**Gestión de Configuración de Software**

La Gestión de Configuración de Software (SCM) es una disciplina diseñada para identificar y definir los elementos en el sistema que probablemente cambien, controlando el cambio de estos elementos a lo largo de su ciclo de vida, estableciendo relaciones entre ellos, definiendo mecanismos para gestionar distintas versiones de estos elementos, y auditando e informando de los cambios realizados.

Su propósito es establecer y mantener la integridad de los productos del proyecto de software a lo largo de su ciclo de vida.

Es necesaria porque los requerimientos del sistema siempre cambian durante su desarrollo y su uso, y se tienen que incorporar estos requerimientos en nuevas versiones del sistema y si no se gestionan los sistemas en evolución es fácil perder la pista de cuáles cambios y versiones del componente se incorporaron en cada versión del sistema, se pueden producir pérdidas de componentes, necesidad de realizar doble mantenimiento de los mismos, superposición de cambios, cambios no validados, etc.

Es una actividad “paraguas”, transversal a todo el proyecto con aplicación en las diferentes disciplinas. Es continua ya que aparece desde el primer momento que empieza el proyecto y continúa hasta que el sistema deja de utilizarse.

**Integridad del producto**

Se considera que un producto tiene integridad cuando:

* Satisface las necesidades del cliente, es decir, que hace lo que se esperaba que hiciera.
* Puede ser fácil y completamente rastreado durante su ciclo de vida (de las diferentes etapas, la relación que hay entre los artefactos).
* Satisface criterio de performance.
* Cumple con sus expectativas de costo.

**Elementos de SCM**

**1 – Identificación de los Ítems de Configuración.**

Consiste en identificar todos aquellos artefactos que forman parte del producto que pueden sufrir cambios o necesitan ser compartidos entre los miembros del equipo y sobre los cuales necesitamos conocer su estado y evolución (Determinar, dentro de todo el conjunto de objetos, cuáles van a ser IC). Pueden ser: requerimientos, documentos de diseño, código fuente, código ejecutable, etc.

Hay una etapa al inicio del proyecto en donde se definen cuáles van a ser lo criterios para determinar los IC, lo que se va a ver reflejado en el plan de gestión de configuración, el cual a su vez, también es un IC.

**2 – Control de cambios.**

El software va a estar sujeto a cambios durante el desarrollo del mismo y una vez que esté finalizado también.

El control de cambios tiene su origen en un requerimiento de cambio a uno o varios IC que se encuentran en una línea base y su objetivo es tener un procedimiento formal que me permita dar soporte a esos cambios y que involucre diferentes actores y una evaluación del impacto del cambio.

Existe un comité de control de cambios que va a estar conformado por representantes de todas las áreas en donde ese cambio va a impactar y son los encargados de analizar el requerimiento de cambio sobre los IC y de determinar cuándo es conveniente llevarlo a cabo, aspectos técnicos, efectos colaterales, costos del proyecto, impacto general.

Petición de cambio: solicitud formal que se presenta ante el CCC, la cual describe un problema de software, una mejora solicitada o un cambio en los requerimientos de software.

**REPOSITORIOS:** es un depósito de información que contiene los ítems de configuración, manteniendo la historia de cada uno de los IC con sus atributos y relaciones (meta información).

Se utilizan para hacer evaluaciones de impactos de los cambios propuestos. Pueden ser una o varias bases de datos. Posee herramientas de automatización, integración y generación de informes.

* Repositorios centralizados: un servidor contiene todos los archivos con sus versiones. Los administradores tienen mayor control sobre el repositorio pero si falla el servidor estamos al horno.
* Repositorios descentralizados: cada cliente tiene una copia exactamente igual del repositorio completo. Si un servidor muere solo es cuestión de copiar y pegar. Posibilita otros workflows no disponibles en el modelo centralizado.

**3 – Auditorías de configuraciones.**

* **Auditoría física.**

Asegura que lo que está indicado para cada IC en la línea base o en la actualización se ha alcanzado realmente.

**Verifica** que un ítem es integro en cuanto a su descripción. Se hace entre ítems, se fija si el ítem es correcto o no. Se realiza primero por ser de verificación.

* **Auditoría funcional.**

Evaluación independiente de los productos de software, verificando que la funcionalidad y performance reales de cada ítem de configuración sean consistentes con la especificación de los requerimientos.

**Validan** que el producto final esté de acuerdo con los requerimientos. La puedo ver a través de la matriz de trazabilidad (puedo saber el CU que se implementó, donde se implementó, etc.). Usa la rastreabilidad como herramienta.

Las auditorías deben ser objetivas e independientes, razón por la cual deben ser llevadas a cabo por personas que no formen parte del equipo de desarrollo. Son un proceso de control cuyo objetivo es revisar que las cosas del proyecto estén funcionando como se dijo que iba a funcionar y que todo lo relacionado a la gestión de configuración esté plasmado en el plan.

Las auditorías de configuración deben realizarse desde el principio del proyecto ya que su postergación a etapas posteriores puede llegar a hacerlo fracasar y deben planificarse porque cuestan tiempo y dinero. Se hace sobre una línea base etiquetada (porque para controlar necesito tener contra que comparar).

Suministra visibilidad y rastreabilidad del ciclo de vida del producto de software, es decir, que con ella nos vamos asegurando de que cada ítem es consistente y que está de acuerdo con un requerimiento.

**4 – Generación de informes.**

Permite mantener los registros de la evolución del sistema. Maneja mucha información y salidas por lo que se suele implementar dentro de procesos automáticos. Incluye reportes de la rastreabilidad de todos los cambios realizados a las líneas base durante el ciclo de vida.

Permiten obtener información, a través de las herramientas automatizadas, sobre los reportes y me permite tomar decisiones.

Permite saber cuál es el estado de un ítem, si un requerimiento de cambio fue aprobado o no por el comité, que versión del ítem implementa un cambio aprobado (saber cuál es el componente que tiene la mejora), cuál es la diferencia entre una versión y otra, etc.

**La Configuración**

Un conjunto de ítems de configuración con su correspondiente versión en un momento determinado.

Foto de un momento dado de un repositorio que contiene todos los ítems de configuración que se encuentran en ese momento.

**Línea Base**

Es un conjunto de ítems de configuración con sus correspondientes estados en un momento de tiempo, que han alcanzado una cierta configuración formalmente aprobada, que me sirve de referencia para futuras versiones y que solamente puede ser modificada a través de un procedimiento formal de control de cambios.

Los criterios a tener en cuenta para crear una línea base se establecen en el plan de gestión de configuración. Pueden ser creadas después de hacer una validación con el cliente, después de las pruebas, cuando voy terminando los distintos workflows, después de determinar los requerimientos, etc.

Las líneas base pueden ser identificadas a través de etiquetas que tienen un nombre que hace referencia a lo que se esté tratando en la línea. Y puede haber tantas líneas bases como queramos.

**Ramas**

Las ramas son bifurcaciones del desarrollo principal sobre las que se pueden ir aplicando cambios de manera aislada y que después se integran con la rama principal.

Constituye una potente herramienta que flexibiliza la forma en que los colaboradores cooperan con el proyecto ya que facilitan el aislamiento de los ítems para que puedan trabajar sin que nadie trabaje sobre lo mismo.

Todas las ramas deberían eventualmente integrarse a la principal o ser descartadas. La acción de integrar los cambios a la rama principal se realiza a través de la operación merge.

**Plan de Gestión de Configuración**

Tiene cuatro secciones que corresponden a cada una de las actividades del SCM…

**Testing de Software**

Es el proceso de analizar un ítem de software para detectar las diferencias entre lo existente y lo requerido y evaluar las características de los elementos de software.

Es un proceso destructivo cuyo objetivo es tratar de encontrar defectos, que se asumen que existen, en el software. Un testing se considera exitoso cuando encuentra defectos.

Es un proceso de control que sirve a:

* Proceso de validación: ver si el producto es correcto, es decir, si cumple con los requerimientos o no.
* Proceso de verificación: ver si el producto funciona bien.

Lo que se quiere lograr con el testing no es romper el producto, si no evitar que el cliente encuentre los defectos. Debe planificarse porque representa de un 30 a un 50 por ciento del costo de un software confiable.

**Aseguramiento de Calidad vs Control de Calidad**

El **aseguramiento de calidad** es un proceso que arranca desde el momento cero ya que la calidad del producto depende de las tareas realizadas durante todo el proceso, no puede “inyectarse” al final.

El **control de calidad** es hacer testing, siempre se hace y no se negocia. Se hace al final y por eso se dice que el testing llega tarde.

**Principios del Testing**

* El testing no es exhaustivo, es decir que una cobertura del 100% no se puede lograr porque es caro en recursos.
* Muestra presencia de defectos, no asegura que no existan.
* Se debe adelantar el testing lo más que se pueda, porque detectar errores en forma temprana ahorra esfuerzos, tiempo y recursos. Por ejemplo, estoy haciendo testing temprano si cuando estoy escribiendo un requerimiento funcional ya me voy imaginando que se puede probar.
* Agrupamiento de defectos: el 80% de los defectos se acumulan en el 20% de las funcionalidades y es muy importante saber identificar esas funcionalidades críticas. Por esta razón también es que el testing no es exhaustivo.
* Paradoja del Pesticida: quiere decir que se debe cambiar el testeador porque se acostumbra y comienza a pasar por alto defectos.
* Es de dependiente del contexto (en mi casa funcionaba).
* Un programador debería evitar probar su propio código.
* No debe planificarse el esfuerzo del testing sobre la suposición de que no se van a encontrar defectos.

**¿Cuánto testing es suficiente?**

El **criterio de aceptación** es lo que comúnmente se usa para resolver el problema de determinar cuándo una determinada fase de testing ha sido completada, y consiste en acuerdos que se hacen con el equipo y los clientes sobre lo que el software debería hacer y cómo debe funcionar para que el cliente lo acepte. Mientras más temprano se defina éste criterio, mejor. Algunos parámetros que se suelen utilizar son los costos, % de test corridos sin fallas, fallas predichas aún permanecen en el software, que no haya defectos de una determinada severidad en el software.

También depende de la evaluación del nivel de riesgo y los costos asociados al proyecto (relación costo beneficio).

**Error vs Defecto**

La diferencia entre error y defecto radica en el momento en el que se encuentran. Cuándo, por ejemplo, un error cometido en la etapa de requerimiento se encuentra recién en el testing, entonces ya es un defecto.

El objetivo de la QA es la detección temprana de errores antes de que se conviertan en defectos. Mientras más viejos son los errores más caros son. El testing encuentra defectos.

**Severidad:** cuan grave es el error o defecto que encontramos.

1. Bloqueante: es invalidante, no puedo hacer nada.
2. Crítico: el sistema anda pero la consecuencia del defecto es grave.
3. Mayor:
4. Menor: son pequeñas advertencias pero que te dejan trabajar.
5. Cosmético:

**Prioridad:** cuanto le impacta ese error o defecto al cliente.

1. Urgencia.
2. Alta.
3. Media
4. Baja.

No existe una correspondencia uno a uno entre las severidades y prioridades.

**Niveles de Prueba**

* **Testing unitario.**
* **Testing de integración o de interfaces.**
* **Testing de sistema.**
* **Testing de aceptación.**

**Testing Unitario**

Es el nivel más bajo de testing y consiste en probar los componentes del programa, como métodos o clases de objetos, de manera independiente. Garantiza que el componente funciona y que hace lo que tiene que hacer (verificación y validación).

El testing unitario lo realiza el mismo programador para asegurarse que lo hizo funciona y para que a medida que encuentra errores los corrija, sin dejar constancia oficial de los incidentes, y de esa manera abaratar los costos. Además le permite entender mejor lo que tiene que implementar y de esa manera evitar cometer el error de nuevo.

**Testing de integración o de interfaces**

Apunta a ver cómo funcionan los componentes como un todo, habiendo ya probado esos componentes de manera individual con el testing unitario. Se ve como se ensambla un componente con otro y se asegura que eso funcione. Se prueban las interfaces entre los componentes. Aquellas funcionalidades que tienen mayor riesgo son las que se deben probar primero.

Hay dos estrategias para trabajar la integración:

* De abajo hacia arriba: consiste en tomar un componente y ensamblarlo con otro, lo prueba y así sucesivamente.
* De arriba hacia abajo: consiste en tomar un build y lo va desgranando para probarlo.

Lo que más se usa generalmente es la estrategia de abajo hacia arriba, aunque lo ideal es usar una estrategia que combine la dos.

La mejor manera de hacer un testing exitoso es ir integrando de a poco y no hacerlo todo a último momento para evitar encontrarnos de que todos esos componentes integrados no funcionan.

**Testing de Sistema**

Es la prueba realizada cuando una aplicación está funcionando como un todo y trata de determinar si el sistema en su globalidad opera satisfactoriamente (recuperación de fallas, seguridad y protección, stress, performance, etc.). Está basado en los aspectos de negocio.

Se basa tanto en los requerimientos funcionales como en los no funcionales del sistema.

Cuándo se habla de sistema se hace referencia a los que vamos teniendo del mismo de acuerdo al ciclo de vida que fue planteado. Por ejemplo, si el ciclo de vida es iterativo e incremental, se va a hacer testing de sistema en todas las etapas.

El testing tiene que hacerse dentro de lo que se denomina “ambiente de testing” que tiene que estar fuera de lo que es el ambiente de desarrollo. El ambiente de producción es donde el cliente final está trabajando y la teoría dice que debe haber un ambiente de preproducción que tiene que ser “igual” al de producción para que puedan realizarse las pruebas sin que impacten de manera directa sobre el ambiente donde se encuentra trabajando el usuario final y de esta manera reducir al mínimo el riesgo de incidentes debidos al ambiente específicamente.

**Testing de Aceptación de Usuario**

Es la prueba realizada por el usuario para determinar si la aplicación se ajusta a sus necesidades.

El objetivo de las pruebas de aceptación es el de establecer confianza en el sistema, las partes del sistema o las características específicas y no funcionales del sistema.

Encontrar defectos no es el foco principal de las pruebas de aceptación. Comprende tanto las pruebas realizadas por el usuario en ambiente de laboratorio (pruebas alfa), como las pruebas en el ambiente de trabajo reales (pruebas beta).

**Proceso de Pruebas**

El testing tiene que ser repetible y debe tener un proceso (testing sistemático), ya que si no se puede repetir no sirve porque “un error que no se puede repetir no es un error”.

**Etapas:**

* **Planificación y Control:** es la actividad de verificar que se entienden las metas y los objetivos del cliente, las partes interesadas, el proyecto, y los riesgos de las pruebas que se pretende abordar. Para la construcción del Plan de Testing se tienen que tener en cuenta: riesgos y objetivos del testing, estrategia de testing, recursos y el criterio de aceptación.
* **Identificación y Especificación:** en esta etapa se determina cómo vamos a probar y que es lo que se va a probar.
* **Ejecución:** se desarrollan y se da prioridad a los casos de prueba, se crean los datos de prueba, se implementa y verifica el ambiente, se ejecutan los casos de prueba, se registra el resultado de la ejecución de las pruebas y se comparan los resultados reales con los esperados.
* **Evaluación y Reporte:** se evalúan los criterios de aceptación y se reportan los resultados de las pruebas, se verifican los entregables y que los defectos hayan sido corregidos y se hace una evaluación de cómo resultaron las actividades de testing y se analizan las lecciones aprendidas.

**Caso de Prueba**

Es un conjunto de instrucciones o condiciones que el tester diseña y bajo las cuales determinará si el software está funcionando correctamente o no.

Los datos necesarios para realizar los casos de pruebas se pueden obtener desde documentos del cliente, desde información de relevamiento, desde requerimientos, desde especificaciones de programación y desde el código, pero lo mejor es sacarlos desde los requerimientos para garantizar que cumplan con lo que el sistema tiene que hacer.

Son más específicos que los casos de uso porque toman un escenario en particular de un caso de uso y le da valores específicos para ese escenario.

**Condiciones de prueba**

Esta es la reacción esperada de un sistema frente a un estímulo en particular, este estímulo está constituido por las distintas entradas.

Una condición de prueba debe ser probada por al menos un caso de prueba.

**Estrategias**

Existen dos estrategias de testing:

* **Estrategia de Caja Negra:** es un testing dinámico, porque yo ejecuto el sistema, en el cual sólo se analizan las entradas y los resultados, no hay información sobre la implementación. Debe ser complementado con el de caja blanca. El foco es definir muy bien lo que se quiere probar con condiciones de entradas y de entorno bien fijadas y ver si se obtiene el resultado esperado o no. El proceso es trasparente, es decir no se ve el proceso que el sistema hace para poder trasformar las entradas en salidas, se ve directamente el resultado y con el diseño de los casos de pruebas se puede ver si se obtuvo lo que se esperaba. Las técnicas de caja negra no son sistematizadas, son empíricas porque tienen que ver con la experticia de quién diseña los casos de prueba.

Técnicas de caja negra:

* Basado en especificaciones:
* Partición de equivalencia: tiene como objetivo minimizar los casos de prueba. Busca identificar dentro de los casos de entrada diferentes subconjuntos que produzcan resultados equivalentes.
* Análisis de los valores límites: una vez que tengo las clases de equivalencia pruebo los valores límites de esa clase.
* Basados en la experiencia: tiene que ver con la experticia de quien realiza las pruebas y dentro de esta técnica encontramos la adivinanza de defectos y el testing exploratorio.
* **Estrategia de Caja Blanca:** es estático porque se mira por dentro como está hecho el código, en este caso si se tienen detalles sobre la implementación, por ejemplo, que no tenga código malicioso, que no tenga ciclos infinitos, etc.
* Cobertura de enunciados o caminos básicos: Permite tener una noción de cuan complejo es el código. Busca poder identificar cuáles son los caminos básicos o independientes para una determinada porción de código. La complejidad se mide en el número de caminos independientes, es decir el valor mínimo de casos de prueba para poder cubrir todos los caminos independientes del código porque de esta manera cubro todos los caminos independientes al menos una vez.
* Cobertura de Sentencias: consiste en recorrer cada sentencia de la porción de código al menos una vez.
* Cobertura de Decisión: busca valuar las decisiones por verdadero y por falso al menos una vez.
* Cobertura de Condición: valuar cada condición adentro de su decisión al por verdadero y por falso al menos una vez independientemente del resultado de las decisión.
* Cobertura de decisión/condición: en este caso si nos interesa el resultado de la decisión. Es una mezcla de las dos anteriores. Evalúa todas las condiciones por verdadero y por falso y todas las decisiones por verdadero y por falso paralelamente al menos una vez.
* Cobertura Múltiple: combina todas las valuaciones posibles de la tabla de verdad para las decisiones de todas las condiciones.

Cada estrategia es fuerte en algún aspecto que la otra no lo es y por eso es necesario que se complementen.

**Ciclo de testing**

Un ciclo de pruebas es la ejecución de la totalidad de casos de prueba establecidos aplicados a una misma versión del sistema a probar.

* **Regresión:**  una vez concluido el ciclo de prueba y corregidos los defectos que fueron encontrados, se vuelve a aplicar sobre la nueva versión todo el ciclo de prueba de nuevo para asegurarse de que al corregir los defectos no se introdujeron otros, teniendo en cuenta que “cada error que se corrige produce tres errores nuevos”.
* **Sin Regresión:** una vez que se terminan de corregir los defectos que se encontraron en el ciclo de prueba, únicamente se prueba sólo lo que se corrigió sin aplicar todo el ciclo de prueba nuevamente.

**Tipos de Pruebas**

Estos tipos de prueban que se pueden realizar en el software están asociadas a los distintos niveles de testing, ya que hay algunos de estos tipos de pruebas se pueden llevar a cabo mejor en un determinado nivel de prueba que en otros.

**Smoke Test:** es una corrida genérica que se le hace al sistema para ver si esta versión que va a entrar a testing está en condiciones de ser testeada. Vendría a ser un nivel cero para evitar que se produzcan situaciones como que al momento de probar hay que iniciar sesión y resulta que no existe un botón aceptar, por ejemplo.  
Se llama de esta manera porque se refiere a ensayos físicos realizados con humo en sistema cerrados de tuberías para detectar grietas/roturas.

**Testing Funcional:** cuando se planifica el testing también se definen los tipos de pruebas y las técnicas que se van a aplicar. Al momento de realizar la planificación hay que tener en cuenta que se deben probar tanto los requerimientos no funcionales como los funcionales.

Las pruebas funcionales se basan en funciones y características y su interoperabilidad con sistemas específicos.

* **Basadas en los requerimientos**: tomo un requerimiento y pruebo que esté bien implantado en el sistema de manera individual
* Basadas en procesos de **negocio**: hace un circuito del negocio completo vinculado a un proceso de negocio y a partir de eso arma un circuito de prueba.

**Testing no Funcional:** es la prueba que se lleva cabo para ver cómo funciona el sistema y se deben realizar porque los requerimientos no funcionales son tan importantes como los funcionales.

* **Performance de testing**: se evalúa el nivel de respuesta de un sistema, se mide fundamentalmente por tiempo de respuesta, pero no es el único parámetro que se utiliza para medirla.
* **Prueba de carga:** se prueba la capacidad máxima comprometida de hardware y de comunicaciones en el sistema. Si nosotros le dijimos al usuario que puede haber cien usuarios concurrentes hay que ver si es verdad que con cien usuarios concurrentes el sistema anda o se cae.
* **Pruebas de Stress:** es un paso más allá que la prueba de carga, por ejemplo, que pasa si yo dije 50 usuarios pero entran 100, no se espera que el sistema lo soporte, si no que se está esperando que el sistema se caiga para ver cómo queda ese sistema después, si es recuperable o no y cuánto tiempo se necesita para volver a dejar el sistema en condiciones. Esta prueba se hace fundamentalmente para que no suceda cuando el sistema ya se entregó al cliente, porque una vez entregado ya no se tiene ningún control sobre él.
* **Pruebas de Usabilidad:** la usabilidad es el vínculo que se establece entre los usuarios y el software. No se puede medir directamente y por esta razón es que existen varias formas de probarla.
* **Pruebas de Mantenimiento:** tiene que ver con el esfuerzo requerido para introducir cambios en el software, es decir, cuánto cuesta introducir esos cambios.
* **Pruebas de fiabilidad:** que tanto se puede depender del software, que tanto se puede confiar en el sistema y al igual que la usabilidad no se puede medir directamente, se mide por ejemplo en la cantidad de caídas del sistema, accesos indebidos, fraudes, etc.
* **Pruebas de portabilidad:** capacidad de mover el software a distintitos dispositivos de hardware.

**Testing de interfaces de usuarios:** hay que elegir el conjunto de casos de pruebas que permitan la mayor cobertura y menor costo.

**Testing de performance:** el tiempo de respuesta es algo muy dependiente de los recursos y de la concurrencia, depende en muchas cosas de hardware, del software, de la conectividad, del ancho de banda, entre otras y estas cosas el cliente no las ve y le echa la culpa al sistema nomás.

**Testing de Configuración:** tiene que ver con la instalación y configuración inicial del software para ver que funciona todo de manera integrada con otros software o software de terceros, etc.

**El Testing y el Ciclo de Vida**

¿Cuándo se hace testing? ¿Cómo metemos el testing dentro del ciclo de vida?

* Procesos empíricos: dentro de estos procesos encontramos las metodologías ágiles
* Procesos definidos: dentro de estos procesos podemos encontrar por ejemplo al PUD entre otros

El propósito del testing es apuntar hacia la “V y V”, verificación y validación, ¿estamos construyendo el sistema correctamente?, ¿estamos construyendo el sistema correcto? Respectivamente.

Hay que decidir en qué momento del ciclo de vida se va a comenzar con el testing. Existe un acuerdo entre los dos procesos mencionados anteriormente que dice que hay que adelantar lo más que se pueda el testing, mientras más temprano mejor, porque un requerimiento que no se puede probar tampoco se va a poder construir, la diferencia entre estos procesos es la propuesta que tienen para llevarlo a cabo.

**Modelo en V**

Promueve que se puede hacer gran parte del trabajo anticipadamente antes de tener una línea de código.

Plantea que se desarrolla de lo general a lo particular, es decir que se va probando desde lo más chico a lo más grande. Cuándo ya se tienen definidos los requerimientos entonces ya se pueden ir probando, cuándo se terminó el diseño ya se pueden ir realizando las pruebas unitarias por ejemplo, etc.

**TDD (Desarrollo conducido por testing)**

Es introducido por las metodologías ágiles y es una **metodología de desarrollo,** no de testing, que intenta que funcionen la programación y la prueba de unidad, que se fusionen.

Es una metodología de desarrollo porque es una decisión de cómo construir código, se construyen primero las pruebas unitarias, es decir se programa el componente que se va a utilizar para probar el componente que se va a construir y después de ejecutan las pruebas, que en general son automatizadas, y se va mejorando el código a través de la refactorización.

Clean code: código que sea simple de entender y que haga lo que tienen que hacer con la menor cantidad de líneas posibles.

**Manifiesto de Testing**

Se prefiere testear a lo largo de todo el proceso que testear al final y se prefiere trabajar con la prevención de los defectos razón por la cual trabajan mucho con revisiones técnicas porque permite detectar rápidamente los errores. Recomiendan prácticas alternativas para llevar adelante el desarrollo del software.

Vamos a trabajar en la comprensión del software, del conocimiento del negocio y para que el negocio quiere el software más que tener un chek para ver donde está lo que tenés mal.

Vamos a tratar de trabajar para que el sistema que le entreguemos al cliente sea el mejor sistema que se puede tener más que trabajar para ver como se lo puede romper.

La calidad es responsabilidad de todos y cada uno de los miembros del equipo que participan en la construcción del software y no solo responsabilidad del tester.

Principios para el testing en proyectos ágiles:

* El testing se mueve hacia adelante en el proyecto.
* El testing no es solo una fase.
* Todos hacen testing.
* Reducir la latencia del feedback.
* Las respuestas representan expectativas.
* Mantener el código limpio, corregir los defectos rápido.
* Reducir la sobrecarga de documentación de las pruebas.
* Las pruebas son parte del “done”.
* De probar al final a conducido por pruebas.

Hay que tratar de automatizar las pruebas lo más que se pueda.

Lo que cambia en las metodologías ágiles es el cómo hacer testing, pero no el que, es decir que el testing se tiene que hacer.

**Revisiones Técnicas**

**Verificación y Validación**

Es un proceso de ciclo de vida completo que inicia con las revisiones de los requerimientos y continúa con las revisiones de diseño, inspecciones del código hasta la prueba.

Debe planificarse porque es un proceso caro, el costo más representativo es el esfuerzo el cual representa un 80%, y debe comenzar en etapas tempranas.

La validación responde a la pregunta ¿estamos construyendo el producto correcto? Mientras que la verificación responde a ¿estamos construyendo el producto correctamente?

**Falla:** error en un producto de trabajo. Estas fallas pueden ser mayores (condición que puede causar una falla operacional o producir un resultado inesperado durante la ejecución de la operación especificada), menores (condición que no es deseable en el producto pero que no puede ocasionar una falla operacional) y cosmético (error de tipeo, ortografía, errores menores que no se cuentan como fallas).

**Producto de trabajo:** salida de cualquier actividad correspondiente al ciclo de vida de desarrollo.

El propósito de la verificación es la detección temprana de defectos antes de que se conviertan en fallas del sistema, lo que provoca que el sistema no haga lo que tiene que hacer, y esto se ocasiona por el “teléfono descompuesto”.

**Principios:**

* La prevención es la mejor cura, sobretodo porque es más barato.
* Evitar es más efectivo que eliminar.
* La retroalimentación enseña efectivamente.
* Prioriza lo rentable.
* Olvidarse de la perfección ya que la misma no se puede conseguir entonces hay que llegar a un acuerdo sobre cuando se considera bueno.

**Revisiones Técnicas**

Es un proceso de V & V estático que tiene como principal objetivo detectar los defectos y corregirlos en las etapas tempranas del desarrollo.

Puede inspeccionarse cualquier representación legible del software.

Se aplican en varios momentos del desarrollo.

Algunas clases de errores se le pasan por alto más fácilmente al que los origina que a otras personas.

Motiva a realizar un mejor trabajo y no requieren que el programa se ejecute.

**Ventajas:**

* Pueden descubrirse muchos errores.
* Pueden inspeccionarse versiones incompletas.
* Pueden considerarse otros atributos de calidad.

**Desventajas**

* Es muy difícil introducir las inspecciones formales.
* Sobrecargan al inicio los costos y conducen a un ahorro sólo después que los equipos adquieran experiencia en su uso.
* Requieren tiempo para organizarse y parecen ralentizar el proceso de desarrollo.

Dos tipos de revisiones técnicas son:

* **Walkthroughs:** Tiene buenos resultados pero se toman pocas métricas. No hay control del proceso, por lo que se considera informal.
  + Objetivos típicos:
    - Mínima sobrecarga
    - Capacitación de desarrolladores
    - Rápido retorno
* **Inspecciones:** Mejores resultados, proceso controlado, roles definidos, lo que lo convierte en un técnica formal. Métricas útiles a lo largo de todo el ciclo de vida del desarrollo.
  + Objetivos típicos
    - Detectar y remover todos los defectos eficiente y efectivamente.

**Métricas sugeridas**

1. Densidad de defectos
2. Total de defectos encontrados
3. Esfuerzo de la inspección
4. Esfuerzo por defecto
5. Porcentaje de re inspecciones
6. Defectos corregidos sobre total de defectos

**Inspección**

Es una actividad de garantía de calidad de software.

Se lleva a cabo mediante una reunión y el éxito depende de su planificación.

Objetivos

1. Descubrir errores
2. Verificar que el software alcanza sus requisitos
3. Garantizar que el software ha sido representado de acuerdo a ciertos estándares
4. Conseguir un software desarrollado de manera uniforme
5. Hacer que los proyectos sean más manejables

Son una forma efectiva y barata de **encontrar** fallas, proveer **métricas** al proyecto, promover el trabajo en grupo y mejorar la **calidad** del producto.

Tiene una duración máxima de dos horas.

**Roles**

1. Autor
   1. Creador/encargado de mantener el producto que va a ser inspeccionado.
   2. Inicia el proceso asignado un moderador y designa junto al moderador el resto de los roles.
   3. Entrega el producto a ser inspeccionado al moderador.
   4. Reporta el tiempo de retrabajo y los defectos al moderador
2. Moderador
   1. Planifica y lidera la revisión.
   2. Con el autor selecciona los roles.
   3. Entrega el producto a inspeccionar a los inspectores antes de la reunión.
   4. Coordina la reunión.
   5. Hace el seguimiento de los defectos reportados.
3. Anotador: Registra los hallazgos de la revisión.
4. Lector: Lee el producto a ser inspeccionado.
5. Inspector: Examina el producto antes de la reunión para encontrar defectos.

**Proceso de inspección**

1. Planificación: Elegir equipo, preparar material y calendario.
2. Visión general: Presentar proceso y producto.
3. Preparación: Análisis individual para encontrar potenciales defectos.
4. Reunión de Inspectores: Análisis del equipo para recolectar potenciales defectos previos, filtrar falsos positivos. Se decide si rechazar, aceptar sin modificaciones o aceptar provisoriamente el producto.
5. Corrección
6. Seguimiento: Verificar correcciones, recolectar datos.

**Walkthrough/Recorrida**

Técnica de **análisis estático** en la que un diseñador/programador dirige miembros del equipo de desarrollo y otros a través de un producto de software y los participantes formulan preguntas y realizan comentarios acerca de posibles errores, violación de estándares, etc.

**Proceso de Software**

Conjunto estructurado de actividades para desarrollar un sistema de software. Estas actividades varían dependiendo de la organización y el tipo de sistema que debe desarrollarse.

Es un conjunto de actividades, métodos, prácticas, y transformaciones que la gente usa para desarrollar o mantener software y sus productos asociados (Sw-CMM).

**Proceso definido**

Asume que podemos repetir el mismo proceso una y otra vez, indefinidamente, y obtener los mismos resultados. La administración y el control provienen de la predictibilidad del proceso definido.

**Proceso empírico**

Asume procesos complicados con variables cambiantes. Cuando se repite el proceso, se pueden llegar a obtener resultados diferentes. La administración y control se realiza través de inspecciones frecuentes y adaptaciones.

**Ciclos de vida**

La serie de pasos a través de los cuales el producto/proyecto progresa.

Un ciclo de vida de un proyecto software es un representación de un proceso. Gráfica una

Descripción del proceso desde una perspectiva particular. También se conocen como modelo de proceso.

Sirven para proveer una guía para la administración de proyectos.

Los modelos especifican:

* Las fases del proceso
* El orden en el cual se llevan a cabo

Hay tres tipos básicos de ciclos de vida:

* **Secuencial**: Una actividad no inicia hasta que ha terminado la anterior. El desarrollo secuencial ha fallado en muchos proyectos grandes y complejos llevando a importantes demoras y sobrecostos. Ejemplo Cascada, Ventajas: bueno para proyectos rígidos y con buena especificación de requerimientos. También funciona bien cuando el equipo de trabajo no tiene mucha experiencia ya que provee una estructura rígida. Desventajas: sin resultados o mejoras visibles y rara vez el cliente va a establecer al principio todos los requerimientos. Es difícil especificar enteramente los requerimientos al principio del proyecto.
* **.Iterativo/Incremental**: Ejemplo Iterativo, permite mejorar y ajustar el proceso. Desarrollo lento por la necesidad de la retroalimentación del cliente.
* **Recursivo**: Ejemplo espiral. Ventajas: Si el proyecto no puede realizarse debido a razones técnicas, esto se detectara rápido y no producirá mucho costo.

Para elegir el mejor ciclo de vida para nuestro proyecto debemos tener en cuenta:

* A qué nivel el cliente y el equipo de trabajo comprenden los requerimientos al principio del proyecto, y si esa comprensión cambiará a lo largo del proyecto.
* A qué nivel el equipo entiende la arquitectura del sistema. Vamos a necesitar cambiar radicalmente esta arquitectura a lo largo del proyecto?
* Cuanta confiabilidad necesito? Cuanto tengo que planear en avanzado durante el proyecto para versiones futuras?

**Ingeniería de Software**

**Ingeniería de sw, métodos y técnicas aplicadas**

**La ingeniería de sw es una construcción multipersona de multiversiones de sw.**

**Es una disciplina de ingeniería que se interesa por todos los aspectos de la producción de sw, desde las primeras etapas de la especificación del sistema hasta el mantenimiento del sistema después que se pone en operación.**

No solo se interesa por los procesos técnicos del desarrollo de sw, sino también incluye actividades como la administración del proyecto de sw y el desarrollo de herramientas, así como métodos y teorías para apoyar la producción de sw.

La ingeniería de sw busca apoyar el desarrollo de sw profesional, en lugar de la programación individual. Incluye técnicas que apoyan la especificación, el diseño y la evolución del programa. Cuando se habla de ingeniería de sw, no solo se refiere a los programas en sí, sino también a la documentación asociada y a los datos de configuración requeridos para hacer que estos programas operen de manera correcta.

Los **principales retos de la ISW** son: diversidad creciente, demandas por tiempos de distribución limitados y desarrollo de sw confiable.

Aproximadamente 60% de los costos de la ingeniería de sw son de desarrollo y el 40% de prueba.

El enfoque sistemático que se usa en la ing. de sw se conoce en ocasiones como proceso de sw. Un proceso de sw es una secuencia de actividades que conducen a la elaboración de un producto de sw. Existen cuatro actividades fundamentales que son comunes a todos los procesos de software y éstas son:

-*Especificación del sw:* Definir el sw que se producirá y las restricciones en su operación.

-*Desarrollo del sw:* Diseñar y programar el sw.

-*Validación del sw:* Se verifica el sw para asegurar que sea lo que el cliente requiere.

-*Evolución del sw*: Se modifica el sw para reflejar los requerimientos cambiantes del cliente y del mercado.

Diferentes tipos de sistemas necesitan diferentes procesos de desarrollo.

**La ing de sw es un enfoque sistemático para la producción de sw que toma en cuenta los costos, fecha, confiabilidad, así como las necesidades de clientes y fabricantes de sw.**

**Software**

Es un set de programas, archivos de configuración y la documentación asociada

Tipos básicos de Software:

* **System Software** (S.O)
* **Utilitarios** (Winrar)
* **Software de Aplicación** (Office)

Tipos de Productos:

* **Productos Genéricos**: sistemas producidos por una organización de desarrollo y se venden al mercado abierto a cualquier cliente. La especificación es controlada por quien lo desarrolla. Ejemplo: sw para PC como BD, Procesadores de texto, etc.
* **Personalizados o A Medida**: son sistemas requeridos por un cliente en particular, la especificación es desarrollada y controlada por la organización que compra el software. Ejemplo: sistema escrito para apoyar cierto proceso empresarial.

Razones para no comparar software y manufactura

El software es menos predecible

No hay producción en masa, casi ningún producto de software es igual a otro.

No todas las fallas son errores

El software no se gasta

El software no está gobernado por las leyes de la física

Algunos problemas con el desarrollo de software

La versión final del producto no satisface las necesidades del cliente.

No es fácil extenderlo y/o adaptarlo. Agregar más funcional en otra versión es casi una misión imposible

Mala documentación

Mala calidad.

Más tiempos y costos que los presupuestados

*Los principales problemas son no cumplir con las expectativas del cliente (requerimientos) y las estimaciones (no dimensionar correctamente el esfuerzo/tiempo) para darle al cliente el SW.*

**Disciplinas Técnicas**

1. Requerimientos
2. Análisis y diseño
3. Prueba
4. Despliegue

**Disciplinas de Gestión**

1. Planificación de proyecto
2. Monitoreo y control de proyectos

**Disciplinas de Soporte**

1. Gestión de configuración de Software
2. Aseguramiento de calidad
3. Métricas