Ingeniería del Software

Codificación
El objetivo en esta etapa es implementar el diseño de la mejor manera posible. La codificación afecta al testing y al mantenimiento. Debido a que estos últimos tienen costos muy elevados, el propósito de la codificación es escribir código que reduzca tales costos.
El objetivo no es reducir los costos de implementación, sino los de testing y mantenimiento. Y facilitar el trabajo de quienes testean y de quienes mantendrán el sistema.
Un programa se lee mucho más frecuentemente que el tiempo que demanda su escritura:

- Los programadores leen el código muchas veces para hacer debugging, extenderlo, modificarlo, etcétera.
- Quienes mantienen el código invierten mucho esfuerzo en la lectura y comprensión del código.
- Otros desarrolladores leen el código cuando agregan otras partes al programa. Entonces: el código debe ser fácil de leer y comprender (pero no necesariamente de escribir).

Principios y pautas para la programación

- Objetivo principal del programador: escribir programas simples y fáciles de leer con la menor cantidad de errores (bugs) posibles.
- Por supuesto: también debe hacerlo rápidamente para que su productividad sea alta.
- Existen principios y pautas para la programación que pueden ayudar a escribir código de alta calidad (i.e. fácilmente comprensible), que sea fácil de testear y mantener.

Programación estructurada
La programación estructurada se inició en los 70, fundamentalmente contra el uso indiscriminado de constructores de control como los “gotos”. Hoy es un paradigma bien establecido y utilizado y su objetivo es simplificar la estructura de los programas de manera que sea fácil razonar sobre ellos.

- Un programa tiene una estructura estática la cual es el orden de las sentencias en el código (el cual es un orden lineal).
- Un programa tiene una estructura dinámica que es el orden el cual las sentencias se ejecutan.
- Cada estructura define un orden en las sentencias.
- La corrección de un programa debe hablar de la estructura dinámica.
- Para mostrar que un programa es correcto debemos mostrar que el comportamiento dinámico es el esperado.
- Pero debemos razonar sobre el código del programa, i.e. la estructura estática. i.e. la justificación del comportamiento de un programa se realiza sobre el código estático.
- Esto sería más simple si las estructuras dinámica y estática fueran similares: una correspondencia cercana facilitará la comprensión del comportamiento dinámico desde la estructura estática.

El objetivo de la programación estructurada es escribir programas cuya estructura dinámica es la misma que la estática, es decir, que las sentencias se ejecutan en el mismo orden que las presenta el código.
Como las sentencias se organizan linealmente (estático), el objetivo es desarrollar programas cuyo flujo de control (dinámico) sea lineal.

- Los constructores de la programación estructurada son de una única entrada y una única salida.
- De esta manera, la ejecución de las sentencias se realiza en el orden en el que aparecen en el código. Por lo que el orden dinámico y el estático son el mismo.
- Los constructores estructurados no pueden ser arbitrarios: deben mostrar un comportamiento claro.
- Se puede mostrar que los constructores “if”, “while”, y la secuencia alcanzan para escribir cualquier tipo de programa.

Entonces la programación estructurada simplifica el flujo de control, facilitando en consecuencia tanto la comprensión de los programas así como el razonamiento (formal o informal) sobre estos.

Ocultamiento de la información

- Las soluciones de software siempre contienen estructuras de datos que guardan información.
- Los programas trabajan sobre estas estructuras de datos para realizar ciertas funciones.
- En general sólo ciertas operaciones se realizan sobre la información, es decir, los datos sólo se manipulan de pocas maneras.
- En consecuencia la información debería ocultarse de manera que sólo quede expuesta a esas pocas operaciones, es decir, las estructuras de datos son ocultadas tras las funciones de acceso que son las únicas que puede usar el programa.
- El ocultamiento de la información reduce el acoplamiento.

Prácticas de programación

- Constructores de control: Utilizar algunos pocos constructores estructurados (en lugar de una gran variedad de ellos).
- Gotos: No usar (limitado a caso donde las alternativas son peores).
- Ocultamiento de la información: ¡Usarla!.
- Tipos definidos por el usuario: Utilizarlos para facilitar la lectura de los programas.
- Tamaño de los módulos: No deberían ser muy largos (sino probable baja cohesión).
- Interfaz del módulo: Hacerla simple.
- Robustez: Manipular situaciones excepcionales.
- Efectos secundarios: Evitarlos, documentar (ej.: variables globales).
- Bloque “catch” vacío: Realizar siempre alguna acción por defecto en lugar de nada.
- “if” o “while” vacío: Pésima práctica.
- Switch case: Usar default.
- Valores de retorno en las lecturas: leer para lograr robustez.
- “return” en “finally”: No usar.
- Fuentes de datos confiables: Desconfiar (usar psw, hash, etc.).
- Dar importancia a las excepciones: los casos excepcionales son los que tienden a hacer que el programa funcione mal.

Estandares de codificacion

- Los programadores pasan más tiempo leyendo código que escribiendo código.
- Leen tanto su propio código como el de otros programadores.
- La legibilidad del código aumenta si todos siguen ciertas convenciones de codificación.
- Los estándares de codificación proveen esas pautas para los programadores.
- Cierta dependencia del lenguaje / comunidad / empresa.

Proceso de codificación incremental

- La codificación comienza ni bien está disponible la especificación del diseño de los módulos.
- Usualmente los módulos se asignan a programadores individuales.
- Desarrollo top-down => los módulos de los niveles superiores se desarrollan primero.
- Desarrollo bottom-up => los módulos de los niveles inferiores se desarrollan primero.
- Para la codificación se pueden utilizar distintos procesos.

Proceso básico: es mejor realizar este de manera incremental.

→Escribir el código del módulo.

→Realizar test de unidad.
→Sin error: arreglar bugs y repetir tests.

Desarrollo dirigido por test

- Este proceso de codificación cambia el orden de las actividades en la codificación.
- En TDD el programador primero escribe los scripts para los tests y luego el código para que estos pasen los casos de tests en el script.
- Se realiza incrementalmente.
- Nueva técnica, parte de Extreme programming (pero se puede usar independientemente).
- En TDD se escribe el código suficiente para pasar el test, es decir, el código está en sincronía con los tests y es testeado por estos casos de test. No es lo mismo en el modelo anterior donde los casos de test podrían testear sólo parte de la funcionalidad.
- La responsabilidad de asegurar cobertura de toda la funcionalidad radica en el diseño de los casos de test y no en la codificación.
- Ayuda a asegurar que todo el código es testeable.
- Se enfoca en cómo será usado el código a desarrollar dado que los tests se escriben primero. Ayuda a validar la interfaz del usuario especificada en diseño.

Cuidado:

- La completitud del código depende de cuan exhaustivo sean los casos de test.
- El código necesitará refactorización para mejorar el código posiblemente confuso

Programación de a pares

También propuesto como práctica en XP, el código se escribe por dos programadores en lugar de uno solo:

- Conjuntamente, ambos programadores diseñan los algoritmos, estructuras de datos, estrategias, etcétera.
- Una persona tipea el código, la otra revisa activamente el código que se tipea.
- Se señalan los errores y conjuntamente formulan soluciones.
- Los roles se alternan periódicamente.

La revisión de código en este modelo es continua y conlleva a un mejor diseño de algoritmos, estructuras de datos, lógica, entre otros. Es más fácil que se escapen condiciones particulares, y la efectividad de este método no es aún bien conocida (pérdida de productividad).

Refactorización

- Usualmente los códigos se modifican con el fin de aumentar su funcionalidad.
- Con el tiempo, aún si el diseño inicial era bueno, los cambios en el código deterioran el diseño.
- Al complicarse el diseño, comienza a hacerse más complicado modificar el código y más susceptible a errores, es decir, la calidad y la productividad en la realización de cambios comienza a disminuir.
- La refactorización es una técnica para mejorar el diseño del código existente.
- Se realiza durante la codificación, pero el propósito no es agregar nuevas características sino mejorar el diseño.

El objetivo de la refactorización no es corregir bugs. Se aplica al código que ya está funcionando. Es la tarea que permite realizar cambios en un programa con el fin de simplificarlo y mejorar su comprensión, sin cambiar el comportamiento observacional de éste.
La estructura interna del software cambia, pero el comportamiento externo permanece igual. El objetivo básico es el de mejorar el diseño plasmado en el código (no es lo mismo que mejorar el diseño durante el proceso de diseño).

Conceptos básicos

- Dado que el fin es mejorar el diseño, la refactorización intenta lograr una o más de las siguientes cosas: **reducir acoplamiento, incrementar cohesión, mejorar respuesta al principio abierto-cerrado.**
- Cualquier modificación al código con el fin de llevar a cabo lo anterior no debe cambiar la funcionalidad.
- La refactorización se realiza durante la codificación y generalmente está asociada a un requerimiento de cambio.
- Sin embargo: no mezclar codificación normal con refactorización. Los cambios por refactorización se realizan separadamente de la codificación normal.
- El principal riesgo de realizar una mejora al código, es que se puede “romper” la funcionalidad existente.
- Para disminuir esta posibilidad:
- Refactorizar en pequeños pasos.
- Disponer de scripts para tests automatizados para testear la funcionalidad existente.
- La refactorización permite que el diseño del código mejore continuamente en lugar de degradarse con el tiempo.
- El código extra de la refactorización se recupera en la reducción del costo en los cambios. No es necesario tener el diseño más general desde el comienzo; se pueden elegir diseños más simples.
- Hace más fácil y menos riesgosa la tarea inicial de diseño.

Refactorizaciones más comunes

- Muchas formas de mejorar el diseño de programas.
- Existen catálogos que se extienden continuamente.
- Para mejorar el diseño se enfocan en métodos, clases y jerarquía de clases.
- Siempre con el objetivo de mejorar acoplamiento, cohesión, y el principio de abierto-cerrado.

Refactorizaciones más comunes - Mejoras de métodos

- **Extracción de métodos:** Se realiza si el método es demasiado largo, el objetivo es separar en métodos cortos cuya signatura indique lo que el método hace. Partes de código se extraen como nuevos métodos. Las variables referenciadas en esta parte se transforman en parámetros. Variables declaradas en esta parte pero utilizadas en otras partes deben definirse en el método original. También se realiza si un método retorna un valor y también cambia el estado del objeto. (Dividir en dos métodos).
- **Agregar/eliminar parámetros:** para simplificar las interfaces donde sea posible. Agregar sólo si los parámetros existentes no proveen la información que se necesita, además es necesario eliminar si los parámetros se agregaron originalmente “por las dudas” pero no se utilizan.

Refactorizaciones más comunes - Mejoras de clases

- **Desplazamiento de métodos:**
    1. Mover un método de una clase a otra.
    2. Se realiza cuando el método actúa demasiado con los objetos de la otra clase.
    3. Inicialmente puede ser conveniente dejar un método en la clase inicial que delegue al nuevo (debería tender a desaparecer).

- **Desplazamiento de atributos:**
    1. Si un atributo se usa más en otra clase, moverlo a esta clase.
    2. Mejora la cohesión y el acoplamiento.

- **Extracción de clases:**

1. Si una clase agrupa múltiples conceptos, separa cada concepto en una clase distinta.
2. Mejora cohesión

- **Reemplazar valores de datos por objetos:**
    1. Algunas veces, una colección de atributos se transforma en una entidad lógica.
    2. Separarlos como una clase y definir objetos para accederlos.

Refactorizaciones más comunes - Mejoras de jerarquías

- **Reemplazar condicionales con polimorfismos:** Si el comportamiento depende de algún indicador de tipo, no se está explotando el poder de la OO. Reemplazar tal análisis de casos a través de una jerarquía de clases apropiada.
- **Subir métodos / atributos:** los elementos comunes deben pertenecer a la superclase. Si la funcionalidad o atributo está duplicado en las subclases, pueden subirse a la superclase.

Modelos de desarrollo de proceso

Un modelo de proceso **especifica un proceso general**, usualmente **como un conjunto de etapas**. Este modelo es adecuado para una clase de proyectos, es decir, un modelo de proceso **provee una estructura genérica de los procesos que puede seguirse en algunos proyectos con el fin de alcanzar sus objetivos**.
Si se elige un modelo para un proyecto, **será necesario adecuarlo al mismo**, este proceso produce la especificación del proceso del proyecto. Por lo tanto:

- **Modelo del proceso:** especificación genérica del proceso.
- **Especificación del proceso:** plan de lo que debe ejecutarse.
- **Proceso**: lo que realmente se ejecuta.

Modelos más comunes

1. Cascada (el modelo más viejo y ampliamente usado).
2. Prototipado.
3. Iterativo (muy usado en la actualidad).
4. Timebox.

Modelo de cascada

Secuencia inicial de las distintas fases:

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

Una fase comienza sólo cuando la anterior finaliza (en principio, no hay feedback), y estas últimas dividen al proyecto, cada una de ellas se encarga de distintas incumbencias.

- El orden lineal de las acciones tiene consecuencias importantes.
- En la práctica no se necesitan requerimientos detallados para el planeamiento. Además, el planeamiento y el análisis de requerimientos se superponen.
- El final de una fase y comienzo de la siguiente claramente identificado por un mecanismo de certificación (i.e. V&V).
- Todas las fases deben producir una salida definida.

Productos de trabajos usuales en este modelo (nombres no necesarios pero si saber que existen a cada salida de las fases):

- 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 (usuario, instalación, etcétera).
- Reportes de revisión, reportes de estado, etc.

Ventajas de este modelo
Conceptualmente es simple y divide claramente el problema en distintas fases que pueden realizarse de manera independiente. Tiene un enfoque natural a la solución de problemas. Por último, es fácil de administrar en un contexto contractual ya que existen fronteras bien definidas entre cada fase.

Uso
- Ampliamente usado.
- Muy adecuado para proyectos donde los requerimientos son bien comprendidos y las decisiones sobre la tecnología son tempranas.
- Es adecuado (y usado con frecuencia) para tipos de proyectos con los cuales los desarrolladores están muy familiarizados con el problema a atacar y el proceso a seguir.

Prototipo
Aborda las limitaciones del modelo de cascada en la especificación de los requerimientos. En lugar de congelar los requerimientos basados en charlas y debates, se construye un prototipo que permita comprender los requerimientos.Permite que el cliente tenga una idea de lo que sería el SW y así conseguir mejor feedback de él, ayudando a disminuir los riesgos de requerimientos. La etapa de análisis de requerimientos es reemplazada por una ‘’mini-cascada’’. El prototipo debe descartarse.

Desarrollo
- Comienza con una versión preliminar de los requerimientos.
- El prototipo **solo incluye las características claves** que necesitan mejor comprensión.
- Es inútil incluir características bien entendidas.
- **El cliente “juega” con el prototipo y provee feedback** importante que mejora la comprensión de los requerimientos.
- **Luego del feedback el prototipo se modifica y se repite el proceso** hasta que los costos y el tiempo **superen los beneficios de este proceso.**
- Teniendo en cuenta el feedback, los requerimientos iniciales se modifican para producir la especificación final de los requerimientos.

El costo del prototipado debe mantenerse bajo, por ende es importante:
- Construir sólo aspectos que se necesite aclarar.
- **“Quick & Dirty”:** la calidad no importa, sólo poder desarrollar el prototipo rápidamente.
- Omitir manejo de excepciones, recuperación, estándares.
- Reducir testing.
- Los costos deben ser un pequeño % del costo total.

Ventajas
Este proceso de desarrollo tiene una **mayor estabilidad en los requerimientos**, estos últimos se congelan más tarde y la experiencia en la construcción del prototipo ayuda al desarrollo principal.
Como desventajas, tiene un potencial impacto en costo y en tiempo.

Aplicación
Cuando los requerimientos son difíciles de determinar y la confianza en ellos es baja (i.e. los requerimientos no se han comprendido).

Desarrollo Iterativo
- Aborda el problema de “todo o nada” del modelo de cascada.
- Combina beneficios del prototipado y de la cascada.
- Desarrolla y entrega el SW incrementalmente.
- Cada incremento es completo en sí mismo.
- Provee un marco para facilitar el testing (el testing de cada incremento es más fácil que el testing del sistema completo).
- Puede verse como una “secuencia de cascadas”.
- El feedback de una iteración puede usarse en iteraciones futuras.

Modelo con mejora iterativa
Primer paso: consiste en una implementación simple para un subconjunto del problema completo. Se debe crear una **lista de control del proyecto (LCP)** que contiene en orden las tareas que se deben realizar para lograr la implementación final.

Cada paso consiste en eliminar la siguiente tarea de la lista haciendo diseño implementación y análisis del sistema parcial, y actualizar la LCP. **Esto se repite hasta vaciar la lista.** Siendo esta última, quien guía los pasos de iteración y lleva la lista de tareas a realizar. Cada entrada en LCP es una tarea a realizarse en un paso de iteración y debe ser lo suficientemente simple como para comprenderla completamente.

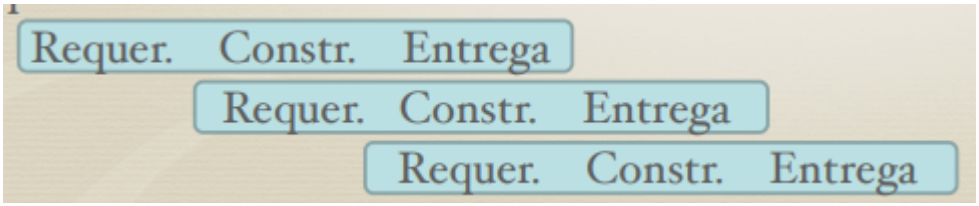
Aplicación del desarrollo iterativo
Muy efectivo en desarrollo de productos: los **desarrolladores mismos proveen la especificación mientras que los usuarios proveen el feedback en cada release**, además de estar basado en la experiencia previa (lo que lleva a una nueva revisión).
Desarrollo a gusto del comprador o customized: las empresas requieren respuestas rápidas, no se puede arriesgar a todo o nada.

Ventajas e inconvenientes
Pagos y entregas incrementales, junto con un feedback para mejorar el desarrollo.
La arquitectura y el diseño pueden no ser óptimos, la revisión del trabajo hecho puede incrementarse y el costo puede ser mayor.

Aplicación
Cuando el tiempo de respuesta es importante, cuando no se puede tomar riesgos de proyectos largos y cuando no se conocen los requerimientos.

Modelo timeboxing
Es un modelo iterativo que presenta una secuencia lineal de iteraciones. Cada una de estas últimas, es una “mini cascada” la cual consiste en decidir la especificación y luego planear la iteración.

Timeboxing: primero fija la duración de las iteraciones y luego determina la especificación. Divide la iteración en partes iguales y usa pipelining para ejecutar iteraciones en paralelo.



Ventajas y desventajas

- Todos los beneficios del iterativo, menor tiempo de entrega y ejecución del proyecto distribuida.
- Grandes equipos de trabajos, administración del proyecto más compleja, y se necesita mucha sincronización.

Aplicación

Cuando los tiempos de entrega cortos son muy importantes.

El proceso de Software

Existe una diferencia entre el producto y el proceso, ya que el producto es el resultado de haber ejecutado un proceso. Como ya sabemos, IS se enfoca en el proceso con la premisa de “Un proceso adecuado ayuda a lograr los objetivos del proyecto con alta C&P”.

Procesos y modelos de procesos

- Un proyecto exitoso es el que satisface las expectativas en costo, tiempo, y calidad.
- Al planear y ejecutar un proyecto de sw, las decisiones se toman con el fin de reducir costos y tiempos e incrementar la calidad.
- **Un modelo de proceso especifica un proceso general, usualmente con fases en las que el proceso debe dividirse, conjuntamente con otras restricciones y condiciones para la ejecución de dichas fases.**
- Un modelo de proceso **no se traduce directamente al proceso**: en general el proceso real es una adaptación del modelo del proceso.
- Importante: es el proceso el que guía un proyecto, e influye significativamente el resultado esperado del proyecto.

Tiene dos procesos fundamentales:

- **Desarrollo:** se enfoca en las actividades para el desarrollo y para garantizar la calidad necesaria para la ingeniería del sw. Es el corazón del proceso de software, y los demás procesos giran en torno a él.
- **Administración del proyecto:** se enfoca en el planeamiento y control del proceso de desarrollo con el fin de cumplir los objetivos.

Cada proceso se ejecuta por gente diferente, los desarrolladores ejecutan el desarrollo mientras que los administradores de proyecto ejecutan el proceso de administración.

Especificación del proceso

El proceso generalmente es un conjunto de fases, y **cada una de estas realiza una tarea bien definida que produce una salida**. Tal salida intermedia se llama **producto de trabajo**. Cada producto de trabajo es una entidad formal y tangible que es capaz de ser verificada, y cada una de estas fases puede ser llevada a cabo mediante diferentes metodologías.

Enfoque "ETVX"
Cada fase sigue el enfoque ETVX (Entry - Task - Verification - Exit):

- **Criterio de entrada:** qué condiciones deben cumplirse para iniciar la fase.
- **Tarea:** Lo que debe realizar esa fase.
- **Verificación:** Las inspecciones/controles/revisiones/verificaciones que deben realizarse a la salida de la fase (i.e. al producto de trabajo).
- **Criterio de salida:** Cuando puede considerarse que la fase fue realizada exitosamente.

Además, cada fase produce información para la administración del proceso

Características deseadas
Proveer alta C&P

- Debe producir sw testable: testing es la tarea más cara dentro del proceso de desarrollo; entre 30 y 50% del esfuerzo total de desarrollo.
- Debe producir sw mantenible: el mantenimiento puede ser más caro que el desarrollo; hasta 80% del costo total durante la vida del sw.
- Debe eliminar defectos en etapas tempranas.
- Debe ser predecible y repetible.
- Debe soportar cambios y producir sw que se adapte a cambios.
- El costo de eliminar un defecto se incrementa a medida que perdura en el proceso de desarrollo.
- Para lograr alta C&P, los errores deberían ser encontrados en la etapa en que se introdujeron. (La "V" en "ETVX").
- Control de calidad: actividad cuyo propósito principal es identificar y eliminar errores.
- CC es limitado → además prevenir defectos.

Los procesos deben conseguir repetir el desempeño cuando se utilizan en distintos proyectos. Es decir, el resultado de utilizar un proceso debe ser predecible, no sólo para estimar costos y esfuerzos, sino para estimar calidad. Esto sirve para estimar un desempeño futuro. Si el proceso es predecible, está bajo control estadístico, y debe estarlo para ser consistente en CyP.

El objetivo del proceso de software es construir sistemas de sw dentro de los costos y el tiempo planeado, cronograma, y que posea la calidad apropiada, satisfaciendo al cliente, alta C&P. Para cada proyecto, el proceso de desarrollo a seguir se especifica durante el planeamiento.

- Es un conjunto de fases.
- Cada fase es una secuencia de pasos que definen una metodología en esa fase.
- Se divide en fases para seguir el principio de "dividir y conquistar".
- Cada fase ataca distintas partes del problema.
- La separación en fases ayuda a validar continuamente el proyecto.

Proceso para la administración del proyecto
Es parte de este proceso, asignar recursos, administrarlos, observar el progreso, tomar acciones correctivas, entre otros, para ejecutar efectivamente las fases y actividades del proceso de desarrollo.La administración del proyecto es una parte esencial de la ejecución del proyecto. Sus fases son :

- **Planeamiento :** se realiza antes de comenzar el proyecto, entre sus tareas claves, se destacan estimación de costos y tiempos, selección de personal, planeación de seguimiento y planeación de control de calidad.
- **Seguimiento y control:** acompaña el proceso de desarrollo, entre sus tareas, se destacan seguimiento y observación de parámetros claves como costo, tiempo, riesgo, así como también los factores que los afectan. De ser necesario, se debe tomar acción correctiva, y las métricas que proveen la información del proceso de desarrollo necesario para el seguimiento.
- **Análisis de terminación:** se debe realizar al final del proyecto de desarrollo, el propósito de esta fase es analizar el desempeño del proceso e identificar las lecciones aprendidas. En procesos iterativos el análisis de terminación se realiza al finalizar cada iteración y se usa para mejorar en iteraciones siguientes.

Proceso de inspección
El objetivo principal de este proceso es detectar defectos en productos de trabajo. Inicialmente utilizado para el código, actualmente usado en todos los tipos de productos de trabajo, y está reconocido como una de las mejores prácticas de la industria ya que mejora tanto la calidad como la productividad. Los defectos introducidos en cualquier etapa del SW, debe eliminarse en cada etapa. Finalmente, las inspecciones pueden realizarse sobre cualquier documento.

Roles y responsabilidades:
- Moderador, quien tiene la responsabilidad general.
- Autor, quien realizó el producto de trabajo.
- Revisor, quien identifica los defectos.
- Lector, lee línea a línea el producto de trabajo para enfocar el progreso de la reunión.
- Escriba, registra las observaciones indicadas.

Planeamiento: aquí se selecciona un equipo de revisión y se identifica al moderador. Se prepara el paquete para la distribución con el producto de trabajo, sus especificaciones, la lista de control con ítems relevantes y estándares.
Preparación y repaso previo: se lleva a cabo en una opcional reunión, aquí se entrega el paquete explicando el propósito de la revisión, y se da un breve intro señalando las áreas de cuidado. En esta etapa todos los miembros del equipo revisan individualmente el producto de trabajo para identificar defectos potenciales en registro individual, se usan listas de control , pautas, y estándares. Idealmente no debería durar más de 2 horas y debería hacerse de corrido.
Reunión de revisión grupal: el propósito aquí es definir la lista final de defectos. Se lleva a cabo mediante un criterio de entrada, donde cada miembro debe haber hecho apropiadamente la revisión individual. En esta reunión, el lector lee línea a línea el producto de trabajo, luego cualquier observación que hubiere (preparada o nueva) es efectuada para luego concluir con una discusión para identificar el defecto. Finalmente, la decisión es registrada por el escriba.
Al final de la reunión: el escriba presenta la lista de defectos/observaciones, si hay pocos defectos el producto de trabajo se acepta; si no, se puede requerir otra revisión. El grupo no propone soluciones, aunque podrían registrarse sugerencias y se prepara un resumen de las inspecciones, que se usa para evaluar la efectividad de la revisión.

El moderador está a cargo de la reunión y juega un rol central:

- Asegura que el foco permanece sobre la identificación de defectos y debe evitar que se prolonguen o se discutan soluciones.
- Lo que se está revisando es el producto de trabajo y no el autor de éste.
- Debe garantizar que la reunión se ejecute ordenada y amigablemente.
- Utiliza el resumen para analizar la efectividad de la revisión.

Corrección y seguimiento: defectos en la lista de defectos son posteriormente corregidos por el autor, y una vez corregidos, , el autor obtiene el visto bueno del moderador o el producto de trabajo se somete a una nueva revisión. Una vez que los defectos/observaciones fueron satisfactoriamente procesados, la revisión finaliza.

Proceso de administración de configuración

La administración de configuración del software (SCM) controla sistemáticamente los cambios producidos y se enfoca en los cambios durante la evolución; los cambios de requerimientos se manejan aparte, requiere tanto disciplina como herramientas.

SCM es usualmente independiente del proceso de desarrollo, los procesos de desarrollo miran el gran esquema, pero no los cambios individuales de ítems/archivos.

A medida que los ítems se producen, se introducen en la SCM, y controla solo los productos del proceso de desarrollo. Durante el proceso los ítems/archivos cambian generando distintas versiones. La administración de configuración debe asegurar que las distintas versiones se combinan apropiadamente sin pérdidas.

Funcionalidades necesarias:

- Recolectar todos 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.

Mecanismos principales:

- Control de acceso.
- Control de versiones.
- Identificación de la configuración.
- Otros mecanismos incluyen: convenciones de nombres, estructuras de directorios, etc.

Ítems de configuración:

- El software comprende muchos ítems (incluyendo ítems accesorios) que son puestos bajo el control de la administración de la configuración.
- Cada ítem es una unidad de modificación y se modifican intensamente: Los cambios a estos ítems se siguen rigurosamente.
- Las distintas versiones de ítems deben combinarse apropiadamente de manera periódica. Baseline: es un arreglo apropiado de ítems de configuración.
- La baseline establece puntos de referencia a lo largo del desarrollo del sistema.
- Captura el estado lógico del sistema y forma la base de cambios posteriores.

Control de versión:

- Fundamental en la administración de configuración.
- Primariamente utilizado en códigos fuentes (pero útil también en otros documentos y otras etapas).
- Ayuda a preservar viejas versiones y a deshacer cambios.
- Herramientas: CVS, SVN, etc

Control de acceso:

- Auxiliado por las mismas herramientas.
- Estas limitan el acceso a personal específico: procedimientos de check-in, checkout.

El proceso: definir actividades que requieren control de cambio. Sus fases principales son:

- **Planeamiento:** identificar ítems; definir la estructura del repositorio; definir control de acceso, puntos de referencia, reconciliación, procedimientos; definir procedimiento de publicación.
- **Ejecución**: Realizar los procedimientos según lo establecido en planeamiento.
- **Auditoría**: Para verificar que no se cometieron errores, ej.: que se mantiene la integridad, que los requerimientos de cambio se realizaron apropiadamente.

Proceso de Administración de Cambios de Requisitos

Los requerimientos pueden cambiar en cualquier momento durante el desarrollo, los cambios no controlados pueden impactar adversamente en el proyecto, tanto en costo como en tiempo. Por eso, los cambios deben permitirse, pero siempre de manera controlada.
Los cambios se inician a través de un requerimiento de cambio, y existe un registro de estos. La administración de procesos se enfoca en la evaluación y mejora del proceso.

Proceso de Administración de Procesos - CMM

Tiene 5 niveles para el proceso de software (el 1ro es ad-hoc), y en cada uno de estos, el proceso tiene ciertas capacidades y establece las bases para pasar al siguiente nivel. Para moverse de un nivel a otro, CMM específica áreas en las cual enfocarse y se utiliza ampliamente en la industria.

Testing

Desperfecto: Un desperfecto de software ocurre si el comportamiento de éste es distinto del esperado/especificado.
Defecto: es lo que causa el desperfecto. Por lo que un defecto es lo mismo que un bug, el cual implica la presencia del defecto.

- Un desperfecto implica la presencia del defecto.
- La existencia del defecto no implica la ocurrencia del desperfecto.
- Un defecto tiene el potencial para causar el desperfecto.
- Qué cosa es considerada un desperfecto depende del proyecto que se está llevando a cabo.

Rol de testing

Las revisiones son procesos humanos: no pueden encontrar todos los defectos, y en consecuencia, habrá defectos de requerimientos, defectos de diseño y defectos de codificación dentro del código. Estos defectos se identifican por medio del testing, por ende, este proceso juega un rol crítico cuando se trata de garantizar calidad.

- Durante el testing, un programa se ejecuta siguiendo un conjunto de casos de test.
- Si hay desperfectos en la ejecución de un test, entonces hay defectos en el software.
- Si no ocurren desperfectos, entonces la confianza en el software crece.
- No podemos negar la presencia de defectos.
- Los defectos se detectan a través de los desperfectos.
- Para detectar los defectos debemos causar desperfectos durante el testing.
- Para identificar el defecto real (que causa el desperfecto) debemos recurrir al debugging.

Oráculo de tests
Para verificar la ocurrencia de un desperfecto en la ejecución de un caso de test, necesitamos conocer el comportamiento correcto, para este caso necesitamos un oráculo de test. Por lo tanto, el oráculo de test es un ente el cual conoce los resultados esperados del test. Y sirve para poder comparar el resultado obtenido (developer) con el esperado (oráculo).

Casos de test y criterios de selección
Si existen defectos, deseamos que los casos de test los evidencien a través de fallas. Deseamos construir un conjunto de casos de test tal que la ejecución satisfactoria de todos ellos implique la ausencia de defectos. Además, como el testing es costoso, deseamos que sea un conjunto reducido.
Estos dos deseos son contradictorios, por ende se usa algún criterio de selección de test, que especifica las condiciones que el conjunto de casos de test debe satisfacer con respecto al programa y/o a la especificación.

La psicología del testing es importante, debe revelar los defectos y los casos del test deben ser destructivos. Existen dos enfoques para diseñar casos de test, de caja negra (funcional) o de caja blanca (estructural) y ambos son complementarios.

Testing de caja negra
El SW a testear se trata como una caja negra, que tiene una entrada, una implementación bajo test y una salida.

- La especificación de la caja negra está dada.
- Para diseñar los casos de test, se utiliza el comportamiento esperado del sistema, es decir, los casos de test se seleccionan sólo a partir de la especificación.
- No se utiliza la estructura interna del código.

Se toma como premisa que el comportamiento esperado está especificado, entonces solo se definen test para ese comportamiento. Para el testing de módulos, la especificación producida en el diseño define el comportamiento esperado, mientras que para el testing del sistema es la SRS quien define el comportamiento esperado.
El testing funcional más minucioso es el exhaustivo, en donde el software está diseñado para trabajar sobre un espacio de entrada, y se testea el mismo con todos los elementos del espacio de entrada. No es viable ya que es demasiado costoso, por lo que se necesitan mejores métodos para seleccionar los casos de test.

Particionado por clase de equivalencia

Se divide el espacio de entrada en estas clases. Parte de la **idea** en la que si el software funciona para un caso de test en una clase, muy probablemente funcione de la misma manera para todos los elementos de esa clase. La obtención del particionado ideal es imposible, por eso se aproxima identificando las clases para las cuales se especifican distintos comportamientos. Su **base lógica** es que la especificación requiere el mismo comportamiento en todos los elementos de una misma clase. Es muy probable que el software se construya de manera tal que falle para todos o para ninguno. Cada condición especificada como entrada es una clase de equivalencia y para lograr robustez, se deben armar clases de equivalencias para entradas inválidas. Cuando el rango completo no se trate uniformemente es necesario dividir en clases.

También se deben considerar las clases de equivalencia de los datos de salida. Generar los casos de test para estas clases eligiendo apropiadamente las entradas.

Una vez elegidas las clases de equivalencia, se deben seleccionar los casos de test:

1. Seleccionar cada caso de test cubriendo tantas clases como sea posible.
2. O dar un caso de test que cubra a lo sumo una clase válida por cada entrada.

Además de los casos de test separados por cada clase inválida.

Análisis de valores límites

Los programas generalmente fallan sobre valores especiales, estos valores usualmente se encuentran en los límites de las clases de equivalencia. Los casos de test que tienen valores límites tienen alto rendimiento o también denominados casos extremos. Un caso de test de valores límites es un conjunto de datos de entrada que se encuentra en el borde de las clases de equivalencias de la entrada o la salida.

Para cada clase de equivalencia:

→Elegir valores en los límites de la clase.

→Elegir valores justo fuera y dentro de los límites.

→Considerar las salidas también y producir casos de test que generen salidas sobre los límites.

En primer lugar se determinan los valores a utilizar para cada variable.

Si la entrada tiene un rango definido => hay 6 valores de límite más un valor normal (total: 7).

1. Ejecutar todas las combinaciones posibles de las distinta variables (si hay n → 7n casos de test!).
2. Seleccionar los casos límites para una variable y mantener las otras en casos normales y considerar el caso de todo normal (total 6n + 1).

Grafo de causa-efecto

- Los análisis de clase de equivalencia y valores límites consideran cada entrada separadamente.
- Para manipular las entradas distintas combinaciones de las clases de equivalencia deben ser ejecutadas.
- La cantidad de combinaciones puede ser grande: si hay n condiciones distintas en la entrada que puedan hacerse válidas o inválidas => 2n clases de equivalencia.
- El grafo de causa-efecto ayuda a seleccionar las combinaciones como condiciones de entrada.

Es necesario identificar causas y efectos en el sistema.

- Causa, distintas condiciones en la entrada que pueden ser verdaderas o falsas.

- Efecto, distintas condiciones de salidas (V/F también).

Hay que identificar cuáles causas pueden producir qué efectos; las causas se pueden combinar, siendo estos los nodos en el grafo. Las aristas determinan dependencias y pueden ser positivas o negativas. Existen nodos and y or para combinar la causalidad.
A partir del grafo de causa-efecto se puede armar una tabla de decisión que lista las combinaciones de condiciones que hacen efectivo cada efecto. La tabla de decisión puede usarse para armar los distintos casos de test.

Testing de a pares

- Usualmente muchos parámetros determinan el comportamiento del sistema.
- Los parámetros pueden ser entradas o seteos y pueden tomar distintos valores (o distintos rangos de valores).
- Muchos defectos involucran sólo una condición (defecto de modo simple).
- Los defectos de modo simple pueden detectarse verificando distintos valores de los distintos parámetros.
- Si hay n parámetros con m (tipos de) valores c/u, podemos testear cada valor distinto en cada test por cada parámetro, es decir, m casos de test en total.

Pero no todas los defectos son de modo simple, el software puede fallar en combinaciones. Los defectos de modo múltiple se revelan con casos de test que contemplen las combinaciones apropiadas, esto se denomina test combinatorio.
El test combinatorio no es factible, ya que n parámetros / m valores →n * m casos de test, lo que llevaría mucho tiempo de testeo. Se investigó que la mayoría de tales defectos se revelan con la interacción de pares de valores, y la mayoría de los defectos tienden a ser de modo simple o de modo doble, y para los de modo doble necesitamos ejercitar cada par, lo que se denomina testing de a pares.
En testing de a pares, todos los pares de valores deben ser ejercitados.
Si n parámetros / m valores, para cada par de parámetros tenemos m*m pares:

- 1er parámetro tendrá m*m contra n-1 otros
- 2do parámetro tendrá m*m contra n-2 otros
- 3er parámetro tendrá m*m contra n-3 otros etc.

Total: m2 * n * (n-1) / 2.

- Un caso de test consiste en algún seteo de los n parámetros.
- El menor conjunto de casos de test se obtiene cuando cada par es cubierto sólo una vez.
- Un caso de test puede cubrir hasta (n-1)+(n-2)+...= n*(n-1)/2 pares.
- En el mejor caso, cuando cada par es cubierto exactamente una vez, tendremos m2 casos de test distintos que proveen una cobertura completa.
- La generación del conjunto de casos de test más chico que provea cobertura completa de los pares no es trivial.
- Existen algoritmos eficientes para generación de casos de test que pueden reducir el esfuerzo de testing considerablemente.
- El testing de a pares es un enfoque práctico y muy utilizado en la industria.

Testing basado en estados

Algunos sistemas no tienen estados: para las mismas entradas, se exhiben siempre las mismas salidas. En muchos sistemas, el comportamiento depende del estado del sistema, i.e. para la misma entrada, el comportamiento podría diferir en distintas ocasiones, el

comportamiento y la salida depende tanto de la entrada como del estado actual del sistema. Este estado representa el impacto acumulado en las entradas pasadas, el testing basado en estado está dirigido a tales sistemas.

- Un sistema puede modelarse como una máquina de estados.
- El espacio de estados puede ser demasiado grande (es el producto cartesiano de todos los dominios de todas las variables).
- El espacio de estados puede particionarse en pocos estados, cada uno representando un estado lógico de interés del sistema.
- El modelo de estado se construye generalmente a partir de tales estados.

Un modelo de estados tiene cuatro componentes:

- **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 actual.

El modelo de estado muestra la ocurrencia de las transiciones y las acciones que se realizan, usualmente este se construye a partir de las especificaciones o requerimientos. El desafío más importante es, a partir de la especificación/requerimientos, **identificar el conjunto de estados que captura las propiedades claves pero es lo suficientemente pequeño como para modelar.**
El modelo de estado puede crearse a partir de la especificación o del diseño.
En el caso de los objetos, los modelos de estado se construyen usualmente durante el proceso del diseño. L**os casos de test se seleccionan con el modelo de estado y se utilizan posteriormente para testear la implementación.**
Criterios para generar los casos de test:

- **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).

El test basado en estados **se enfoca en el testing de estados y transiciones**, en donde **se testean distintos escenarios** que de otra manera podrían pasarse por alto. El modelo de estado **se realiza usualmente luego de que la información de diseño se hace disponible**.
En este sentido, se habla a veces de testing de caja gris (dado que no es de caja negra puro).

Testing de caja blanca
Se enfoca en el código y su objetivo es ejecutar las distintas estructuras del programa con el fin de descubrir errores, por ende, los casos de test derivan del código. Existen varios criterios para seleccionar el conjunto de casos de test.

Criterio basado en el flujo de control: observa la cobertura del grafo de flujo de control. Aquí se considera al programa como un grafo de flujo de control. Los nodos representan bloques de código o conjuntos de sentencias que siempre se ejecutan juntas, mientras que una arista (i, j) representa una posible transferencia de control del nodo i al j.
Suponemos la existencia de un nodo inicial/final y un camino es una secuencia del nodo inicial al final.
→ **Cobertura de sentencia:** cada sentencia se ejecuta al menos una vez durante el testing, el conjunto de caminos ejecutados en el mismo, debe incluir todos los nodos. Tiene la limitación de que puede no requerir que una decisión evalúa a falso en un if si no hay else. Además, no es posible garantizar 100% de cobertura debido a que puede haber nodos inalcanzables.
Ejemplo: `abs(x): if(x >=0) then x := -x
return(x)`
El conjunto de casos de test { (x = 0, 0) } tiene el 100% de cobertura pero el error pasa desapercibido.
→ **Cobertura de ramificaciones:** cada arista debe ejecutarse al menos una vez en el testing, o cada decisión debe ejercitarse como verdadera y como falsa. 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.
Ejemplo: `abs(x): if(x <= 0) then x := -x
return(x)`
Cobertura de Sentencias: {(x=-1, 1)}
Cobertura Ramificaciones: {(x=1, 1); (x=0, 0)}
→ **Cobertura de caminos:** todos los posibles caminos del estado inicial al final, deben ser ejercitados. Esta cobertura implica cobertura de bifurcación, pero existe el problema de que la cantidad de caminos puede ser infinita (loops). Notar además que puede haber caminos que no son realizables.

Criterio basado en el flujo de datos: observa la cobertura de la relación definición-uso en las variables. Aquí se construye un grafo de definición etiquetado apropiadamente al grafo de flujo de control.

- **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

El grafo de def-uso se construye asociando variables a nodos y aristas del grafo de flujo de control.

- Por cada nodo i, def(i) es el conjunto de variables para el cual hay una definición en i.
- Por cada nodo i, c-use(i) es el conjunto de variables para el cual hay un uso-c.
- Para una arista (i,j), p-use(i,j) es el conjunto de variables para el cual hay un uso-p.

Un camino de i a j se dice libre de definiciones con respecto a una var x si no hay definiciones de x en todos los nodos intermedios.

Criterios:

- Todas las definiciones: por cada nodo i y cada x en 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.

Testing incremental
Los objetivos del testing son detectar tantos defectos como sea posible a un bajo costo. Pero tiene objetivos contrapuestos, ya que incrementar el testing permite encontrar más efectos pero a la vez incrementa el costo. La idea del testing incremental es agregar partes no testeadas incrementalmente a la parte ya testeada, este es incremental es esencial para conseguir los objetivos antedichos.

Testing de integración
Se enfoca en la interacción de módulos de un subsistema. Estos módulos que ya fueron testeados de manera unitaria se combinan para formar subsistemas que son sujetos a testing de integración. Los casos de tests deben generarse con el objeto de ejercitar de distinta manera la interacción entre los módulos, existiendo un énfasis en el testing de las interfaces entre los módulos, se podría omitir si el sistema no es muy grande.

Testing de regresión
Se realiza cuando se introduce algún cambio al software y verifica que las funcionalidades previas continúen funcionando bien. Se necesitan los registros previos para poder comparar. Los tests deben quedar apropiadamente documentados y es importante priorizar los casos de tests necesarios cuando el test suite completo no pueda ejecutarse cada vez que se realiza un cambio.

Registros de defectos y seguimiento
Un software grande puede tener miles de defectos, encontrados por muchas personas distintas. Usualmente, las personas que los corrigen no son las mismas que lo encontraron y debido a este gran alcance el registro y la corrección de los defectos no puede realizarse informalmente. Los defectos encontrados usualmente se registran en un sistema seguidor de defectos (“tracking”) que permite rastrearlos hasta que se “cierren”. El registro de defectos y su seguimiento es una de las mejores prácticas en la industria.
Un defecto en un proyecto de software tiene su propio ciclo de vida, y durante este se registra información sobre el defecto en las distintas etapas para ayudar al debugging y al análisis. Los defectos se categorizan generalmente en algunos tipos; los tipos de los defectos son registrados. Algunas categorías:

- Funcional.
- Lógica.
- Estándares.
- Asignación.
- Interfaz de usuario.
- Interfaz de componente.
- Desempeño.
- Documentación.

También se registra la severidad del defecto en términos de su impacto en el software.

- La severidad es útil para priorizar la corrección. Una posible categorización:
- **Crítico:** puede demorar el proceso; afecta a muchos usuarios.

- **Mayor**: tiene mucho impacto pero posee soluciones provisorias; requiere de mucho esfuerzo para corregirlo, pero tiene menor impacto en el cronograma.
- **Menor**: defecto aislado que se manifiesta raramente y que tiene poco impacto.
- **Cosmético**: pequeños errores sin impacto en el funcionamiento correcto del sistema.

Idealmente, todos los defectos deben cerrarse. Algunas veces, las organizaciones entregan software con defectos conocidos.
Las organizaciones tienen estándares para determinar cuándo un producto se puede entregar. El registro de defectos puede utilizarse para seguir la tendencia de cómo ocurren los arribos de los defectos y sus correcciones.

Planeamiento del proyecto de software
El planeamiento es la **actividad principal que produce un plan el cual forma la base del seguimiento, se realiza antes de comenzar con el desarrollo propiamente dicho, y requiere como entrada los requerimientos y la arquitectura**. Durante esta etapa, se planean todas las tareas que la administración del proyecto necesita realizar. Durante el seguimiento y control, el plan es ejecutado y actualizado.

Tópicos más importantes:
1. Planeamiento del proceso.
2. Estimación del esfuerzo.
3. Estimación de tiempos y recursos.
4. Plan para la administración de la configuración.
5. Planeamiento de la calidad.
6. Administración del riesgo.
7. Plan para el seguimiento del proyecto.

Estimación del esfuerzo
Dado un conjunto de requerimientos es deseable o necesario saber cuánto costará en tiempo y dinero el desarrollo del sw. El esfuerzo se mide usualmente en personas/mes. Considerando la recarga de costos por persona, la estimación del esfuerzo puede convertirse directamente en costo. Esta estimación es clave para el planeamiento, pues de ello dependen los tiempos, costos y recursos. Muchos problemas en la ejecución del proyecto se deben a una estimación inapropiada, y es importante para factibilidad, análisis costo-beneficio, efectuar ofertas, entre otros. La estimación mejora a medida que se incrementa la información sobre el proyecto, y la más temprana es la más propensa a inexactitud que la estimación avanzada en el proyecto.
→ **Método COCOMO:** estimación con un error dentro del 20% en el 68% de los casos.

Construcción de modelos : un modelo intenta determinar la estimación del esfuerzo a partir de los valores de ciertos parámetros, tales valores dependen del proyecto. El modelo reduce el problema de estimar el esfuerzo del proyecto al de estimar ciertos parámetros claves del proyecto. Estos últimos deben poder medirse en etapas muy tempranas del proyecto, existiendo los enfoques top-down y bottom-up. **Se podría decir que es una función, y el factor principal es el tamaño del proyecto**.
→ **Estimación top-down:** determinar el esfuerzo total y luego calcular el esfuerzo de cada parte del proyecto.
→ Tamaño global.

→ Esfuerzo = a * tamaño^b a, b = analisis de regresion proyectos pasados.

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

→ **Estimación bottom-up:**

1. Identificar los módulos del sistema y clasificarlos como simples, medios, o complejos.
2. Determinar el esfuerzo promedio de codificación para cada tipo de módulo.
3. Obtener el esfuerzo total de codificación en base a la clasificación anterior y al 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.

Modelo COCOMO

Tiene un enfoque top-down, utiliza el tamaño ajustado con algunos factores. El procedimiento se descompone en:

a. Obtener el estimador inicial usando el tamaño : E_i = a * tamaño^b

Sistema	a	b
Orgánico (simple)	3.2	1.05
Semi-rigido	3.0	1.12
Rígido (ambicioso y novedoso)	2.8	1.20

b. Determinar un conjunto de 15 factores de multiplicación representando distintos atributos:

Atributos SW	Atributos HW	Atributos personal	Atributos proyecto
RELY: confiabilidad	**TIME: uso CPU**	ACAP: calificación de analistas	MODP: prácticas modernas de prog.
DATA: bases de datos	**STOR: uso memoria**	AEXP: experiencia del personal	TOOL: herramientas desarrollo SW
CPLX: complejidad	**VIRT: máquina virtual**	PCAP: calificación de los programadores	SCED: cumplimiento de la planificación
	TURN: modelo de explotación	LEXP: exp. en leng	

c. Ajustar el estimador de esfuerzo escalando según el factor de multiplicación final.
→ Esfuerzo = E_i * ∏(f_k) f_k factor de ajuste.

d. Calcular el estimador de esfuerzo de cada fase principal.

Fase	Tamaño			
	Pequeño 2 KLOC	Intermedio 8 KLOC	Medio 32 KLOC	Grande 128 KLOC
Diseño del producto	16%	16%	16%	16%
Diseño detallado	26%	25%	24%	23%
Codificación y test de unidad	42%	40%	38%	36%
Integración y test	16%	19%	22%	25%

Planeamiento de la administración de la configuración del software
Se deben identificar los ítems de la 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.
Algunas actividades:

- Identificar los ítems de configuración (IC).
- Definir un esquema de nomenclatura para cada IC.
- Definir la estructura de directorios necesaria.
- Definir el procedimiento para el versionado y los métodos para rastrear los cambios en los IC.
- Definir las restricciones de acceso.
- Definir los procedimientos para el control de cambios.
- Identificar y definir las responsabilidades del administrador de configuración.
- Identificar los puntos en los que se crearán las “baselines”.
- Definir el procedimiento de backup.
- Definir el procedimiento de “release”.

Planeamiento de control de calidad
El objetivo básico es entregar un SW de calidad, se utiliza una medida de calidad estándar para ello, que es la densidad de defectos entregados. Un defecto causa que el sw se comporte de manera inconsistente, por ende el objetivo del proyecto es entregar un SW con baja densidad de estos. **Su propósito es especificar las actividades que se necesitan realizar para identificar y eliminar defectos.**
→ **Introducción y eliminación de errores:** el desarrollo de SW es una actividad altamente dependiente de personas, por lo que es propensa a errores. Los defectos se introducen en cualquier etapa, y como el objetivo de calidad es la baja densidad de defectos, estos deben ser eliminados. Esto se realiza fundamentalmente mediante las actividades de control de calidad (QC) incluyendo revisiones y testing. **Existen tres enfoques para la administración de control de calidad:**

- **Enfoque cuantitativo:** va más allá de requerir que se ejecute el procedimiento. Analiza los datos recolectados de los defectos y establece juicios sobre la calidad (métricas, densidad de defectos). La información del pasado es muy importante para la predicción de defectos, algunos parámetros claves son tasas de introducción y eliminación de defectos.
- **Enfoque ad hoc:** se hacen test y revisiones de acuerdo a cuándo y cómo se necesiten.

- **Enfoque de procedimiento:** el plan define qué tareas de QC se realizarán y cuando, siendo las principales revisión y testing. Provee procedimientos y lineamientos a seguir en el testing y en la revisión. Durante la ejecución del proyecto asegurar el seguimiento del plan y los procedimientos.

Administración de riesgos

Cualquier proyecto puede fallar debido a eventos no previstos, la administración de riesgo es un intento de minimizar las chances de fallas. El riesgo es cualquier condición o evento de ocurrencia incierta que puede causar la falla del proyecto. **Tales eventos no son comunes, por lo que el objetivo de la administración de riesgo es minimizar el impacto de la materialización de los riesgos.**

→ **Identificación del riesgo:** identificar los posibles riesgos del proyecto, es decir, aquellos eventos que podrían ocurrir y causar la falla del proyecto. La forma de hacerlo es mediante: listas de control, experiencias pasadas, brainstorming, entre otros.

→ **Factores de riesgo más importantes:**

- Personal: insuficiente o inapropiadamente entrenado.
- Tiempos y costos irreales.
- Componentes externos: incompatibles o de baja calidad.
- Discrepancia con los requerimientos; “gold plating”.
- Discrepancia con la interfaz del usuario.
- Arquitectura, desempeño, calidad: inadecuada o insuficiente evaluación.
- Flujo continuo en los cambios de requerimientos.
- Software legado.
- Tareas desarrolladas externamente: inadecuadas o demoradas.

→ **Análisis de riesgos y definición de prioridades:** la cantidad de riesgos puede ser grande, por lo que se deben priorizar para enfocar la atención en las áreas de alto riesgo. Para ello, es importante establecer la probabilidad de materialización de los riesgos identificados y la pérdida que éstos originaron. Ordenar de acuerdo al “valor de exposición al riesgo”, (RE = prob. ocurrencia indeseada * impacto ocurrencia indeseada). Por lo que, RE es el valor esperado de la pérdida debido a un riesgo y hay que generar planes para tratar con el.

→ Ocurrencias: Bajas, Medias o Altas.
→ Impactos: Bajos, Medios o Altos.
→ Riesgos: AA y AM/MA.

→ **Control de riesgos:** si es posible evitarlo, se evita. En otros casos planear y ejecutar los pasos necesarios para mitigarlos, como definir las acciones a seguir en el proyecto de manera que, si el riesgo se materializa, su impacto sea mínimo (involucra costo extra).

Ejemplos de mitigación de riesgo: “Demasiados cambios de requerimientos”.

- Convencer al cliente que los cambios de requerimientos tienen un alto impacto en los tiempos.
- Definir un procedimiento para cambios de requerimientos.
- Mantener el impacto acumulado de los cambios y hacérselo notar al cliente.
- Negociar pagos del esfuerzo real.

→ **Plan de mitigación de riesgos:** incluye los pasos a realizar, estos deben planificarse en el tiempo y ejecutarse. Son distintos de los que se deben realizar si el riesgo se materializa, los cuales se efectúan solo si es necesario. Los riesgos deben revisarse periódicamente.

Planificación del seguimiento del proyecto

- El plan de administración del proyecto es meramente un documento que sirve como guía.
- Éste debe ejecutarse.
- Para asegurar que la ejecución se realiza como se planeó, ésta debe seguirse y controlarse.
- El seguimiento (monitoring) requiere de mediciones y métodos que las interpreten.
- El plan de seguimiento incluye:
  → Planificar qué medir, cuándo y cómo.
  → Cómo analizar y reportar estos datos.

Principales medidas

- Tiempo (cronograma): es la más importante de las medidas.
- Esfuerzo: es el principal recurso.Usualmente se sigue (track) a través de herramientas de reporte de esfuerzo.
- Defectos: determinan calidad. Usualmente se siguen con herramientas de registros y seguimientos.
- Tamaño: mucha información se normaliza respecto al tamaño.

Seguimiento del proyecto (monitoring and tracking)

Objetivo: hacer visible la ejecución del proyecto de manera de realizar acciones correctivas cuando sea necesario con el fin de asegurar el éxito del proyecto.

- Distintos niveles de seguimientos:
  → nivel de actividad,
  → reportes de estado,
  → análisis de metas parciales.
- Las mediciones proveen los datos para estos seguimientos.

Seguimiento a nivel de actividad: asegura que cada actividad se realiza apropiadamente y a tiempo, realizando diariamente por los administradores de proyecto. Una tarea realizada se marca con 100%; las herramientas pueden determinar el estado de las tareas de más alto nivel.

Reporte de estado: usualmente se prepara semanalmente y contiene un resumen de actividades completadas y pendientes, junto con cuestiones que necesitan atención o deben ser resueltas.

Análisis de metas parciales: se realiza una mayor revisión con cada meta parcial. Análisis de esfuerzos y tiempos reales vs. estimados. Si la desviación es amplia → medidas correctivas, contiene una revisión de los riesgos.