
- Consistencia
Los métodos exitosos, tienen que poderse replicar.
- Cambios
Adaptarse principalmente a modificaciones en, la SRS, y empresas/instituciones.

Fases del Proceso de Desarrollo	Entrada	Salida
Análisis de Requisitos y Especificación	Necesidades del Cliente/Usuario	SRS (Software Requirements Specification)
Arquitectura y Diseño	SRS	Diseño Arquitectónico
Codificación	Diseño Arquitectónico	Código
Testing	código	Código Testeado + Resultados del Testing
Entrega e Instalación	Código Testeado + Resultados del Testing	-

Calidad del Software (Según estándar ISO)

1 Funcionalidad

La capacidad de proveer funciones que cumplen, las necesidades establecidas e implicadas por los requisitos.

2 Confiabilidad

Capacidad de realizar las funciones requeridas, bajo las condiciones establecidas durante un tiempo específico.

3 Usabilidad

Capacidad de ser comprendido, aprendido, y usado.

4 Eficiencia

Capacidad de proveer desempeño apropiado, relativo a la cantidad de recursos usados.

5 Mantenibilidad

Capacidad de ser modificado (no redundante, fácil de leer, etc.) buscando, corregir, mejorar y adaptar.

6 Portabilidad

Capacidad de ser adaptado a distintos entornos sin aplicar otras acciones que las previstas en este propósito.

Análisis y Especificación de Requerimientos

Características SRS

1 Correcta

Cada requerimiento representa precisamente alguna característica **deseada por el cliente en el sistema final.**

2 Completa

Todas las características **deseadas por el cliente** están descritas.

3 No Ambigua

Cada requerimiento tiene exactamente un significado (no se superpone con otros).

4 Consistente

Ningún requerimiento contradice a otro.

5 Verificable:

Si para cada requerimiento existe, algún proceso efectivo; capaz de verificar que, el software final lo satisface.

6 Rastreable

Se debe poder determinar el origen de cada requerimiento, y cómo este se relaciona con los elementos del software. Dado un requerimiento se debe poder detectar, en qué elementos de diseño o código tiene impacto (*hacia adelante*). Dado un elemento de diseño o código se debe poder rastrear que requerimientos está atendiendo (*hacia atrás*).

7 Modificable

Si la estructura y estilo de la SRS es tal que, permite incorporar cambios fácilmente **preservando completitud y consistencia.**

8 Ordenada en Aspectos de Importancia y Estabilidad

Los requisitos críticos o importantes, deben ser claramente indicados; especificando, aquellos más relevantes para el cliente/usuario. Además, generar un orden de prioridades, contemplando cuáles son esenciales y difícilmente cambian en el tiempo, de los propensos a cambios; esto disminuye errores derivados de cambios en los requerimientos.

Casos de Uso:

Especifican únicamente, la funcionalidad provista por el sistema. Capturando, su comportamiento externo; como interacciones entre, los usuarios, y este

Actor: Una persona o un sistema que interactúa con el sistema propuesto para alcanzar un objetivo. Un actor es una entidad lógica; actores receptores y actores transmisores son distintos (aún si es el mismo individuo).

Actor primario: El actor principal que inicia el caso de uso. El caso de uso debe satisfacer su objetivo (AP es el interesado). La ejecución real puede ser realizada por un sistema u otra persona en representación del actor primario.

Paso: Es una acción lógicamente completa realizada tanto por el actor como por el sistema. Es una interacción entre el usuario y el sistema.

Escenario Alternativo: Es una variación del flujo principal que puede ocurrir bajo ciertas condiciones pero no necesariamente implica un error o fallo. Por ejemplo, si el usuario elige una opción diferente o si se introduce una información válida pero no común. El escenario alternativo sigue llevando a una conclusión exitosa del caso de uso, aunque por un camino distinto al flujo principal.

Escenario Excepcional: Es una desviación del flujo principal debido a errores, fallos, o condiciones anómalas que impiden completar el caso de uso con éxito. Este tipo de escenario maneja situaciones donde algo va mal, como la entrada de datos incorrectos o la pérdida de conexión, y describe cómo el sistema debe responder para manejar estas excepciones de manera controlada y segura.

Escenario exitoso principal: Cuando todo funciona normalmente y se alcanza el objetivo.

Caso de Uso número: Nombre

Actor Principal:

Actor Secundario: Si existiera.

Precondición: !Si no hay, se indica cómo" !Ninguno".

Escenario Exitoso Principal

Paso 1: Acción (Descripción del significado en términos del sistema.) ¡SIEMPRE EMPIEZA EL AP !

Paso 2: Acción (Descripción del significado en términos del sistema.)

...

Paso N: Acción (Descripción del significado en términos del sistema.)

Escenarios Excepcionales:

Excepción numero_de_paso.a:

Solución

Excepción numero_de_paso.b:

Solución

...

Excepción nuevo_numero_de_paso.a:

Solución

Excepción nuevo_numero_de_paso.b:

Solución

...

Escenarios Alternativos

Alternativo numero_de_paso_generado_de_difurcación.a

Paso numero_de_paso_generado_de_difurcación:

Condición de difurcación

Paso numero_de_paso_generado_de_difurcación + 1:

...

Paso numero_de_paso_generado_de_difurcación + N:

Alternativo numero_de_paso_generado_de_difurcación.b

Paso numero_de_paso_generado_de_difurcación:

Condición de difurcación

Paso numero_de_paso_generado_de_difurcación + 1:

...

Paso numero_de_paso_generado_de_difurcación + N:

Alternativo nuevo_numero_de_paso_generado_de_difurcación.a

Paso nuevo_numero_de_paso_generado_de_difurcación:

Condición de difurcación

Paso nuevo_numero_de_paso_generado_de_difurcación + 1:

...

Paso nuevo_numero_de_paso_generado_de_difurcación + N:

Alternativo nuevo_numero_de_paso_generado_de_difurcación.b

Paso nuevo_numero_de_paso_generado_de_difurcación:

Condición de difurcación

Paso nuevo_numero_de_paso_generado_de_difurcación + 1:

...

Paso nuevo_numero_de_paso_generado_de_difurcación + N:

...

Punto Función:

Estimación del

Métrica que, ~~estima~~ el tamaño del software final (medido en líneas de código); usando exclusivamente la funcionalidad especificada en la SRS. Para ello: 1.º identificamos las funciones del sistema, enfocándonos en 5 tipos; 2.º las clasificamos según su complejidad en: simples, promedio, o complejas; 3.º obtenemos el UFP; 4.º evaluamos 14 características del entorno, para discernir su complejidad y determinar el CAF; 6.º obtenemos los puntos función ajustados, multiplicando el UFP por el CAF; finalmente, estos se convierten en una estimación de las LOC mediante factores de conversión específicos del lenguaje de programación.

Cálculo del PUNTO FUNCIÓN NO AJUSTADO (UFP):

1. **Entradas Externas:** Datos o comandos que llegan desde fuera de la aplicación.
2. **Salidas Externas:** Información o resultados que la aplicación muestra o envía hacia afuera.
3. **Archivos Lógicos Internos:** Grupos lógicos de datos usados internamente por la aplicación.
4. **Archivos de Interfaz Externa:** Archivos compartidos o pasados entre la aplicación y otros sistemas.
5. **Transacciones Externas:** Interacciones inmediatas con sistemas externos (por ejemplo, consultas a bases de datos).

Tipo de Función	Complejidad		
	Simple	Promedio	Compleja
Entradas externas:	3	4	6
Salidas externas	4	5	7
Archivos lógicos internos	7	10	15
Archivos de interfaz externa	5	7	10
Transacciones externas	3	4	6

$$UFP = \sum_{i=1}^5 [\sum_{j=1}^3 (w_{ij} c_{ij})] \text{ Con } w_{ij} \text{ peso del } i \text{ tipo de función, con } j \text{ nivel de complejidad; } c_{ij} \text{ cantidad de funciones de ese tipo y nivel.}$$

Cálculo del Factor de Ajuste de Complejidad (CAF):

Ajustar el UFP de acuerdo a la complejidad del entorno. Se evalúa según las siguientes características:

1. Comunicación de datos
2. Procesamiento distribuido
3. Objetivos de desempeño
4. Carga en la configuración de operación
5. Tasa de transacción
6. Ingreso de datos online
7. Eficiencia del usuario final
8. Actualización online
9. Complejidad del procesamiento lógico
10. Reusabilidad
11. Facilidad para la instalación
12. Facilidad para la operación
13. Múltiples sitios
14. Intención de facilitar cambios

Cada uno de los ítems anteriores, debe evaluarse como:

Valor	pi
No presente	0
Influencia insignificante	1
Influencia moderada	2
Influencia promedio	3
Influencia significativa	4
Influencia fuerte	5

$$CAF = 0.65 * 0.01 \sum_{i=1}^{14} p_i \quad \text{Puntos de función: } CAF * UFP$$

Arquitectura del software

Todo sistema complejo se compone de, subsistemas que interactúan entre sí. Un enfoque para diseñar sistemas consiste en; dividirlos en, partes lógicas, **comprendibles, individualmente**; y describir sus relaciones. La arquitectura es, el diseño de más alto nivel, y busca optimizar esta división.

La arquitectura de SW de un sistema es la estructura del sistema que comprende los elementos del SW, las **propiedades externamente visibles** de tales elementos, y la relación entre ellas. En general, se poseen, diferentes estructuras (vistas), de aspectos ortogonales.

Vista de componentes y conectores (C&C)

: Describe una estructura en ejecución del sistema. Identificando las componentes existentes, y como interactúan entre sí en tiempo de ejecución. Posee 2 elementos principales; componentes, elementos computacionales o de almacenamiento; y conectores, mecanismos de interacción entre componentes.

Ó vista

Estilos arquitectónicos de, la vista C&C

: un estilo arquitectónico, una familia de arquitecturas que satisfacen las restricciones de este estilo;

Tubos y Filtros (Pipe and Filter): Un sistema que use este estilo, utiliza una red de transformadores para realizar el resultado deseado. Un filtro realiza transformaciones, y pasa los datos a otro; mediante, un tubo.

- Adecuado para sistemas que fundamentalmente realizan **transformaciones de datos**.
- Conectores: Tubos
- Componentes: Filtros
- Restricciones:
 1. Cada filtro debe trabajar sin conocer la identidad de los filtros productores o consumidores.
 2. Un tubo debe conectar un puerto de salida de un filtro a un puerto de entrada de otro filtro.
 3. Un sistema puro de tubos y filtros usualmente requiere que cada filtro tenga su propio hilo de control.
 4. Un filtro es una entidad **independiente** y asincrónica que, se limita a consumir y producir datos.
 5. Un filtro, no necesita conocer la identidad de los filtros, a los que envían datos, y tampoco de aquellos del los que recibe datos.
 6. Los filtros realizan, buffering, y sincronización.
 7. Un tubo es, un canal unidireccional que transporta un flujo de datos de un filtro a otro.
 8. Los tubos solo conecta 2 componentes.

Datos Compartidos

- Útil para sistemas orientados a bases de datos; o sistemas de web.
- Componentes: Repositorio de datos, y usuario de datos.
- Conectores: Lectura/Escritura
- Restricciones:
 1. La comunicación entre los usuarios de los datos solo se hace a través del repositorio.
 2. El repositorio de datos, provee almacenamiento permanente confiable.
 3. Los usuarios de datos, acceden a los datos en el repositorio, realizan cálculos, y ponen los resultados otra vez en el repositorio.
- Variantes:
 1. Pizarra: Cuando se agregan/modifican datos en el repositorio, se informa a todos los usuarios (entidad activa).
 2. Repositorio: El repositorio es pasivo.

Cliente-Servidor

- Componentes: Clientes, y servidores.
- Conectores: Solicitud/Respuesta (Request/Reply)
- Restricciones:
 1. Los clientes solo se comunican con el servidor, pero no con otros clientes.
 2. La comunicación siempre es iniciada por el cliente, quien le envía una solicitud al servidor y espera una respuesta de este.
 3. En general, tiene una capa de base de datos.
 4. La comunicación es asincronica.

Estilo Publicar-Suscribir (Publish-Subscribe, estilo c&c)

- Dos tipos de componentes: las que publican eventos y las que se suscriben a eventos.
- Cada vez que un evento es publicado, se invoca a las componentes suscriptas a dicho evento.

Estilo Peer-to-Peer

- Un único tipo de componente.
- Cada componente le puede pedir servicios a otro. ≈ Modelo de computación orientado a objetos.

Estilo de Procesos que se Comunican: Procesos que se comunican entre sí, a partir del pasaje de mensajes.

Método ATAM (Arquitectura de Evaluación y Análisis):

ATAM nos permite, evaluar arquitecturas; analizando sus propiedades, y concesiones entre ellas.

Paso 1-Recolectar los Escenarios: Se recopilan los escenarios que describen, las interacciones del sistema. Estos son, situaciones o eventos que el sistema debe manejar.

Paso 2-Recolectar Requerimientos y/o Restricciones: Se definen los requerimientos y restricciones específicos para cada escenario. Esto implica establecer claramente qué se espera que el sistema haga en cada situación. Además, se deben especificar los niveles deseados para los atributos de interés (preferiblemente cuantificados).

Paso 3-Describir las Vistas arquitectónicas: Las diferentes vistas del sistema que serán evaluadas se recopilan. Distintas vistas pueden ser necesaria para distintos análisis.

Paso 4-Análisis Específico para Cada Atributo: Se analizan las vistas bajo distintos escenarios de forma separada para cada atributo de interés; esto determina los niveles que la arquitectura puede proveer para c/u; y se compara con los requeridos; esto da la base para la elección entre una arquitectura u otra, o la modificación de la arquitectura propuesta.

Paso 5-Identificar Puntos Sensibles y de Compromiso: Se lleva a cabo un análisis de sensibilidad el cual determina, el impacto que tiene un elemento de la arquitectura en un atributo de calidad específico. Los elementos de mayor impacto son, puntos de sensibilidad. Además se hace un análisis de compromiso; los puntos de compromiso son, los elementos de sensibilidad para varios atributos.

que son puntos de

Diseño

Principio abierto-cerrado:

Las entidades de software deben ser abiertas para extenderlas y cerradas para modificarlas; el comportamiento puede expandirse para, adaptarse a nuevos requerimientos;; pero, el código existente debe permanecer inmutable. Este principio permite, minimizar el riesgo de dañar funciones existentes al ingresar código.

Acoplamiento y cohesión:

Empecemos definiendo módulo como, una parte lógicamente separable de un programa; es decir, se caracteriza por ser una unidad **discreta e independiente** respecto a la compilación y el código. La independencia entre ellos depende de, si c/u puede funcionar completamente sin la presencia de otro. Existen 2 criterios utilizados para seleccionar módulos capaces de, soportar abstracciones bien definidas, y solucionables/modificables separadamente; **acoplamiento, y cohesión**. El 1.^o refiere a, la inter-modularidad pues, captura la noción de dependencia entre módulos; se busca, minimizarlo. El 2.^o es un concepto intra-modular, ya que alude a cuán relacionado están los elementos de un módulo; el objetivo es, maximizarlo.

Abierto-cerrado en OO:

se satisface si, se usa apropiadamente la herencia y el polimorfismo. La 1.^a, crea nuevas subclases para extender el comportamiento sin modificar la clase original.

Principio de sustitución de Liskov:

Un programa que utiliza un objeto O con clase C debería permanecer inalterado si O se reemplaza por cualquier objeto de una subclase de C.

Abierto-cerrado & Liskov:

Si las jerarquías de un programa siguen Liskov, entonces el programa responde al principio abierto-cerrado.

Niveles del diseño

- **Diseño arquitectónico:** Identifica las componentes necesarias del sistema, su comportamiento y relaciones.
- **Diseño de alto nivel:** Es la vista de módulos del sistema. Es decir: cuáles son los módulos del sistema, qué deben hacer, y cómo se organizan e interconectan. Se divide en diseño orientado a, funciones, y objetos.
- **Diseño detallado o diseño lógico:** Establece cómo se implementan las componentes/módulos de manera que satisfagan sus especificaciones. Incluye detalles del procesamiento lógico (i.e. algoritmos) y de las estructuras de datos. Muy cercano al código.

Principales criterios para evaluar el diseño

- **Correctitud:** se puede resumir en las siguientes 2 preguntas
 - ¿El diseño implementa los requerimientos?
 - ¿Es factible el diseño dada todas las restricciones especificadas en la SRS?
- **Eficacia:** Alude a si el diseño, hace uso apropiado de los recursos del sistema (principalmente CPU y memoria).
- **Simplicidad:** Un diseño simple permite, la comprensión del sistema. Facilitando, su mantenimiento; mejorando, el testing, debugging, y modificabilidad del código.

Principios fundamentales

1. **Partición y jerarquía:** Divide al problema, en pequeñas partes manejables. Las cuales deben poder, solucionarse y modificarse separadamente; tratando de mantener la mayor independencia posible entre ellas. Esta fragmentación determina, una jerarquía entre componentes en el sistema; usualmente formada por, la relación "es parte de". Claro que estas divisiones no son totalmente independientes entre sí, pues deben comunicarse/operar para solucionar el problema. Es importante entender que, a mayor cantidad de módulos, el costo del participado aumenta; debido al, incremento en la complejidad de la comunicación entre ellos.
2. **Abstracción:** Describe cada componente, solo usando su comportamiento externo; y define la interacción entre ellas, en función de este. Permitiendo, focalizarnos en una sola de ellas sin preocuparnos por las demás; además de facilitarle al diseñador, el manejo de la complejidad; y generar una transición gradual, de lo más abstracto a lo concreto. Existen 2 tipos:
 1. **Abstracción funcional:** Específica, el comportamiento funcional de un módulo; tratándolos como, funciones de entrada/salida; y pudiéndolos especificar por medio de, pre y post condiciones. Forma la base de, las metodologías orientadas a funciones.
 2. **Abstracción de datos:** Entidad del mundo real que, provee servicios al entorno; para las entidades de datos, se espera operaciones de un objeto de datos. Este tipo de abstracción brinda esta visión:
 - i. Los datos se tratan como un objeto junto con, sus operaciones.
 - ii. Las operaciones definidas para un objeto, solo puede aplicarse sobre él.
 - iii. Desde fuera, los detalles internos del objeto permanecen ocultos; siendo sus operaciones, lo único visible.
3. **Modularidad:** Un sistema se dice modular si consiste de componentes discretas (separadas unas de otras) tales que, puedan implementarse separadamente y un cambio a una de ellas tenga mínimo impacto sobre las otras. La modularidad provee, la absrración en el software; es el soporte de, la estructura jerárquica de los programas; mejora la claridad de diseño y facilita la implementación; aunado a lo anterior, reduce los costos de, testing, debugging, y mantenimiento.

Refinamiento top-down y bottom-up

Los enfoques de refinamiento top-down y bottom-up son estrategias complementarias en el diseño de sistemas. El 1^{ro} lo descompone, comenzando desde una visión global y abstracta, desglosándolo en componentes más específicos y detallados hasta llegar a módulos implementables directamente. El 2.^o se centra en crear primero los componentes bajo nivel, combinándolos gradualmente para construir sistemas más complejos y de alto nivel.

Diseño Orientado a Funciones

El **diseño funcional** se enfoca en descomponer un sistema en funciones o procedimientos que transforman entradas en salidas. Se basa en la abstracción y la descomposición funcionales, donde el software es visto como una función de transformación. Los módulos se definen en términos de sus funciones, y se organizan jerárquicamente para lograr un bajo acoplamiento y alta cohesión.

Tipos de Acoplamiento: El diseño funcional se enfoca en descomponer un sistema en funciones o procedimientos que transforman entradas en salidas. Se basa en la abstracción y la descomposición funcionales, donde el software es visto como una función de transformación. Los módulos se definen en términos de sus funciones, y se organizan jerárquicamente para lograr un bajo acoplamiento y alta cohesión.

Acoplamiento	Tipo de conexión	Complejidad de las interfaces	Tipo de flujo de información
Bajo	Al módulo por entradas específicas	Simples, obvias	 Datos
Alto	A elementos internos	Complicadas, oscuras	 Control  Híbridos

Tipos de cohesión: (Ordenadas de la más débil a la más fuerte.)

1. **Casual:** Solo están agrupados en el módulo; no hay otra relación entre ellos más allá de la pertenencia al módulo que los contiene.
2. **Lógica:** Existe alguna relación lógica entre los elementos del módulo; los elementos realizan funciones dentro de la misma clase lógica.
3. **Temporal:** Los elementos están relacionados en el tiempo y se ejecutan juntos.
4. **Procedural:** Contiene elementos que pertenecen a una misma unidad procedural.
5. **Comunicacional:** Los elementos están juntos porque operan sobre el mismo dato.
6. **Secuencial:** Los elementos están juntos porque la salida de uno se corresponde con la entrada del otro.
7. **Funcional:** Todos los elementos están relacionados para llevar a cabo solo una función.

Metodología de Diseño

Estructurado

~~Estr~~

Es una metodología de diseño. Estas tienen el fin de, dar pautas para auxiliar al diseñador en el proceso de diseño; aunque esta actividad, no se puede reducir a una secuencia de pasos mecánicos.

La SDM, está dirigida al diseño orientado a funciones; pues, ve al software como una función de transformación que convierte a una entrada en una salida; y hace uso de, la abstracción y descomposición funcionales. Su objetivo es, especificar módulos de funciones y sus conexiones; siguiendo, una estructura jerárquica con bajo acoplamiento y alta cohesión. Esta metodología intenta, una factorización completa; es decir, descomponer cada módulo en, subordinados donde se realiza la computación, y un principal encargado de la coordinación; pues si se logra entonces, el procesamiento real se hará en los módulos atómicos (aquellos que no tienen subordinados). SDM posee los siguientes pasos:

1. **Reformular el problema como un DFD:** Este, captura el flujo de datos del sistema propuesto; ignorando detalles procedurales, sin ser tan restrictivo como en análisis y especificación de requerimientos. El fin es identificar las entradas, salidas, fuentes , transformadores, y sumideros del sistema. Idealmente, se deben realizar varios DFDs, antes de decidirse por uno.
2. **Identificar las entradas y salidas más abstractas:** Aquí separamos los transformadores que dan un formato útil a las entradas y salidas; de aquellos encargados de hacer las transformaciones reales (transformadores centrales). Para esto, identificamos; la MAI, ítems de datos del DFD, considerados entrantes, y con la mayor distancia a la entrada real. ; y la MAO, análoga a la anterior pero para las salidas. Los transformadores centrales son los ubicados entre la MAI, y la MAO.
3. **Realizar el primer nivel de factorización:** Especificamos, el módulo principal; junto con uno subordinado por cada, ítem de datos de la MAI, elemento de datos de la MAO, y transformador central. Estos distintos tipos de módulos son independientes, y pueden diseñarse separadamente.
4. **Factorizar los módulos de entrada, de salida, y transformadores:**
 1. **Factorizar los módulos de entrada:** Para cada módulo de entrada sé, repite el “Paso 3” considerando a c/u como un principal; esta factorización se repite hasta, llegar a la entrada física.
 2. **Factorizar los módulos de salida:** El caso de los módulos de salida es, simétrico al anterior.
 3. **Factorizar los transformadores centrales:** Por último, no hay reglas para factorizar los módulos transformadores; pero, se suele utilizar refinamiento top-down: buscando, los sub transformadores que lo conforman; luego se repite el proceso para c/u de ellos hasta, alcanzar los módulos atómicos.
5. **Mejorar la estructura:** mediante heurísticas, análisis de transacciones, etc.

Métricas

Proveen una evaluación cuantitativa del diseño; permitiendo, mejorar el producto final.

1. **De red:** Se enfoca en la estructura del diagrama de estructuras; un buen diagrama es aquel que, para cada módulo, tiene un solo módulo invocador (reduciendo acoplamiento). Cuanta mayor desviación tenga a esta forma de árbol, más impuro será. Antes de dar la fórmula, observar que esta métrica no contempla las rutinas comunes como las bibliotecas. La impureza del grafo se calcula como el número de nodos menos el número de aristas menos 1; si es cero, entonces es un árbol.
2. **De estabilidad:** La estabilidad trata de capturar el impacto de los cambios en el diseño. Se busca aumentar dicho parámetro. La estabilidad de un módulo está dada por la cantidad de suposiciones que otros módulos hacen de este; lo cual depende de la interfaz del módulo y del uso de datos globales.
3. **De flujo de información:** Parte del supuesto que, el acoplamiento también se incrementa con la complejidad del flujo de información. Tienen en cuenta:
 - La **complejidad intra-modular:** Estimado como el tamaño del módulo en LOC.
 - La **complejidad inter-modular:**
 - Inflow: Flujo de información entrante al módulo.
 - Outflow: Flujo de información saliente al módulo.
 - La **complejidad del diseño del módulo C:** $DC = \text{tamaño} * (\text{inflow} * \text{outflow})^2$ → define la complejidad sólo en la cantidad de información que fluye hacia adentro y hacia fuera y el tamaño del módulo*₁
 - $DC = \text{fun_in} * \text{fun_out} + \text{inflow} * \text{outflow}$
 - fun_in: cantidad de módulos que llaman a C.
 - fun_out: Cantidad de módulos que son llamados por C.

Ahora bien, luego, se usa el promedio de la complejidad de los módulos, y su desviación estándar; para, identificar los módulos complejos, junto a aquellos propensos a errores.

- Si $DC >$ complejidad media + desviación estándar
 - Puede llegar a ser un módulo, muy propenso a errores.
- Si complejidad media $< DC <$ complejidad media + desviación estándar
 - Es complejo.
- Caso contrario
 - Se considera normal.

*₁ también es importante la cantidad de módulos desde y hacia donde fluye la info. En base a esto, el impacto del tamaño del módulo empieza a resultar considerarse insignificante.

Diseño Orientado a Objetos

diseño orientado a objetos se centra en descomponer un sistema en objetos que encapsulan datos y comportamientos. Se basa en los principios de abstracción, encapsulamiento, herencia y polimorfismo. Los objetos son instancias de clases y se comunican mediante mensajes para realizar las funciones del sistema. Este enfoque organiza el software en términos de objetos que representan entidades del mundo real o conceptos del dominio del problema.

- **Clase:** Es una plantilla que define de manera genérica cómo van a ser los objetos de un determinado tipo.
- **Objetos:** Instancias de las clases.
- **Polimorfismo:** La herencia de clases induce polimorfismo, ya que un objeto puede ser de distintos tipos. Un objeto de tipo Y es también un objeto de tipo X si Y es subclase de X.
- **Tipos de relaciones entre objetos**
 1. Asociación: Si un objeto interactúa con otro durante un tiempo determinado.
 2. Agregación: Si un objeto es parte de otra clase, relación derivada de una asociación.
 3. Composición: Si un objeto está compuesto por otro/u otros.
- **Herencia:** La herencia es una relación entre clases que permite la definición e implementación de una clase basada en la definición de una clase existente.
- **Tipos de herencia**
 1. Estricta: Una subclase toma todas las características de su superclase.
 2. No estricta: La subclase redefine algunas de las características.

Tipos de acoplamientos

1. **Acoplamiento por interacción:** Ocurre debido a métodos de una clase que, invocan métodos de otra clase.
 - **Mayor:** Ocurre debido a métodos de una clase que, invocan a métodos de otra clase.
 1. Los métodos acceden partes internas de otros métodos.
 2. Los métodos manipulan directamente variables de otras clases.
 3. La información se pasa a través de variables temporales.
 - **Menor:** Si los métodos se comunican directamente a través de los parámetros.
 - **Nora:** Con el menor número de parámetros posible, pasando la menor cantidad de información posible, pasando solo datos (y no control).
2. **Acoplamiento de componentes:** Ocurre cuando una clase A tiene variables de otra clase C, ya sea porque:
 - a. A tiene variables de instancia de tipo C
 - b. A tiene parámetros de tipo C
 - c. A tiene métodos con variables locales de tipo C.
- 3.
4. Cuando A está acoplado con C, también lo está, con todas sus subclases.
 - **Menor:** Si las variables de clase C en A son, o bien atributos, o bien parámetros en un método, es decir, son visibles.
5. **Acoplamiento de herencia:** Siempre que hagamos herencia, tendremos acoplamiento; pues, dos clases están acopladas si una es subclase de la otra.
 - **Mayor:** Si las subclases modifican la firma de un método o eliminan un método.
 - **Menor:** Si la subclase solo agrega variables de instancia y métodos pero no modifica los existentes en la superclase.

Tipos de cohesión

1. **De método:** ¿Por qué los elementos están juntos en el mismo método?
 - **Mayor:** Si cada método implementa una única función claramente definida con todos sus elementos contribuyendo a ella.
2. **De clase:** ¿Por qué distintos atributos y métodos están en la misma clase?
 - **Mayor:** si una clase representa un único concepto con todos sus elementos contribuyendo a este concepto.
 - **Menor:** Si una clase encapsula múltiples conceptos.
3. **De la herencia:** ¿Por qué distintas clases están juntas en una misma jerarquía?
 - Hay 2 razones para definir subclases: generalización-especificación, y reusó.
 - **Mayor:** generalización-especificación.
 - **Menor:** reusó.

Metodología OMT

Es una metodología de diseño. Estas tienen el fin de, dar pautas para auxiliar al diseñador en el proceso de diseño; aunque esta actividad, no se puede reducir a una secuencia de pasos mecánicos.

OMT está dirigida a, diseño orientado a objetos; y permite usar el modelado obtenido en el análisis OO para, llegar a un modelo detallado final. Object Modeling Technique, consta de los siguientes pasos:

1. **Producir el diagrama de clases:** Usar el modelo realizado, en la etapa de análisis.
2. **Producir el modelo dinámico y usarlo para definir operaciones de las clases:** El modelo dinámico describe el comportamiento temporal del sistema, mostrando cómo los objetos interactúan a lo largo del tiempo y cómo responden a los eventos (solicitudes de operaciones). Permite, agregar al diagrama de clases; aquellas operaciones que, su estructura estática no las deja ver.
3. **Producir el modelo funcional y usarlo para definir operaciones de las clases:** El modelo funcional especifica como computar las salidas a partir de las entradas, sin considerar los aspectos de control. Permitiendo describir, las operaciones existentes en el sistema. Las cuales, en OO se realizan sobre objetos; y deben aparecer en el diagrama de clases (como una operación o múltiples operaciones). Notar que, los transformadores del DFD representan estas operaciones.
4. **Definir las clases internas y sus operaciones:** Inicialmente, las clases se derivan del dominio del problema, pero el diseño final debe reflejar un plan detallado de la implementación, incluyendo aspectos como algoritmos y optimización. Para ello se evalúa críticamente cada clase para determinar si es necesaria en su forma actual, ya que algunas pueden ser descartadas si no son esenciales para la implementación. Luego, se deben considerar las implementaciones de las operaciones para cada clase; pues es posible que se requieran operaciones de bajo nivel en clases auxiliares más simples, como árboles o diccionarios, conocidas como clases contenedoras.
5. **Optimizar y empaquetar:** Se centra en optimizar el modelo mediante:
 - a. Añadir asociaciones redundantes para mejorar el acceso a los datos.
 - b. Guardando atributos derivados, evitando cálculos complejos, y asegurar consistencia.
 - c. Usar tipos genéricos permitiendo, la reutilización del código..
 - d. (importante) Ajustar la herencia, considerando subir en la jerarquía las operaciones comunes, y la generación de clases abstractas por sobre otras; mejorando la reutilización.
 - e. Aplicar los principios de acoplamiento, cohesión, y abierto-cerrado; para mejorar, la calidad del diseño.

Métricas

Weighted Methods for Class

- **WMC:** La complejidad de la clase depende de la cantidad de métodos en la misma ,y su complejidad.

WMC = A la sumatoria de la complejidad de cada método

Si el WMC es alto entonces, la clase es más propensa a errores.

Depth of Inheritance Tree

- **DIT:** El DIT de una clase C es, la longitud del camino de la raíz a ella; para herencia multiple, se toma el camino más largo. Una clase muy por debajo en la jerarquía de clases puede, heredar muchos métodos; lo cual dificulta, la predicción de su comportamiento. A mayor DIT, la clase es más propensa a errores.

Number of Children

- **NOC:** Es la cantidad de subclases inmediatas de, una clase C. A mayor NOC crece el reusó, y aumenta la influencia de C sobre otros elementos de diseño; por lo que, se debe dar mayor importancia a la corrección de esta clase.

Coupling between Classes

- **CBC:** Es la Cantidad de clases a las cuales, la clase C está acoplada. A mayor CBC, hay menor independencia; y esto vuelve a C propensa a errores.

Response For Class

- **RFC:** Captura el grado de conexión de los métodos de una clase C, con respecto a otras clases. Se define como: la cantidad de métodos de C más, los métodos de otras clases receptores de mensajes enviados por métodos de C. Un RFC alto, implica posibles dificultades en el testeo de esa clase. Un alto RFC es muy significativo a la hora de predecir clases propensas a errores.

Diseño

Detallado

PDL: El diseño de alto nivel no especifica la lógica, ya que esto es incumbencia del diseño detallado. Una notación textual provee mayor información, siendo el lenguaje natural ambiguo e impreciso, y los lenguajes formales detallados pero poco útiles para comunicar el diseño. Necesitamos un lenguaje preciso, con pocos detalles, independiente del lenguaje de programación y fácil de implementar. PDL es una solución, con sintaxis externa de un lenguaje funcional y vocabulario en lenguaje natural, capturando la lógica del procedimiento con pocos detalles de implementación. Se usa para especificar el diseño desde la arquitectura hasta el diseño detallado, útil en refinamiento top-down, y ciertas partes son automatizables. Para implementarlo, las pseudo-sentencias deben convertirse en sentencias del lenguaje de programación elegido.

Codificación

Programación estructurada:

Los programas tienen dos estructuras:
La Estructura estática, y dinámica. Aluden al, orden de las sentencias; la 1.^a en el código, y la 2.^a en la ejecución.

La programación estructurada busca, escribir programas donde coincidían ambas estructuras. Es decir, que las sentencias se ejecuten con igual orden al presentado en el código. Lo cual permite comprender el comportamiento dinámico desde la estructura estática. Esto se logra con, constructores no arbitrarios (semántica clara), de única entrada y salida.

Ocultamiento de la información

Las estructuras de datos son ocultadas tras, las funciones de acceso; pudiendo solo ser manipuladas, atrevés ellas. Esto reduce, el acoplamiento.

Proceso de codificación incremental

Enmarcado en la fase de, codificación; más específicamente, en los procesos de codificación. El proceso de codificación incremental está dividido en,3 ideas; escribir el código del módulo; realizar el test de unidad (debiera ser el unico que hace el mismo codificador); y si existe un error, arreglarlo, y ejecutar nuevamente los tests.

Pasos:

1. Tomar un módulo, qué haya sido especificado.
2. Escribir el código, de alguna funcionalidad de este; cumpliendo la especificación.
3. Hacer, los scripts para los tests; a los fines de, testear esa funcionalidad.
4. Se corren los, scripts de tests, recién escritos
 - A. Si existen errores, se corrigen; y retomamos desde, 4.
 - B. Caso contrario,
 - B.1. Chequear sí, se cubrió toda la especificación del módulo.
 - B.A: Caso afirmativo, ^{No} retomamos desde 1.
 - B.B: Caso contrario, ^{Sí es} avanzamos con otro módulo.

Proceso de codificación de desarrollo dirigido por test

Enmarcado en la fase de, codificación; más específicamente, en los procesos de codificación. El TDD, cambia el orden de las actividades asociadas a la codificación. Es decir, en este caso el programador 1.º escribe los scripts para los tests, y luego el código para pasarlo. Además, se realiza incrementalmente.

Es importante tener presente que; aquí debe estar testeada la totalidad de la funcionalidad. Además la completitud del código depende, de cuan exhaustivo sean los tests.

Pasos:

1. Tomar un módulo, qué haya sido especificado.
2. Hacer, los scripts de tests; para una funcionalidad del módulo seleccionado.
3. Construir el código, para pasar los scripts de tests., recién escritos
4. Se corren los scripts de tests, recién escritos
 - A. Si existen errores, se corrigen; y luego se retoma desde, 4.
 - B. Caso contrario, chequear si el código necesita mejoras. (muchas veces una refactorización)
 - A. Caso afirmativo, se realizan, y luego se retoma desde 4.
 - B. Caso contrario
 - A. Si Se completó la especificación del módulo, pasar a otro.
 - B. Caso contrario, retomar desde 1.

Programación de a pares

Enmarcado en la fase de, codificación; más específicamente, en los procesos de codificación. La programación de a pares consiste en que, dos programadores colaboran estrechamente para escribir el código. Juntos, diseñan algoritmos, estructuras de datos y estrategias, identifican errores y formulan soluciones. Mientras uno codifica, el otro revisa activamente el código, alternando roles periódicamente. Este modelo asegura una revisión continua del código, mejorando el diseño de algoritmos, estructuras de datos y lógica, y reduciendo la probabilidad de que se escapen condiciones particulares. Sin embargo, la efectividad de este método en términos de productividad aún no está bien determinada.

Refactorización

La refactorización es la tarea que permite realizar cambios en un programa con el fin de simplificarlo y mejorar su comprensión. Además, busca mejorar el código.

La técnica de refactorización pertenece, a la fase de codificación, y se aplica sobre un programa existente en funcionamiento. Busca hacer un código más, testeable y mantenible. Modificando su estructura interna pero, manteniendo intacto el comportamiento externo. Intenta lograr una o más de las siguientes cosas: reducir acoplamiento, aumentar la cohesión, y mejorar el principio abierto cerrado.

refactorizar

El principal riesgo de esta tarea es, romper el funcionamiento existente. Para disminuir esta posibilidad, ~~de rectoría~~ de a pequeños pasos, y al final de c/u se corren scripts de test automatizados que testean las funcionalidades existentes.

Refactorizaciones más comunes:

Siempre con el objetivo de, reducir acoplamiento, aumentar la cohesión, y mejorar el principio abierto cerrado.

- Métodos
 - Extracción de método
 - Si es largo, se separa en otros más cortos donde con solo leer sus signatures sepamos qué hacen.
 - Si retorna un valor y también modifica el estado del objeto, se divide en 2 métodos.
 - Agregar o eliminar un parámetro
 - Agregar solo si los parámetros existentes no proveen la información que se necesita.
 - Eliminar si los parámetros no se utilizan.
- Clases
 - Desplazamiento de métodos ^{demasiado}
 - Si el método interactúa con los objetos de la otra clase, inicialmente puede ser conveniente dejar un método en la clase inicial que delegue al nuevo (debería tender a desaparecer).
 - Desplazamientos de atributos
 - Si un atributo se usa más en otra clase, moverlo a esta clase.
 - Extracción de clases
 - Si una clase agrupa múltiples conceptos, separa cada c/u en una clase distinta.
 - Reemplazar valores de datos por objetos:
 - Si una colección de atributos se transforma en una entidad lógica, separarlos como una clase y definir objetos para accederlos.
- Jerarquía de clases
 - Reemplazar condiciones con polimorfismos
 - Si el comportamiento depende de algún indicador de tipo, remplazar el análisis por casos atreves de una jerarquía apropiada.
 - Subir métodos o atributos
 - Si la funcionalidad o atributo este duplicado en las subclases, pueden subirse a la superclase.

Proceso de Software

CONCEPTOS IMPORTANTES

PRODUCTO DE TRABAJO:

Las componentes del proceso de software; suelen dividirse en fases. Las cuales, hacen una tarea bien definida, y retorna un “producto del trabajo”; entidad, formal, tangible, y verificable.

ÍTEMES:

Un proyecto de software produce diversos ítems, una unidad de modificación intensiva.

El “Proceso de Software” se descompone en dos grandes áreas: el “Proceso de la Ingeniería del Producto” y el “Proceso de Administración del Proceso”. Dentro del “Proceso de la Ingeniería del Producto”, se encuentran el “0. Proceso de Desarrollo”, “1. Proceso de Administración del Proyecto”, “2. Proceso de Inspección” y “3. Proceso de Administración de Configuración”. Además, el “Proceso de Administración del Proceso” se subdivide en el “4. Proceso de Administración de Cambios” y el “5. Proceso de Administración del Proceso”, indicando una estructura jerárquica y organizada para gestionar el desarrollo y la administración del software.

Enfoque ETVX

Las componentes del proceso de software suelen dividirse en fases. Estas siguen el enfoque ETVX.

Criterio de entrada: (Entry)

- Condiciones a cumplir para, iniciarla.
- ~~Lo que debe hacer~~

*₂

- Verificación: (Verification)

- Las inspecciones/controles/revisiones/verificaciones a realizar sobre su, producto de trabajo.

- Criterio de salida: (Exit)

- Cuando puede considerarse, una finalización exitosa de la misma.

- Además cada fase produce, información para la administración del proceso.

*₂

Tarea: (Task)

- Lo que debe realizar esa fase

Proceso de administración de cambio de requerimientos

Dado que, los requerimientos pueden cambiar en cualquier momento durante el desarrollo. Estas modificaciones impactan en, los ítems de configuración y productos de trabajo. Ergo, no controlarlos puede afectar negativamente al proyecto, tanto en costo como en tiempo.

Esta componente del, proceso de software; permite realizar los cambios controladamente. El proceso consiste en:

1. Registrar los cambios
2. Realizar análisis de impacto sobre, los productos de trabajo, e ítems.
3. Estimar el impacto en, esfuerzo y el cronograma.
4. Analizar el impacto con las personas involucradas.
5. Reprocesar, los productos de trabajo, e ítems.

Las modificaciones se inician por, un requerimiento de cambio; estos son almacenados en, un registro. El análisis de impacto para estos requerimientos, incluye identificar los cambios necesarios en los distintos ítems y la naturaleza del cambio. El impacto del cambio en el proyecto es analizado para decidir, si hacerlo efectivo o no. Además, los cambios acumulativos también se registran.

Proceso de administración de la configuración (SCM)

Un proyecto de software produce diversos ítems, cada uno es una unidad de modificación intensiva. La SCM controla sistemáticamente los cambios en ellos, enfocándose en los relativos al proceso de desarrollo, exceptuando los vinculados a requerimientos.

Un ítem se elabora, y cuando creemos que el desarrollo está satisfecho, pasa a la etapa de revisión. Si es aceptado, entra en la administración de la configuración.

Durante el proceso, los ítems cambian, generando diferentes versiones que deben ser combinadas periódicamente sin pérdidas por la SCM. Esta última tarea tiene un concepto asociado denominado "baseline", que es el arreglo apropiado de ítems de configuración. Establecen puntos de referencia a lo largo del desarrollo del sistema, capturan el estado lógico del sistema y forman la base de los cambios posteriores.

Las funcionalidades necesarias son:

- Recolectar todas las fuentes, documentos y otra información del sistema actual.
- Evitar cambios o eliminaciones desautorizadas.
- Deshacer cambios o revertir a una versión especificada.
- Hacer disponible la última versión del programa (la última que funcionó).

Mecanismos principales:

- Control de acceso: Limita el acceso a personal específico con procedimientos de check-in y check-out.
- Control de versiones: Ayuda a preservar viejas versiones y deshacer cambios. No solo aplica al código fuente.
- Identificación de la configuración: Otros mecanismos incluyen: convenciones de nombres, estructuras de directorios, etc.

Fases:

1. **Definir las actividades que requieren control de cambio.**
2. **Planeamiento:**
 - Identificar ítems.
 - Definir la estructura del repositorio.
 - Definir control de acceso, puntos de referencia, reconciliación, procedimientos.
 - Definir procedimiento de publicación.
3. **Ejecución:**
 - Realizar los procedimientos según lo establecido en el planeamiento.
4. **Auditoría:**
 - Verificar que no se cometieron errores, por ejemplo, que se mantiene la integridad y que los requerimientos de cambio se realizaron apropiadamente.

Proceso de administración de procesos

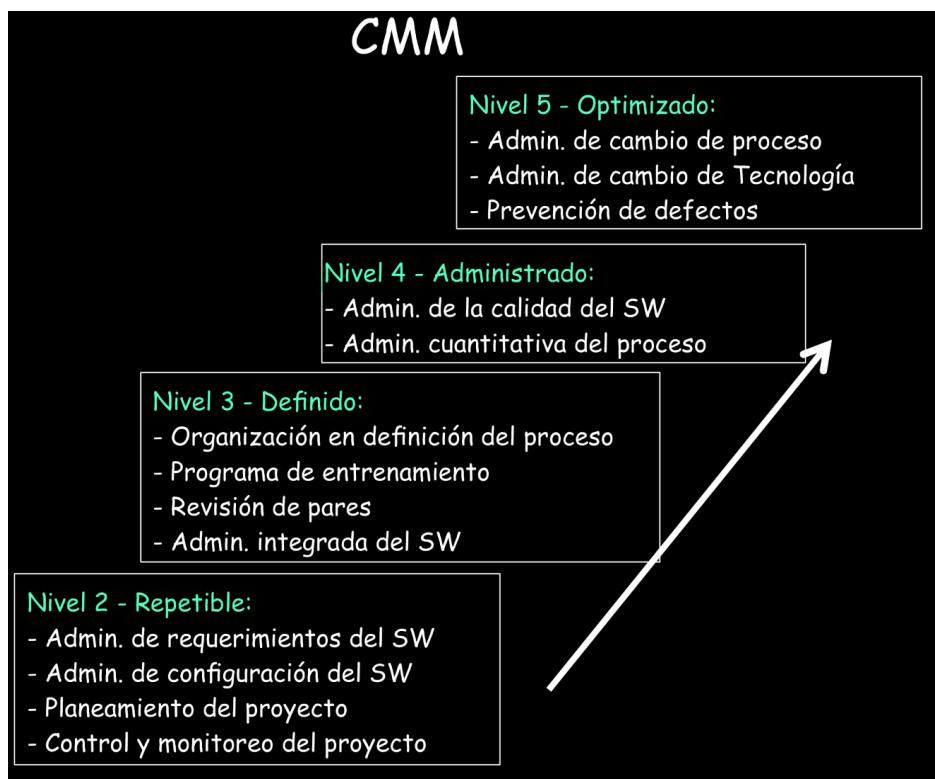
Un proceso no es una entidad estática; este debe cambiar para mejorar la calidad y productividad.

El proceso de administración de procesos es una componente del proceso de software. Se enfoca en la evaluación y mejora de los procesos, a diferencia de la administración del proyecto, la cual aborda, como su nombre lo indica, al proyecto.

Para que una organización mejore el proceso, debe poder comprenderlo, documentarlo sólidamente, ejecutarlo apropiadamente y recolectar los datos que permitan comprender su desempeño en los proyectos.

Por otro lado, es mejor que los cambios a los procesos se hagan incrementalmente, de a pasos pequeños, porque si fallan, es difícil revertirlos. Cada paso debe ser cuidadosamente seleccionado, decidiendo qué cambios hacer y cuándo.

También existen marcos que ayudan en la mejora de procesos, como el CMM. Este tiene 5 niveles, donde el nivel 1 es ad hoc. Para pasar al siguiente nivel, el proceso debe cumplir con capacidades específicas y proveer las bases para el próximo elemento jerárquico.



Proceso de inspección

Proceso miembro de las componentes del proceso de software. Enfocada en, detectar los defectos en todos los tipos de producto de trabajo; pero no, resolverlos. Se estructura con roles y responsabilidades, definidos para cada participante:

- **Moderador:** Quien tiene la responsabilidad general.
- **Autor:** Quien realizó el producto de trabajo.
- **Revisor:** Quien identifica los defectos. En general es, un grupo.
- **Lector:** Lee línea a línea el producto de trabajo para, enfocar el progreso de la reunión. Es relativo; puede no existir como ente físico pero, si como rol.
- **Escríba:** Quien registra las observaciones indicadas.

Sus pasos son:

1. **Planeamiento**
 - a. Identificar al moderador, quien tiene la responsabilidad principal. En general es quien se encarga de realizar los ítems que siguen dentro de este paso.
 - b. Seleccionar el equipo de revisión.
 - c. Se prepara el paquete para la distribución. Lo distribuido al revisor contiene, que es lo que se va a revisar:
 - El producto de trabajo afectado, y sus especificaciones.
 - Listas de control con ítems relevantes,
 - Estándares.
2. **Preparación y repaso previo (overview).**
 - a. Pequeña reunión opcional entre, el moderador, y los revisores; donde:
 - Se entrega el paquete.
 - Se explica el propósito de la reunión.Luego,
 - Se introduce, brevemente, las áreas de cuidado.
 - b. Idealmente en a lo sumo 2 hs, y de corrido; c/u de los revisores individualmente, revisan el producto de trabajo. Identificando defectos potenciales en un registro individual; usando checklists, pautas, y estándares.
3. **Reunión de revisión grupal**

Busca definir la lista final de defectos. Supone que, cada revisor realizó apropiadamente, la revisión individual; las cuales son revisados por el moderador.

La reunión se rige por estas directrices:

 - i. El lector lee línea a línea el producto de trabajo.
 - ii. En cualquier línea, cualquier observación que hubiere (preparada o nueva) es efectuada.
 - iii. Se sigue una discusión para identificar el defecto.
 - iv. La decisión es registrada por el escríba.

Al final de ella, el escríba presenta la lista de defectos/ observaciones. Si son pocos, el producto de trabajo es aceptado; caso contrario, puede requerir otra revisión. Si bien se pueden registrar sugerencias, el grupo no debe proponer soluciones. Por último, se prepara un resumen de las inspecciones; usado para evaluar la efectividad de la inspección.
4. **Corrección y seguimiento**

Los elementos de la lista de defectos/observaciones son, corregidos por el autor. Para luego obtener el visto bueno del moderador, o ir a una nueva revisión. Una vez los defectos/observaciones sean resuelto satisfactoriamente, se concluye el proceso.

Proceso para la administración del proyecto

Con el fin de, ejecutar eficientemente cada fase y actividad del proceso de desarrollo. El proceso para la administración del proyecto debe, asignarle recursos y administrarlos, ver el progreso, tomar acciones correctivas, etc.

Fases:

- **Planeamiento**
Previo al, proceso de desarrollo; estima los costos y tiempos, selecciona el personal, planea el seguimiento, planea el control de la calidad, etc. Retorna un plan, usado como base del seguimiento.
- **Seguimiento y control**
Durante todo el, proceso de desarrollo; Sigue y observa parámetros claves como costo, tiempos, riesgo, y los factores de influencias o estos. Tomando acciones correctivas, Si es necesario.

Las métricas dan la información del proceso de desarrollo necesaria usada en esta etapa.

- **Análisis de terminación**
Al finalizar él, proceso de desarrollo; analiza el desempeño de este, e identifica los errores cometidos y como cambiarlos.

En procesos iterativos el análisis de terminación se realiza al finalizar cada iteración y se usa para mejorar en iteraciones siguientes.

Procesos de Desarrollo

Componente del proceso de software

Cada modelo de proceso de desarrollo, realizan las siguientes fases; de distintas maneras, y en diferentes órdenes:

- Análisis de requerimientos y especificación:
 - Comprende el problema, y especifica “el que” no “el cómo” de este.
 - Producto de trabajo:
 - Especificación de los requerimientos del sistema (SRS).
- Diseño: (entra al “como”)
 - 3 tipos:
 - Arquitectónico:
 - Establece las componentes y los conectores, constitutivos del sistema.
 - De alto nivel:
 - Establece los módulos y estructuras de datos, necesarios para implementar la arquitectura.
 - Detallado
 - Establece la lógica de, los módulos.
 - Producto del trabajo:
 - Documentación correspondiente a c/u de los diseños.
- Codificación:
 - Transforma al diseño en, un código simple y fácil de comprender (legible).
 - Producto de trabajo:
 - Código
- Testing
 - Identificar y eliminar, la mayoría de los defectos; aumentando la calidad.
 - Producto de trabajo:
 - El plan del test, conjuntamente con los resultados.
 - Código final, testeado y confiable.

Por último, la entrega e instalación.

¿Por qué se divide en fases?

En primer lugar, para aplicar la técnica de, dividir y conquistar. Además, permite que, cada etapa ataca distintas partes del problema. Por último ayuda a, validar continuamente el proyecto.

Cascada

Cascada es, un modelo de proceso de desarrollo. Divide el proyecto en fases estrictamente secuenciales con incumbencias separadas entre ellas, un proceso de verificación en cada una, y entregas entre etapas consecutivas.

Primero, se analiza la factibilidad de resolver el problema planteado, y si es viable, realizamos la siguiente secuencia:

1. Análisis y especificación de requerimientos.
2. Diseño de alto nivel.
3. Diseño detallado.
4. Codificación.
5. Testing.
6. Instalación.

Finalmente, se procede a operar el software y mantenerlo.

Los productos de trabajo usuales en este modelo son:

- Documento de requisitos / SRS.
- Plan del proyecto.
- Documentos de diseño (arquitectura, sistema, diseño detallado).
- Plan de test y reportes de test.
- Código final.
- Manuales del software.
- Además, de reportes para cada una de las fases.

	Fortalezas	Debilidades	Aplicación
	Simple. Fácil de ejecutar. Intuitivo y lógico.	"Todo o nada": muy riesgoso. Req. se congelan muy temprano. Puede escoger hw/ tecno. vieja. No permite cambios. No hay feedback del usuario.	Problemas conocidos. Proyectos de corta duración. Automatización de procesos manuales existentes.

Cascada con feedback

Cascada con feedback es, un modelo de proceso de desarrollo. Consiste en una cascada típica, donde cada fase puede volver exclusivamente a la inmediatamente anterior y luego regresar. Esto permite manejar de mejor manera los cambios de requerimientos en el proceso de desarrollo.

Prototipado

Prototipado es, un modelo de proceso de desarrollo. Ataca el riesgo de requerimientos, problemático en cascada. Generando un prototipo que debe ser descartados; para dilucidar, requisitos poco claros. Dando al cliente, una idea más tangible, del software.

Se logra, manteniendo la estructura de cascada; pero la 1.^a etapa se cambia por, una repetición regenerativa y retroalimentada por el cliente, de todas las faces pero con testing liviano. Iterando hasta entender todos los requisitos, o si costo tiempo supera a los beneficios. Luego se descarta el prototipo, y se sigue con la secuencia normal.

Fortalezas	Debilidades	Aplicación
Ayuda a la recolección de requerimientos. Reduce el riesgo de req. mal comprendidos. Suele lograr sistemas finales más estables.	Comienzo pesado. Posiblemente mayores costos y tiempos. No permite cambios tardíos.	Sistemas con usuarios novatos. Cuando hay mucha incertidumbre en los requerimientos. Cuando las interfaces con el usuario son muy importantes.

Desarrollo iterativo

El desarrollo iterativo es, un modelo de proceso de desarrollo. Ataca el problema de "todo o nada" presente en el modelo en cascada, aplicando un desarrollo y entrega de software incremental, donde cada incremento es completo en sí mismo y su feedback puede ser usado en futuras iteraciones.

Primero se crea la LCP, una lista ordenada con las tareas necesarias para lograr el código final; cada una de ellas es suficientemente completa para ser toda una iteración, y tan simple que se pueda comprender en su totalidad. Cada paso iterativo consiste en, bajo el orden dado por la LCP, eliminar el siguiente ítem haciendo diseño, implementación, análisis del sistema parcial y actualizar la lista. Por último, se itera hasta vaciar la LCP.

Fortalezas	Debilidades	Aplicación
Entregas regulares y rápidas. Reduce riesgo de req. mal comprendidos. Acepta cambios naturalmente. Permite feedback del usuario. Prioriza requisitos.	Sobrecarga de planeamiento en cada iteración. Posible incremento costo total (trabajo en una iteración puede deshacerse en otra). Arq. y diseño pueden ser afectados con tantos cambios.	Para empresas donde el tiempo es esencial. Donde no puede enfrentarse el riesgo de proyectos largos. Cuando los requerimientos son desconocidos y sólo se comprenderán con el tiempo.

Timeboxing

Es, un estilo del modelo de proceso de desarrollo iterativo. En Timeboxing se realiza el desarrollo en iteraciones temporalmente iguales (time boxes), ejecutadas en paralelo mediante pipelining. Cada time box se divide en, etapas fijas de aproximadamente igual duración; las cuales realizan una tarea definida, son independientes entre sí, y c/u tiene asignada un equipo de trabajo.

Fortalezas	Debilidades	Aplicación
Todas las fortalezas del iterativo. Planeamiento y negociación un poco más fácil. Ciclo de entrega muy corto.	La administración del proyecto es compleja. Es posible el incremento de los costos. Equipos de trabajo muy grandes.	Donde es necesario tiempos de entrega muy cortos. Hay flexibilidad en agrupar características (features).

Testing

Desperfecto y defecto (bug)

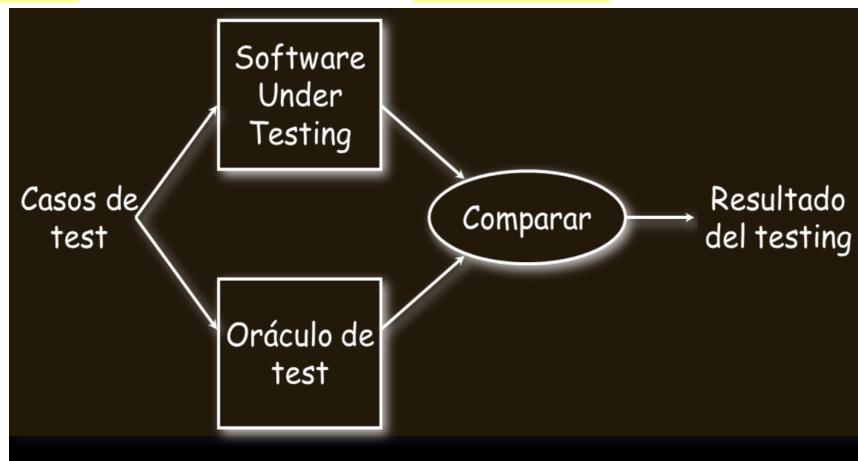
En un proyecto, se busca principalmente; desarrollar un producto con alta, calidad y productividad. La 1.^a calidad tiene diversas aristas, siendo confiabilidad una de las más fuertes. La cual habla sobre, la posibilidad que tiene un software de fallar. Ergo, la calidad persigue proveer un producto con, la menor cantidad de defectos como sea posible; pues a mayor cantidad de ellos, baja la confiabilidad.

Un desperfecto de software, ocurre si el código se comporta distinto a lo esperado/especificado en el proyecto; es observable, y su existencia implica la de un bug; pues este se define como, su causa, pero; poseer el 2.^º no deriva necesariamente en, la ocurrencia del 1.^º, aunque tenga el potencial de generarlo.

Por otro lado, las revisiones realizadas al final de cada fase, no detectan todos los defectos. Para asegurar calidad, los que persisten deben identificarse mediante el testing. Durante esta etapa, un programa se ejecuta siguiendo un conjunto de casos de test (hay una ejecución del sistema); si aparecen desperfectos al correrlos, entonces, el software tiene bugs; caso contrario, sube la confianza, pero no se puede descartar la presencia de defectos. Su objetivo final es, producir desperfectos; para luego recurrir al debugging permitiendo hallar los defectos detonantes de estos.

Oráculos de tests:

Para verificar la ocurrencia de un desperfecto en la ejecución de un caso de test, necesitamos conocer el comportamiento esperado, siendo el oráculo quien nos lo brinda. Permitiendo el siguiente flujo de trabajo:



Si bien, idealmente debe dar el resultado correcto; ~~muchas~~ veces es un humano quien cumple este rol, por lo que se vuelve propenso a cometer errores. Además, usa la especificación para decidir el comportamiento correcto; pero esta, también puede presentar fallos.

En algunos casos, los oráculos pueden generarse automáticamente desde la especificación. Que aunado a la automatización de los test, logra automatizar buena parte del testing.

Criterios de selección de tests

Necesitamos que la ejecución satisfactoria del conjunto de casos de test implique ausencia de defectos; y debido al costo del testing, queremos un set reducido. Ambas condiciones son contradictorias, de esta contradicción nace el concepto de criterios de selección de casos de test. Este especifica las condiciones que el conjunto de casos de test debe satisfacer con respecto al programa y/o a la especificación.

Propiedades fundamentales de los criterios de selección de test

- **Confiabilidad:** Un criterio es confiable si todos los conjuntos de casos de test que satisfacen el criterio detectan los mismos errores.
- **Validez:** Un criterio es válido si para cualquier error en el programa hay un conjunto de casos de test que satisfagan el criterio y detecten el error.

Es prácticamente imposible obtener un criterio que sea confiable y válido al mismo tiempo, y que también sea satisfecho por una cantidad manejable de casos de test.

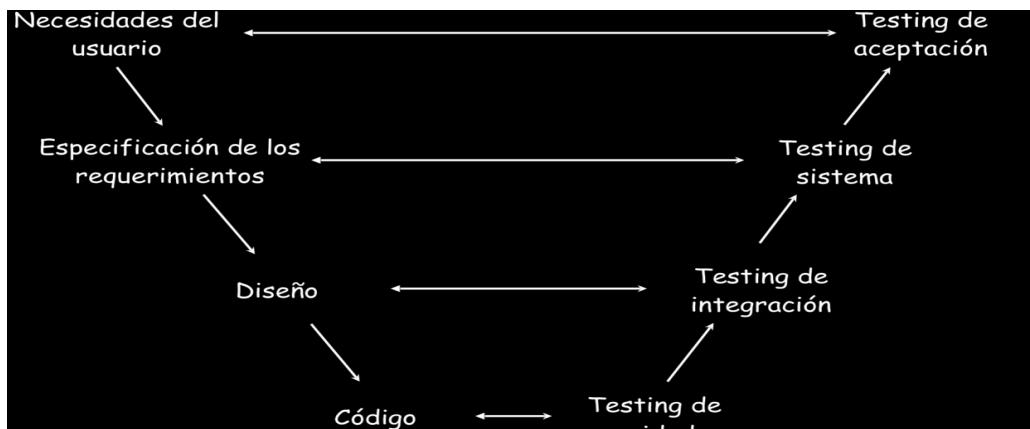
Enfoques

Existen dos grandes tipos de testing: caja negra o funcional, y caja blanca o estructural. Es importante mencionar que ambos son complementarios.

- **Caja negra:** Trabaja dándole una entrada a la implementación bajo testeo y analizando la salida. Supone la ausencia de la estructura interna del código a testear, pero sí considera la especificación de su comportamiento esperado. Los tests se definen solo para el comportamiento esperado especificado. En testing de módulos, esto está dado por la especificación producida en el diseño; en el caso del testing de sistema, la SRS lo provee. El testing funcional más minucioso es el exhaustivo, donde se prueba el IUT con todo el espacio de entradas. Sin embargo, este es inviable o imposible debido al alto costo. Por esto, se necesitan mejores métodos para seleccionar los casos de test.
- **Caja blanca:** Persigue el objetivo de ejecutar las distintas estructuras del programa con el fin de encontrar errores. Debido a que aquí se parte de disponer del código como supuesto fundamental, los casos de test son derivados de este.
- **Comparación:** Ambas técnicas son complementarias. La caja blanca es buena para detectar errores en la lógica del programa (errores estructurales) y es útil a bajo nivel, donde el programa es manejable; suele ser usada en "partes chicas" como el testing de unidad. Mientras que la caja negra es útil si queremos detectar errores en entrada/salida (errores funcionales) y es útil a alto nivel, donde se busca analizar el comportamiento de todo un sistema.

Niveles de testing:

El código contiene defectos de, requerimientos, diseño, codificación, etc.; es decir, tiene niveles, y la naturaleza de los defectos para c/u de ellos es diferente. Un solo tipo de testing sería incapaz de detectar, está gran variedad de tipos de defectos. Por lo tanto, se utilizan distintos niveles de testing para resolver diferentes tipos de defectos.



Test de unidad: Los distintos módulos del programa, separadamente contra el diseño; el cual, actúa como especificación del módulo. Se enfoca en, los defectos inyectados durante la codificación; persiguiendo el objetivo de, testear la lógica interna del módulo. Frecuentemente, el mismo programado es, quien lo efectúa.

Testing de Integración: Se enfoca en, la interacción entre módulos de un sistema. Los módulos que han pasado por testing unitario, se combinan para formar subsistemas sujetos a testing de integración. Los casos de test deben, generarse con el objetivo de ejercitarse de diferente manera la interacción entre los módulos; chequeando que para todas las interacciones posibles haya un test de integración. Se hace énfasis en el testeо de las interfaces entre módulos; y a veces es omitido cuando el sistema no es muy grande.

Testing de sistema: El sistema de software completo, es testeado. Se enfoca en, verificar si el software implementa los requerimientos; realiza el ejercicio de validar el sistema con respecto a los requerimientos. Generalmente, es la etapa final del testing antes de que el software sea entregado. Debería ser realizado por, personal independiente; pero, los defectos son eliminados por los desarrolladores.

Testing de aceptación: Se enfoca en, verificar si el software satisface las necesidades del usuario. Generalmente, se realiza con el usuario o cliente en su entorno con datos reales; pero los defectos son eliminados por, los desarrolladores. Solo después de, considerarse satisfactorio el testing de aceptación; el software, se puede considerar en ejecución.

Plan del test

Un plan de pruebas es fundamental para garantizar que un sistema cumpla con sus requisitos y funcione correctamente. Sirve como una guía detallada para todo el proceso de testing, asegurando que todas las áreas críticas del sistema sean evaluadas de manera sistemática y eficiente.

1. **Definición del objetivo del test:** Establece los objetivos específicos y el alcance del testing, indicando lo que se pretende verificar en el sistema.
2. **Identificación de recursos:** Determina los recursos necesarios, incluyendo el personal, el hardware y software necesarios para llevar a cabo las pruebas.
3. **Criterios de inicio y finalización:** Define los criterios para empezar y terminar las pruebas, especificando qué condiciones deben cumplirse para considerar que las pruebas han sido exitosas.
4. **Establecimiento del entorno de prueba:** Describe el entorno en el cual se realizarán las pruebas, incluyendo la configuración del sistema y cualquier software o hardware adicional necesario.
5. **Planificación del test:** Incluye la programación y la asignación de tareas específicas, detallando cuándo y cómo se llevarán a cabo las pruebas.
6. **Identificación de riesgos y contingencias:** Enumera los posibles riesgos y problemas que pueden surgir durante las pruebas y describe las estrategias para mitigar estos riesgos.
7. **Documentación y reporte de resultados:** Define cómo se documentarán los resultados de las pruebas y cómo se informarán los hallazgos a las partes interesadas.

Este plan busca asegurar que todas las áreas críticas del sistema sean probadas de manera exhaustiva y que los resultados sean precisos y útiles para la toma de decisiones futuras sobre el desarrollo del software.

CAJA NEGRA

Particionado por clases de equivalencia (Métodos de selección de tests, caja negra))

El testing funcional mas minucioso es, el exhaustivo. Donde, se testea el IUT con todo el espacio de entradas. Pero este es inviable o imposible debido, al alto costo. Por esto, se necesitan mejores métodos para seleccionar los casos de test.

Este método divide el espacio de entradas en conjuntos disjuntos; tales que, no exista elementos sin pertenecer a uno de ellos. Cada clase de equivalencia será un subconjunto del inicial, donde, todos sus elementos deben tener igual comportamiento. Obtener la división ideal, es imposible; por ende, nos aproximamos identificando las clases que especifican distintos comportamientos. Para mejorar la robustez, se arman sets con inputs válidos, y otros de inválidos.

Este tipo de fragmentado, nos permite tener los siguientes supuestos

- Si un elemento de una clase falla, suponemos que todos sus miembros también.
- Si un elemento de una clase no falla, suponemos que todos sus miembros tampoco

Formas de seleccionar los casos de test

1. Seleccionar cada caso de test cubriendo tantas clases como sea posible.
2. Un caso de test que cubra a lo sumo una clase válida por cada entrada. Más uno por cada clase inválida.

- OBSERVACIÓN: SE PUEDE HACER LO MISMO CON EL ESPACIO DE DATOS DE SALIDA.
- Ejemplo Inventado: Dado un software con 2 inputs: una lista de números "l" con como máximo N elementos, y un entero "n". El programa retorna, los "n" números más repetidos en la lista "l"; donde sus elementos solo son enteros de, 8, 16, 32, o 64 bits. Se trabaja separadamente, a los distintos tipos de enteros.

Inputs	Clases de Equivalencias Válidas	Clases de Equivalencias Inválidas
l	1. Contiene enteros de 8 bits. 2. Contiene enteros de 16 bits. 3. Contiene enteros de 32 bits 4. Contiene enteros de 64 bits 5. Longitud de la lista $\leq N$	1. Números no enteros. 2. Longitud de la lista $> N$
n	6. Está en el rango válido.	3. Esta fuera del rango válido

Forma 1: La lista "l" con longitud $\leq N$, conteniendo los 4 tipos de datos; junto con, "n" igual a 4. Más un caso de test por cada clase de equivalencia inválida. Total: $1 + 3 = 4$ casos de test.

Forma 2: Una lista aparte por cada caso de test (uno para enteros de 8 bits, uno para enteros de 16 bits, etc.). Más, los casos inválidos. Total: $5 + 3 = 8$ casos de test

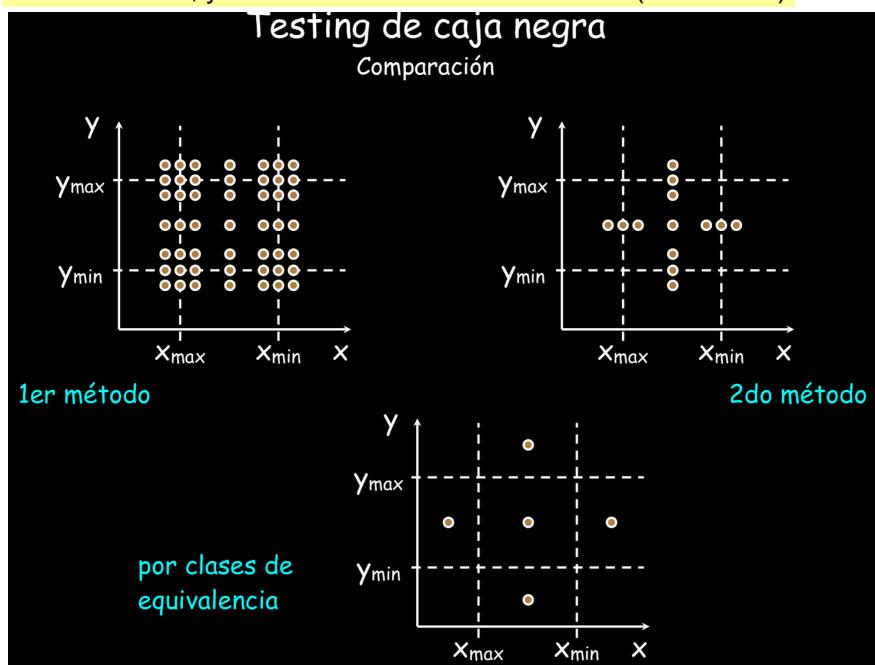
Análisis valores límite (Métodos de selección de tests, caja negra):

El testing funcional más minucioso es, el exhaustivo. Donde, se testea el IUT con todo el espacio de entradas. Pero este es inviable o imposible debido, al alto costo. Por esto, se necesitan mejores métodos para seleccionar los casos de test.

A partir del, particionado por clases de equivalencias; el cual, parte al espacio de entradas en conjuntos disjuntos; tales que, no exista elementos sin vivir en uno de ellos. Se demostró que, los programas generalmente fallan en los valores bordes entre estos sets. Entonces, se trabaja con casos de tests de valores límite; donde c/u es, un conjunto de datos de entrada que habita la frontera de las clases de equivalencias de la entrada o la salida.

Para cada set, se toma valores justo fuera y dentro de los límites de él; junto con, un valor normal. 1.º se determinan los valores a utilizar para cada variable; si la entrada tiene un rango definido entonces, tenemos 6 valores de límite, más uno normal (7 en total); en el caso de múltiples entradas, existen 2 estrategias para combinarlas en un caso de test:

1. Ejecutar todas las combinaciones posibles de las distintas variables (para n inputs, tenemos 7^n casos de test!).
2. Tomar los casos límites para una variable, manteniendo las otras en casos normales, y considerar el caso de todo normal (total $6n + 1$).



Grafo, causa y efecto (Métodos de selección de tests, caja negra):

El testing funcional más minucioso es, el exhaustivo. Donde, se testea el IUT con todo el espacio de entradas. Pero este es inviable o imposible debido, al alto costo. Por esto, se necesitan mejores métodos para seleccionar los casos de test.

Particionado en clases de equivalencia, y análisis de valores límite abordan cada conjunto de datos de entrada por separado, lo que requiere la ejecución de múltiples combinaciones de clases para distintos escenarios. Esto deriva en una cantidad significativa de combinaciones.

El Grafo de causa-efecto permite, seleccionar estas combinaciones como condiciones de entrada. Un efecto, es una condición que puede ser considerada verdadera o falsa para las salidas; su homólogo para las entradas se denomina (causa). La técnica busca identificar, cuáles causas, pueden detonar qué efectos. Para ello, se construye un grafo con, causas y efectos como nodos, y aristas que determinan dependencia (positiva o negativa); además de, vértices "and" u "or" para combinar la causalidad. Esta representación permite, armar una tabla de decisión; la cual, lista las condiciones que hacen efectivo cada efecto; de esta, se pueden derivar los casos de test.

ENUNCIADO

- Una base de datos bancaria que permite dos comandos:

Acreditar una cantidad en una cuenta.
Debitar una cantidad de una cuenta.

- Requerimientos:

Si se pide acreditar y el número de cuenta es válido => acreditar.
Si se pide debitar, el número de cuenta es válido, y la cantidad es menor al balance => debitar.
Si el comando es invalido => mensaje apropiado.

IDENTIFICAR CAUSAS Y EFECTOS

- Causas:

C1: el comando es "acreditar"
C2: el comando es "debitar"
C3: número de cuenta es válido
C4: la cantidad es válida

- Efectos:

E1: Imprimir "Comando inválido"
E2: Imprimir "Número de cuenta inválida"
E3: Imprimir "Cantidad débito inválida"
E4: Acreditar en la cuenta
E5: Debitar de la cuenta

GRAFO GENERADO

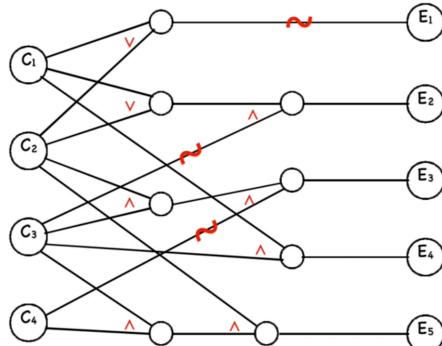


TABLA DE DECISIÓN

#	1	2	3	4	5	6
C1	0	1	X	X	X	1
C2	0	X	1	1	1	X
C3	X	0	0	1	1	1
C4	X	X	X	0	1	X
E1	1					
E2		1	1			
E3				1		
E4					1	
E5						1

Testing de a pares (Métodos de selección de tests, caja negra):

El testing funcional más minucioso es, el exhaustivo. Donde, se testea el IUT con todo el espacio de entradas. Pero este es inviable o imposible debido, al alto costo. Por esto, se necesitan mejores métodos para seleccionar los casos de test.

Usualmente, el comportamiento de un sistema está determinado por múltiples parámetros. Algunos fallos pueden surgir debido a una única condición (modo simple), lo que requeriría m casos de prueba para cubrirlos si tenemos n parámetros, cada uno con m valores.

Sin embargo, ciertos defectos son desencadenados por combinaciones de condiciones (modo múltiple) y son detectados mediante casos de prueba que incluyan las combinaciones adecuadas. La cantidad de pruebas combinatorias puede ser excesiva. Afortunadamente, se demostró que la mayoría de los test combinatorios pueden ser reflejados con situaciones de pares de valores de entrada (modo doble); el testing de a pares se sustenta en este principio.

En esta técnica, es necesario ejecutar todos los pares de valores. Es decir, si tenemos n parámetros con m valores cada uno, el número total de casos de prueba sería $m * n$. Cada caso de prueba implica un conjunto de valores para los n parámetros. El conjunto más pequeño de pruebas se logra cuando cada par es cubierto solo una vez, lo que da como resultado m^2 casos de prueba únicos que proporcionan una cobertura completa.

Ejemplo: Consideremos un sistema de control de acceso con tres parámetros: tipo de usuario (normal, administrador), método de autenticación (contraseña, huella dactilar) y nivel de acceso (básico, avanzado). Si cada parámetro tiene dos valores (binario), la técnica de pares de valores implica realizar pruebas para todas las combinaciones posibles de pares. En este caso, se necesitarían $2 * 2 = 4$ pruebas en total para cubrir todas las combinaciones, lo que garantizaría una cobertura exhaustiva para detectar posibles fallos desencadenados por combinaciones específicas de valores.

Testing basado en estados (Métodos de selección de tests, caja negra):

El testing funcional más minucioso es, el exhaustivo. Donde, se testea el IUT con todo el espacio de entradas. Pero este es inviable o imposible debido, al alto costo. Por esto, se necesitan mejores métodos para seleccionar los casos de test.

El testing basado en estados sirve para, sistemas donde; su comportamiento depende, de la entrada, y el impacto acumulado de las entradas pasadas (estado del sistema).

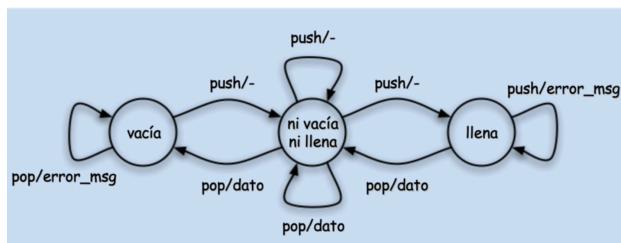
Este tipo de sistemas pueden modelarse como, máquinas de estados. Donde cada estado sea representativo en términos lógicos para el sistema, pero es lo suficientemente pequeño para modelarlo. Este modelo muestra, la en entrada al estado, la salida producida, y el proco estado. Esta representación se puede construir a partir de, la especificación o el diseño.

Componente modelo de estados

- Un conjunto de estados: son estados lógicos representando el impacto acumulativo del sistema.
- Un conjunto de transiciones: representa el cambio de estado en respuesta a algún evento de entrada.
- Un conjunto de eventos: son las entradas al sistema.
- Un conjunto de acciones: son las salidas producidas en respuesta a los eventos de acuerdo al estado actu

Ejemplo: Una pila de tamaño acotado.

El interés esta en los casos en que la pila está vacía, llena, o ni uno ni lo otro.



Este modelo nos permite planear el testing mientras se esta codificando, y para cuando el sistem este construido se pueden aplicar los test planeados.

Criterios para generar los casos de test. Ej.:

- Cobertura de transiciones: el conjunto T de casos de test debe asegurar que toda transición sea ejecutada.
- Cobertura de par de transiciones: T debe ejecutar todo par de transiciones adyacentes que entran y salen de un estado.
- Cobertura de árbol de transiciones: T debe ejecutar todos los caminos simples (del estado inicial al final o a uno visitado)

CAJA BLANCA.

Tipos de testing estructural:

- Criterio basado en el flujo de control.
Observa la cobertura del grafo de flujo de control.
- Criterio basado en el flujo de datos.
Observa la cobertura de la relación definición-uso en las variables.
- Criterio basado en mutación.
Observa a diversos mutantes del programa original.

Criterios basados en flujo de control:

Considera al programa como un grafo de flujo de control. Donde suponemos la existencia de un nodo inicial y de uno final. Nodos, Representan bloques de código. Conjunto de sentencias que siempre se ejecutan juntas. Aristas, Posible transferencia de control del nodo i al nodo j. Suponemos, la existencia de un nodo inicial, y final. Camino, secuencia del nodo inicial hacia, el nodo final.

Criterio de cobertura de sentencia

Cada sentencia se ejecuta al menos una vez durante el testing.
i.e. el conjunto de caminos ejecutados durante el testing debe incluir todos los nodos.

- Limitación: Puede no requerir que una decisión evalúe a falso en un if si no hay else:
Ej.:
$$\text{abs}(x) : \begin{cases} \text{if } (x \geq 0) \text{ then } x = -x; \\ \text{return}(x) \end{cases}$$
El conjunto de casos de test $\{(x = 0, 0)\}$ tiene el 100% de cobertura pero el error pasa desapercibido.
- No es posible garantizar 100% de cobertura debido a que puede haber nodos inalcanzables.

Criterio de cobertura de ramificaciones

Cada arista debe ejecutarse al menos una vez en el testing.
Cada decisión debe ejercitarse como verdadera y como falsa durante el testing.

- La cobertura de ramificaciones implica cobertura de sentencias.
- Si hay múltiples condiciones en una decisión, luego no todas las condiciones se ejercitan como verdadera y falsa.
-

Todo test suite que cubre ramificaciones, también cubre sentencias.

Pero: puede haber test suites que cubren ramificaciones que no encuentren el defecto mientras haya tests suites que cubren sentencias que sí lo detecten.

Ej.:
$$\text{abs}(x) : \begin{cases} \text{if } (x \leq 0) \text{ then } x = x; \\ \text{return}(x) \end{cases}$$

Cobertura de Sentencias: $\{(x = -1, 1)\}$

Cobertura Ramificaciones: $\{(x = 1, 1); (x = 0, 0)\}$

Criterio de cobertura de caminos

Todos los posibles caminos del estado inicial al final deben ser ejecutados. • Cobertura de caminos implica cobertura de bifurcación. Problema: la cantidad de caminos puede ser infinita (considerar loops). Notar además que puede haber caminos que no son realizable.

Criterio basados en flujo de datos

Se construye un grafo de definición-uso etiquetando apropiadamente el grafo de flujo de control.

Una sentencia en el grafo de flujo de control puede ser de tres tipos:

- **def**: representa la definición de una variable (i.e. cuando la var está a la izquierda de la asignación).
- **uso-c**: cuando la variable se usa para cómputo.
- **uso-p**: cuando la variable se utiliza en un predicado para transferencia de control.

Criterios:

- **Todas las definiciones**: por cada nodo i y cada x en $\text{def}(i)$ hay un camino libre de definiciones con respecto a x hasta un uso-c o uso-p de x .
- **Todos los usos-p**: todos los usos-p de todas las definiciones deben testearse.
- **otros criterios**: todos los usos-c, algunos usos-p, algunos usos-c.

```
1. scanf(x, y); if (y < 0)
2.     pow = 0 - y;
3. else pow = y;
4. z = 1.0;
5. while (pow != 0)
6.     { z = z * x; pow = pow - 1; }
7. if (y < 0)
8.     z = 1.0/z;
9. printf(z);
```

Ejemplo

1. Identificar el conjunto de caminos que satisface el criterio deseado.

2. Identificar los casos de test que ejercitan dichos caminos.

Cobertura de ramificación:

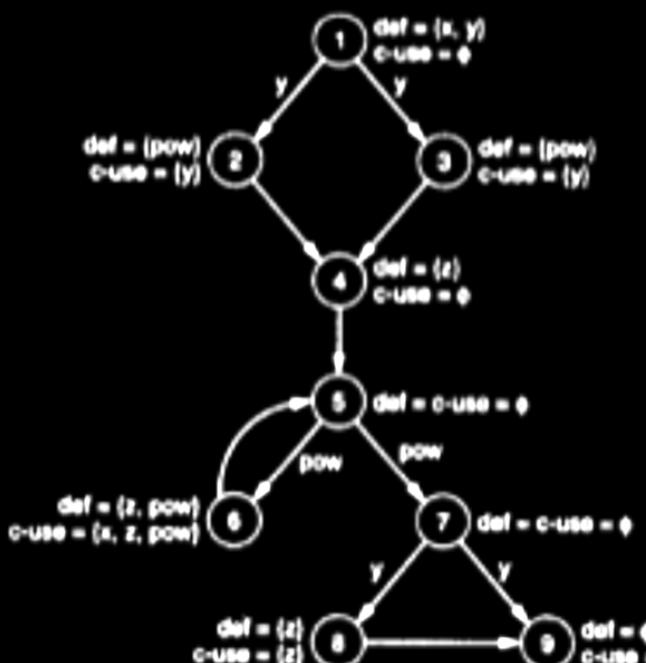
Caminos: (1,2,4,5,6,5,7,8,9) y (1,3,4,5,6,5,7,9)

Tests: ((x=3, y=1); z=...) y ((x=3, y=-1); z=...)

Cobertura de todas las definiciones:

Caminos: (1,2,4,5,6,5,6,5,7,8,9) y (1,3,4,5,6,5,7,9)

Tests: ((x=3, y=2); z=...) y ((x=3, y=-1); z=...)



Planeamiento del proceso de software

Es la primera fase del proceso de administración del proyecto; que a su vez es, una componente del proceso de software. El planeamiento se realiza antes, del desarrollo del proyecto. Requiere como entradas, los requerimientos, y la arquitectura. Durante esta etapa se, planea todas las tareas que la administración del proyecto necesitará realizar. Este plan se ejecuta y actualiza, durante el seguimiento y control.

Tópicos más importantes

1. Planeamiento del proceso.

Planear como, se ejecutara el proyecto. Por ende esto incluye

- a. Determinar el modelo de proceso seguir; y adecuarlo a las necesidades del proyecto.
- b. Definir las etapas con, criterios de salida y entrada para c/u; además de, actividades de verificación a realizar en cada etapa.
- c. Definir metas parciales, usadas para analizar el proceso del proyecto. (milestones)

2. Estimación del esfuerzo.

Se mide (usualmente) en personas/mes; y considerando la recarga de costos por persona, puede convertirse directamente en costo. Permitiendo, evaluar la factibilidad del proyecto, analizar costos-beneficio, efectuar ofertas, etc.

Además, es clave en el planeamiento; de ella dependerá, los tiempos, costos, y recursos (humánanos, principalmente).

Observemos que sí, se estima inapropiadamente; causa problemas en la ejecución del proyecto.

El cálculo empieza con un tentativo, y se vuelve a calcular a medida que aumenta la información del proyecto; acercándose, cada vez más, a una "ideal".

Existen diversos modelos para, calcularlo. Estos reducen la estimación a, estimar correctamente valores de ciertos parámetros medidos en etapa temprana (tamaño suele ser el principal). Para su construcción, hay 2 enfoques generales:

- Top-Down: Determinar el esfuerzo total, y luego calcularlo para cada parte del proyecto.
 1. Se estima el tamaño global, medido en, 100 líneas de código (KLOC).
 2. Se calcula el esfuerzo multiplicándolo por, "a" y "b"; constantes determinadas por el análisis de regresión sobre proyectos pasados.

Los datos para la distribución del esfuerzo en cada fase a partir de proyectos similares.

- Bottom-Up:

1. Identifica los modulos del sistema, y clasificarlos como simples, medianos, o complejos.
2. Determinar el esfuerzo promedio de codificar, c/u de estos tipos de modulos.
3. Se obtiene el esfuerzo total de la codificación en base a, la clasificación anterior, y el conteo de cada tipo.
4. Utilizar la distribución de esfuerzos de proyectos similares para estimar el esfuerzo de cada tarea y finalmente el esfuerzo total.
5. Refinar los estimadores anteriores en base a factores específicos del proyecto.

3. Estimación de tiempos y recursos, o planificación y recursos humanos.

4. Plan para la administración de la configuración.

Se deben identificar los ítems de configuración y especificar los procedimientos a usar para controlar e implementar los cambios de estos ítems.

El planeamiento de la administración de configuración se realiza cuando el proyecto ha sido iniciado y ya se conoce la especificación de los requerimientos y el entorno de operación.

5. Planeamiento de la calidad.

6. Administración del riesgo.

7. Plan para el seguimiento del proyecto

COCOMO

Enmarcado en la fase de, estimación del esfuerzo; perteneciente a, la 1.^a etapa del proceso de administración del proyecto; quien a su vez es, una componente del proceso de software

Es un modelo que permite, estimar el esfuerzo. Como todos sus pares, reducen la estimación a, estimar correctamente valores de ciertos parámetros medidos en etapa temprana (tamaño suele ser el principal). También comparte con ellos el hecho de que, el cálculo empieza con un tentativo, y se vuelve a calcular a medida que aumenta la información del proyecto; acercándose, cada vez más, a una "ideal".

Está construido bajo un enfoque, Top-Down; es decir, determina el esfuerzo total, y luego calcularlo para cada parte del proyecto. Además, utiliza el tamaño (medido en KLOC), ajustados con algunos factores.

Su procedimiento es:

- A. Obtener el estimado inicial, usando el tamaño.

Obtiene el valor $E_i = a * Tamaño^b$ que sería Tamaño es, cuan grande uno cree que sería el sistema.

Los valores de a y b, se obtienen de una tabla Las constantes a y b se determinan a través de análisis de regresión sobre columnas a y b. proyectos pasados

Filas, estilo de SISTEMA,

Orgánico: Relativamente simple, y desarrollado por pequeños equipos.

Semi-rígido: Mezcal entre, orgánico, y rígido.

Rígido: Más ambiciosos, y novedosos; con restricciones estrictas impuestas por el entorno; con altos requerimientos en aspectos como, confiabilidad, e interfaz.

- B. Determina, 15 factores de multiplicación;

- Atributos del software:

RELY: Confiabilidad

DATA: Tamaño de la base de datos

CPLX: Complejidad de las funciones, datos, interfaces...

- Atributos del hardware:

TIME: Limitaciones en el porcentaje del uso de la CPU

STOR: Limitaciones en el porcentaje del uso de la memoria

VIRT: Volatilidad de la máquina virtual

TURN: Frecuencia de cambio en el modelo de explotación

- Atributos del personal:

ACAP: Calificación de los analistas

AEXP: Experiencia del personal en aplicaciones similares

PCAP: Calificación de los programadores

VEXP: Experiencia del personal en la máquina virtual

LEXP: Experiencia en el lenguaje de programación a usar

- Atributos del proyecto:

MODP: Uso de prácticas modernas de programación

TOOL: Uso de herramientas de desarrollo de software

SCED: Limitaciones en él, címprome

Los valores de los factores de ajuste se obtiene de una tabla; cuyas filas son los atributos, y las columnas su grado de presencia (muy bajo, bajo, normal, alto, muy alto, y extra alto).

- C. Ajusta el estimado de esfuerzo escalando según el factor de multiplicación final.

$$esfuerzo = E_i * \prod_{i=1}^{15} f_i$$

- D. Calcula el estimado de esfuerzo para, c/u de las fases principales.

Para cada tipo de sistema (orgánico, semi-rígido, y rígido) existe una tabla que es indexada; donde sus filas son las fases, y las columnas el tipo de tamaño en función de las kLOCs.

3. Planificación y recursos humanos

Enmarcado en la fase de estimación del esfuerzo; perteneciente a, la 1.^a etapa del proceso de administración del proyecto; quien a su vez es, una componente del proceso de software

Posee 2 niveles:

Global

Abarca las metas parciales (milestones), y la fecha final.

Depende, fuertemente, del esfuerzo estimado. Para una estimación dada hay cierta flexibilidad, en función de los recursos asignados. Pero los cambios, no son tan lineales.

Un método para

es calcular el tiempo programado del proyecto M (en meses); como una función del esfuerzo en personas-mes. Esta no es linea, y se determina analizando datos pasados. Seguidamente

Ahora bien para, determinar la duración de cada meta parcial principal del proyecto. Ahí que contemplar que, la distribución de los RRHH no suele ser homogenea; generalmente sigue una curva escalonada donde el reparto del personal es, construcción, diseño, testing. Con esta curva y la estimación del esfuerzo, se puede determinar el tiempo de las metas parciales. Esto implica que la distribuciones en las fases de, esfuerzo y timeposo; pasen a ser distintas por que, tal vez hemos contratado más gente, y aunque el esfuerzo sea mayor, los tiempos sean distintos. Además, usualmente la etapa de construcción lleva el mayor esfuerzo pero, no necesariamente más duración.

Detallada

La asignación de tareas de más bajo nivel, a los recursos (en general personas). Para alcanzar una meta, muchas tareas de bajo nivel deben ejecutarse. Estas son realizadas por una sola persona en no más de 2 o 3 días. La planificación detallada se aboca a este tipo de labores, siempre preservando la planificación de alto nivel, y siendo consistente con las metas. Las tareas son subactividades de, la nivel de metas; por ende el esfuerzo a nivel individual debe sumar apropiadamente la duración total.

Es un proceso iterativo, si no se pueden acomodar todas las tareas se revisa la planificación global. Además, evoluciona con el tiempo. Por último, cualquier actividad a realizarse debe quedar reflejarse en la planificación detallada; para c/u se asiga, un nombre, un esfuerzo, una fecha, una duración, recursos, y el porcentaje de la realización usado para seguimiento.

Estructuras de equipo de trabajo

Para asignar las tareas en la planificación detallada, es necesario un equipo de trabajo estructurado, existiendo 3 tipos.

- Organizaciones jerárquicas
 - Tienen un administrador del proyecto, con la responsabilidad global. Además de, programadores, testers, y administradores de configuración para ejecutar estas labores. Existen otros roles, y una persona puede cubrir más de uno.
- Equipos democráticos
 - Suele funcionar en, equipos pequeños. El liderazgo es, rotativo.
- Alternativa
 - Para el desarrollo de grandes programas,
 - Reconoce 3 tareas principales; desarrollo, testing, y administración del proyecto. C/u de ellas tienen un equipo, y un líder; y las 3 reportan a un líder general.
 - Se espera que sean independientes los equipos de, testing, y desarrollo.
 - Los administradores del programa proveen, las especificaciones de lo que se debe construir; y aseguran que desarrollo y testing, estén apropiadamente coordinados.

6 Administración del riesgos

Enmarcado en la fase de estimación del esfuerzo; perteneciente a, la 1.^a etapa del proceso de administración del proyecto; quien a su vez es, una componente del proceso de software

Procura minimizar el impacto (en los costos, calidad, y tiempos) de la materialización de los riesgos. Entendiendo por riesgo, a cualquier condición o evento de ocurrencia incierta que puede causar la falla del proyecto. Tales eventos no son comunes (Ejemplo: no incluye movimiento de personal o cambio de requerimientos, los que deben ser tratados por la administración del proyecto).

Posee 2 grandes áreas:

- **Evaluación**

Identificar, analizar, y definir las prioridades de los riesgos. Es realizada durante el planeamiento del proyecto.

- **Identificación del riesgo:** Identificar los riesgos del proyecto; es decir aquellos eventos que podrían ocurrir, y occasionar la falla del proyecto. Se hace a través de, listas de control, experiencias, lluvia de ideas, etc.
- **Análisis del riesgo y decisión de prioridades:** Se analizan las posibilidades de ocurrencia del riesgo. Para lo cual, se define el "valor de exposición al riesgo" $RE = \text{Probabilidad de ocurrencia de este riesgo} * \text{el impacto}$; es decir, el valor esperado de la perdida, debido al que el riesgo suceda. Se planificaran aquellos riesgos que tengan RE elevado.

Procedimiento:

1. Clasificar las probabilidades de ocurrencia como, bajas, medias, o altas.
2. Clasificar el impacto como, bajo, medio, o alto.
3. Identificar los riesgos que tengan altas altas probabilidades de ocurrencia, y que su impacto sea alto.

- **Control de los riesgos**

Se lleva a cabo, durante la realización del proyecto. Planea la administración de los riesgos, la resolución, y el seguimiento. Si es posible evitar el riesgo, se evita. En otros casos, se planean y ejecutan los pasos necesarios para mitigarlos; es decir, definir las acciones a seguir en el proyecto de manera que si el riesgo se materializa, su impacto sea mínimo.

Plan de la mitigación de los riesgos; incluyen los pasos a seguir, cuando ocurren; en consecuencia a, costo extra. Se ejecutan luego, de la ocurrencia del riesgo. Este plane no solo debe hacer al principio, si no a medida que el sistema se ejecuta; tiene que revisarse que, los riesgos que planeamos que podían suceder sigan siendo los mismos, y no aparezcan nuevos.

