

Unidad 1: Ingeniería de Software en Contexto

Introduction a la Ingeniería del Software. Que es?

Software: Conjunto de programas y documentación asociada. Existen 3 tipos, software system , software de aplicación y utilitarios. El buen software debe entregar al usuario la funcionalidad y desempeño requeridos, sustentable, confiable y utilizable.

Problemas comunes al desarrollar:

- | | |
|--------------------------|--|
| -No satisface al cliente | -Mayores costos y tiempos que los presupuestados |
| -Mala calidad | -Mala documentación |

Ingeniería de Software: Es una disciplina de la ingeniería que se interesa por todos los aspectos de la producción de software (procesos técnicos y las actividades como administración del proyecto de software y el desarrollo de herramientas).

Estado actual y antecedentes. La Crisis del Software

The Mythical Man Month

La mayoría de los proyectos de software se han torcidos por la falta de tiempo de calendario más que cualquiera de las otras causas combinadas. Porque es esta una causa del desastre tan comun?

Primer, nuestras técnicas de estimación están pobremente desarrolladas.

Segundo, nuestras técnicas de estimación confunden el esfuerzo con el progreso, escondiendo la suposición que los hombres y los meses son intercambiables.

Tercero, el progreso del calendario es monitoreado pobremente.

Cuarto, cuando un desfasaje del calendario es reconocido, la natural y tradicional respuesta es agregar mas fuerza humana. Como arrojando gasolina a un fuego, esto complica aun mas las cosas, mucho mas. Mas fuego requiere más gasolina, y así comienza un ciclo regenerativo que termina en desastre.

El segundo fantasioso modo de pensar es expresado en la mismísima unidad de esfuerzo utilizada para estimar y calendarizar: the hombre-mes. El costo sin embargo varia según la cantidad de hombres y de meses. Pero el progreso no. Implica que los meses y los hombres son intercambiables.

Los hombres y los meses son comodities intercambiables solo cuando una tarea puede ser participada entre muchos trabajadores sin comunicación entre ellos.

El problema agregado de la comunicación esta hecho de dos partes, el entrenamiento y la intercomunicacion. Cada trabajador debe estar entrenado en la tecnologia, los objetivos del esfuerzo, la estrategia global, y el plan de trabajo. Este entrenamiento no puede ser particionado , así como el esfuerzo agregado varia linealmente con el numero de trabajadores.

La intercomunicación es peor. Si cada parte de la tarea debe ser coordinada separadamente con cada otra parte el esfuerzo aumenta exponencialmente.

Ingeniería de Software un Enfoque Practico (Capítulo 1):

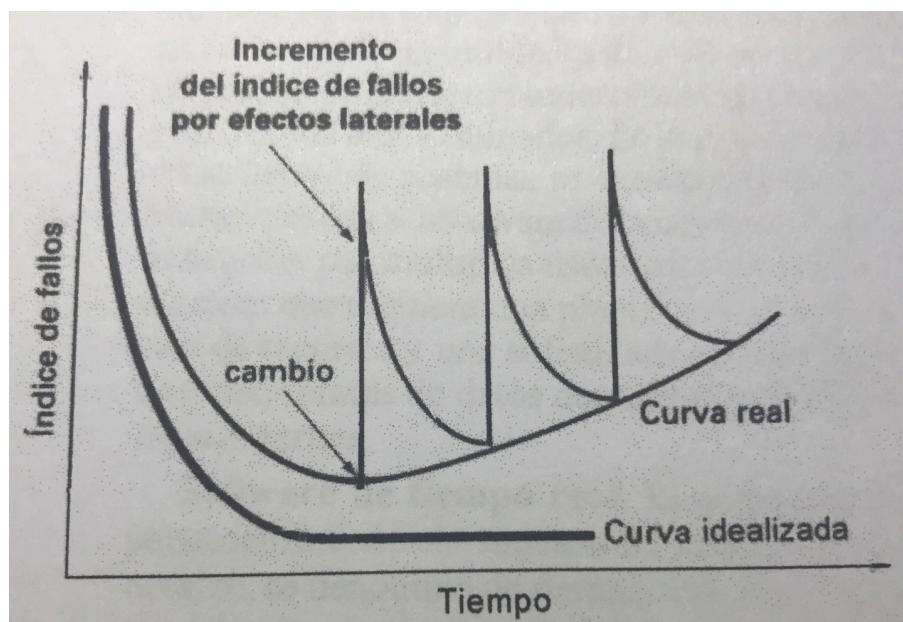
Características del Software:

-El software se desarrolla no se fabrica en un sentido clásico: Los costes del software se encuentran en la ingeniería. Eso significa que los proyectos de software no se pueden gestionar como si fueran proyectos de planificación

Durante su vida, el software sufre cambios(mantenimiento).Conforme se hacen los cambios, es bastante probable que se introduzcan nuevos defectos haciendo que la curva de fallos tenga picos. Lentamente el nivel mínimo de fallos comienza a crecer - el software se va deteriorando debido a los cambios.

Cuando un componente de hardware se estropea se sustituye por una pieza de repuesto. No hay piezas de repuesto para el software. Cada fallo en el software indica un error en el diseño o en el proceso mediante el que se tradujo el diseño a código de maquina. Por tanto, el mantenimiento del software tiene una complejidad considerablemente mayor que la del mantenimiento del hardware.

Aunque la industrie tiende a ensamblar componentes, la mayoría del software se construye a medida.



Crisis del Software:

El mal abarca los problemas asociados a cómo desarrollar software, como mantener el volumen cada vez mayor de software existente y como poder esperar mantenernos al corriente de la demanda creciente de software.

Mitos y Realidades del Software

Mito: Si fallamos en la planificación, podemos añadir mas programadores y adelantar el tiempo perdido.

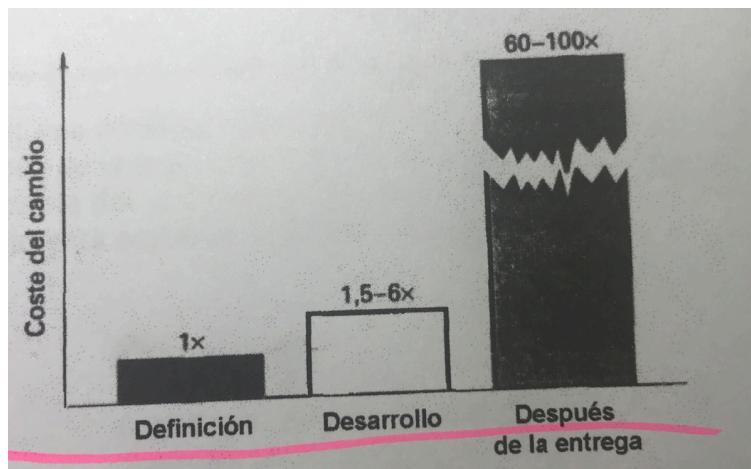
Realidad: El desarrollo de software no es un proceso mecánico como al fabricacion. "añadir gente a un proyecto de software retrasado lo retrasa mas". Puede añadirse gente pero solo de una manera planificada y bien coordinada.

Mito: Los requisitos del proyecto cambian continuamente, pero los cambios pueden acomodarse fácilmente ya que el software es flexible.

Realidad: Es verdad que los requisitos del software cambian, pero el impacto del cambio varia según el momento en el que se introduzcan.

Mito: Una vez que escribimos el programa y hacemos que funcione, nuestro trabajo ha terminado.

Realidad: "Cuanto mas pronto se comience a escribir código mas se tardara en terminarlo". Los datos industriales indican que entre el 60 y el 80 por ciento de todo el esfuerzo dedicado a un programa se realizara después de que se le haya entregado al cliente por primera vez.



Mito: Lo único que se entrega al terminar el proyecto es el programa funcionando.

Realidad: Un programa que funciona es solo una parte de una configuración del software que incluye muchos elementos. La documentación proporciona el fundamento para un buen desarrollo, y lo que es más importante, proporciona guías para la tarea de mantenimiento del software.

Disciplinas que conforman la ingeniería de Software

Disciplinas:

Técnicas:

- Requerimientos
- Análisis y diseño
- Prueba
- Construcción
- Despliegue

De Gestión:

- Planificación de proyecto
- Monitoreo y control de proyectos

De Soporte:

- Gestión de configuración de SW (SCM)
- Aseguramiento de calidad
- Métricas

Los atributos esenciales de los productos de software son: mantenimiento, confiabilidad, seguridad, eficiencia y aceptabilidad.

Procesos de Desarrollo Empíricos vs Definidos

Proceso de Software: Conjunto estructurado de actividades para desarrollar un sistema de software. Estas actividades varían de acuerdo a la organización y el tipo de software a desarrollar. Aún así, existen cuatro **actividades fundamentales** que son comunes a todos los procesos de software:

- 1) **Especificación:** definir la funcionalidad del sw y las restricciones de operación.

- 2)Desarrollo:** se debe desarrollar el sw para cumplir con las especificaciones.
- 3)Validación:** hay que validar que el software para asegurarse que cumple con lo que el cliente quiere.
- 4)Evolución:** el sw debe evolucionar para satisfacer las necesidades cambiantes del cliente

Tipos de Procesos:

- Definidos/plan driven:** Son aquellos donde todas las actividades del proceso se planean por anticipado y el avance se mide contra dicho plan.
 - Asume que podemos repetir el mismo proceso indefinidamente y obtener los mismos resultados.
 - La administración y control provienen de la predictibilidad del proceso.
- Empíricos/Ágiles:** La planeación es incremental y más fácil de modificar el proceso para reflejar los requerimientos cambiantes del cliente.
 - Asume procesos complicados con variables cambiantes.
 - Se obtienen resultados diferentes al repetir el proceso.
 - La administración y control es a través de inspecciones frecuentes y adaptaciones.
- Patrón de conocimiento en procesos empíricos:** ..->Asumir->Construir->Retroalimentar->Revisar->Adaptar->..

!Para que un proceso sea administrado tiene que ser modelado ---> Ciclos de Vida

Ciclos de vida (Modelos de Proceso) y su influencia en la administración de proyectos de Software

Ventajas y desventajas de C/U de los ciclos de vida. Criterios para la elección de ciclos de vida en función de las necesidades del proyecto y las características del producto

Ciclo de Vida: Es la representación simplificada de un proceso. Gráfica la descripción del proceso desde una perspectiva particular. Los modelos especifican las fases(actividades) del proceso y el orden en el cual se llevan a cabo.

Clasificación:

Secuencial: Este toma las actividades fundamentales del proceso y las representa como fases separadas, tales como especificación de requerimientos, diseño , implementación, et

Características:

- Cada fase produce documentación, esto hace que el modelo sea visible.
- Deben establecerse compromisos en una etapa temprana, lo que dificulta responder a los requerimientos cambiantes del cliente.
- Debe usarse cuando los requerimientos se entiendan bien y sea improbable el cambio radical durante el desarrollo del sistema.

Iterativo/Incremental: Este enfoque vincula las actividades de especificación, desarrollo y validación. El sistema se desarrolla como una serie de versiones(incrementos) y cada versión añade funcionalidad a la versión anterior.

Características:

-Se basa en la idea de diseñar una implementación inicial, exponer esta al comentario del usuario y luego desarrollar sus distintas versiones hasta producir el sistema adecuado.

-Resulta más barato y fácil realizar cambios en el software a medida que se diseña.

-Cada incremento incorpora función que necesita el cliente.

Beneficios:

- 1) Se reduce el costo de adaptar los requerimientos cambiantes del cliente. La cantidad de análisis y documentación que debe reelaborarse es mucho menor.
- 2) Es más sencillo obtener retroalimentación del cliente sobre el trabajo que se realiza. Los clientes encuentran difícil juzgar el avance a partir de documentos de diseño.
- 3) Mas rápida la entrega e implementación de SW útil al cliente. Los clientes tienen la posibilidad de usar y ganar valor del software más temprano.

Este enfoque puede estar basado en un plan, ser ágil o mas, usualmente, una mezcla de dichos enfoques. En un enfoque basado en un plan se identifican por adelantado los incrementos de sistema, si se adopta un enfoque agil. Se detectan los primeros incrementos, aunque el desarrollo de incrementos posteriores depende del avance y las prioridades del cliente.

Desde un perspectiva administrativa el enfoque incremental tiene dos problemas:

Problemas:

- 1) El proceso no es visible. Los administradores necesitan entregas regulares para medir el avance.
- 2) La estructura del sistema tiende a degradarse conforme se tienen nuevos incrementos. La incorporación de más cambios de SW se vuelve cada vez más difícil y costosa.

!!Los problemas del desarrollo incremental se tornan agudos para sistemas grandes, complejos y de larga duración donde diversos equipos desarrollan partes diferentes del sistema. Los grandes sistemas necesitan de un marco o una arquitectura estable y es necesario definir con claridad, respecto a dicha arquitectura, las responsabilidades de los distintos equipos que trabajan en partes del sistema. Esto debe planearse por adelantado en vez de desarrollarse de manera incremental.

Recursivo(Solo es nombrada):

Perspectiva sobre ciclos de vida en libro Desarrollo y gestión de proyectos informáticos

La principal función del ciclo de vida es establecer el orden en el cual un proyecto especifica, prototipa, diseña , implementa, revisa , prueba y realiza sus otras actividades. Establece el criterio que utilizas para determinar si proceder de una tarea a otra.

El modelo de vida que es elegido tiene mucha influencia sobre el éxito del proyecto.

Dependiendo en el ciclo que se elige, se puede mejorar:

- La velocidad de desarrollo
- el control y monitoreo del proyecto
- la relación con clientes

Y minimizar la exposición a riesgos.

Secuencial/Modelo en cascada puro: El proyecto progresá a través de una secuencia ordenada de pasos desde el concepto de software inicial hasta la prueba del mismo. El proyecto mantiene una revisión al final de cada fase para determinar si está listo para avanzar a la siguiente fase. El modelo es manejado por documentos, es decir que los productos del trabajo que son llevados de fase en fase son documentos.

Ayuda a minimizar el planeamiento más adelante, porque ya se puede realizar todo en el principio. No provee resultados tangibles en la forma de software hasta el final del proceso, pero cuando alguien es familiar con el ,la documentación generada le provee indicaciones importantes durante el ciclo de vida.

Cuando utilizar secuencial:

- Cuando se tiene una definición estable del mismo y cuando se está trabajando con metodologías técnicas bien entendidas.Este modelo ayuda a encontrar los errores en etapas tempranas de bajo costo.
- Cuando se planea el mantenimiento de un producto o exportar uno a una nueva plataforma.

Desventajas:

- Se debe especificar los requerimientos al comienzo del proyecto, antes que el trabajo de diseño se haya hecho o algo de código haya sido escrito, lo que puede llevar mucho trabajo.
- Puede llevar una excesiva cantidad de documentación.

-El modelo secuencia genera pocos signos visibles de progreso hasta casi el final.

Lo que puede crear una falsa percepción de desarrollo lento. A los clientes les gusta tener afirmaciones tangibles que su proyecto va a ser entregado en tiempo.

Componentes de un Proyecto de Sistemas de Información

Proyecto de Sistema de Información:

Proyecto: Un proyecto es una planificación que consiste en un conjunto de actividades que se encuentran interrelacionadas y coordinadas (Definición del PMI).

Características:

- Están **orientados a objetivos**.
- Tienen una **duración limitada**, tienen principio y fin.
- Implican **tareas relacionadas basadas en esfuerzos y recursos**.
- Cada proyecto **es único**.

Orientación a objetivos:

- Los proyectos están orientados a obtener resultados y esto se refleja a través de objetivos.
- Los objetivos guían al proyecto.
- Los objetivos no deben ser ambiguos.
- Objetivos claros y alcanzables.

Duración limitada:

- Los proyectos son temporales, cuando se alcanzan los objetivos este se termina.

Tareas relacionadas basadas en esfuerzos y recursos:

- Complejidad sistémica de los problemas.

Los criterios del éxito varían de un proyecto a otro pero las metas importantes son:

- 1)Entregar el sw al cliente en el tiempo acordado.
- 2)Mantener costos dentro del presupuesto.
- 3)Entregar software que cumpla con las expectativas del cliente.
- 4)Mantener un equipo de desarrollo óptimo y con buen funcionamiento.

Administración de Proyectos: Es la aplicación de conocimientos, habilidades, herramientas y técnicas a las actividades del proyecto para satisfacer los requerimientos del mismo. Su objetivo es tener el trabajo hecho en tiempo, con el presupuesto acordado y habiendo satisfecho las especificaciones o requerimientos.

Incluye:

- Identificar los requerimientos.
- Establecer objetivos claros y alcanzables.

-Adaptar las especificaciones , planes y el enfoque a los diferentes intereses de los stakeholders.

La restricción triple: Triple Constraint

Alcance/Objetivos del Proyecto → Performance requerida

Tiempo → Fecha comprometida

Costo → Restricción de presupuesto

El balance de estos factores afecta directamente la calidad del proyecto. Es responsabilidad del líder de mantenerlos balanceados.

Equipo de Proyecto: Es un grupo de personas comprometidas en alcanzar un conjunto de objetivos de los cuales se sienten responsables. Cuentan con diversos conocimientos y habilidades, desarrollan sinergia, y suelen ser grupos pequeños.

Líder de proyecto: Líder del Proyecto: Para ser un líder uno debe sentirse cómodo con los cambios y entender la organización. El líder debe ubicar a la gente en el rol que mejor sepan cumplir y guiarla en el.

El líder debe tener los Hard Skills, que son los conocimientos del producto, herramientas y técnicas, y los Soft Skills, que es la capacidad de trabajar con gente y de los más difíciles de conseguir (comunicación, liderazgo, creatividad).

El líder debe definir el alcance del proyecto, estimar tiempos y requerimientos, identificar los recursos requeridos, identificar y evaluar riesgos, sus planes de contingencia, etc.

Plan de Proyecto: Es un documento utilizado para comunicar al equipo y los clientes cómo se realizará el trabajo. Este incluye las siguientes actividades:

- Definición de alcance
- Definición del proceso y el ciclo de vida.
- Estimación
- Gestión de riesgos
- Asignación de recursos
- Programación de proyectos
- Definición de controles
- Definición de métricas

Alcance del producto: Son todas las características que pueden incluirse en el producto o servicio. Se mide en contra de la especificación de requerimientos.

Alcance del Proyecto: Es todo el trabajo que debe hacerse para entregar el producto de con todas las características y funciones solicitadas. Se mide contra el plan de proyecto.

Riesgo: Problema que puede suceder y alterar el calendario del proyecto o la calidad de software.

Gestión de riesgo: Implica la identificación y valoración de los grandes riesgos del proyecto para establecer la probabilidad de que ocurren. También supone identificar y valorar las consecuencias para el proyecto si dicho riesgo surge. Deben hacerse planes para evitar, gestionar o enfrentar posibles riesgos.

La gestión del riesgo es una de las tareas más sustanciales para un administrador de proyecto. La gestión del riesgo implica anticipar riesgos que pudieran alterar el calendario del proyecto o la calidad del software a entregar, y posteriormente tomar acciones para evitar dichos riesgos.

Categorías de Riesgo:

-Riesgos del proyecto Los riesgos que alteran el calendario o los recursos del proyecto. Un ejemplo de riesgo de proyecto es la renuncia de un diseñador experimentado. Encontrar un diseñador de reemplazo con habilidades y experiencia adecuadas puede demorar mucho tiempo y, en consecuencia, el diseño del software tardará más tiempo en completarse.

-Riesgos del producto Los riesgos que afectan la calidad o el rendimiento del software a desarrollar. Un ejemplo de riesgo de producto es la falla que presenta un componente que se adquirió al no desempeñarse como se esperaba. Esto puede afectar el rendimiento global del sistema, de modo que es más lento de lo previsto.

-Riesgos empresariales Riesgos que afectan a la organización que desarrolla o adquiere el software. Por ejemplo, un competidor que introduce un nuevo producto es un riesgo empresarial. La introducción de un producto competitivo puede significar que las suposiciones hechas sobre las ventas de los productos de software existentes sean excesivamente optimistas.

Proceso de Gestión del Riesgo

1) Identificación del riesgo Hay que identificar posibles riesgos para el proyecto, el producto y la empresa.

2) Análisis de riesgos Se debe valorar la probabilidad y las consecuencias de dichos riesgos.

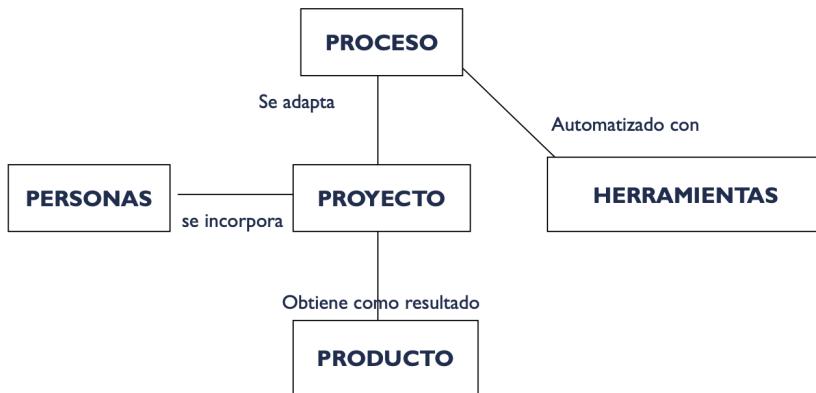
3) Planeación del riesgo Es indispensable elaborar planes para enfrentar el riesgo, evitarlo o minimizar sus efectos en el proyecto.

4) Monitorización del riesgo Hay que valorar regularmente el riesgo y los planes para atenuarlo, y revisarlos cuando se aprenda más sobre el riesgo.

Causas de los fracasos en los proyectos:

- Fallas al definir el problema
- Planificar basado en datos insuficientes
- La planificación la hizo el grupo de planificaciones
- No hay seguimiento del plan de proyecto
- Plan de proyecto pobre en detalles
- Planificación de recursos inadecuada

Vínculos proceso-proyecto-producto en la gestión de un proyecto de desarrollo de software



Gestion de Proyectos Cap 22/23 Somerville

La gestión de proyectos de software es una parte esencial de la ingeniería de software. Los proyectos necesitan administrarse porque la ingeniería de software profesional está sujeta siempre a restricciones organizacionales de presupuesto y fecha. El trabajo del administrador del proyecto es asegurarse de que el proyecto de software cumpla y supere tales restricciones, además de que entregue software de alta calidad.

Las metas importantes son:

- Entregar el software al cliente en el tiempo acordado.
- Mantener costos dentro del presupuesto general.
- Entregar software que cumpla con las expectativas del cliente.
- Mantener un equipo de desarrollo óptimo y con buen funcionamiento.

Tales metas no son únicas para la ingeniería de software, pero sí lo son para todos los proyectos de ingeniería. Sin embargo, la ingeniería de software es diferente en algunas formas a otros tipos de ingeniería que hacen a la gestión del software particularmente desafiante. Algunas de estas diferencias son:

- 1) El producto es intangible. Los administradores de proyectos de software no pueden constatar el progreso con sólo observar el artefacto que se construye.
- 2)Los grandes proyectos de software con frecuencia son proyectos excepcionales. Incluso los administradores que cuentan con vasta experiencia pueden encontrar difícil anticiparse a los problemas. Aunado a esto, los vertiginosos cambios tecnológicos en computadoras y comunicaciones pueden volver obsoleta la experiencia de un administrador.
- 3)Los procesos de software son variables y específicos de la organización

Resúmenes de libros y papers Unidad 1:

Orphans Preferred: Overview: Describe las características que supuestamente tienen los programadores. Y luego define cuales son las reales y cuáles debería tener un buen diseñador de software.

Características de la personalidad de los grandes diseñadores:

- Tienen un gran set de patrones estándar que aplican a cada nuevo problema. Si este entra en un patrón existente, lo pueden resolver con una técnica familiar.
- Tienen maestría de las herramientas que usan.
- No están asustados de la complejidad, incluso son atraídos a la misma. Su objetivo es hacer lo que parece complejo en simple.
- No tienen miedo de utilizar la fuerza bruta para resolver un problema.

-Tienen experiencia en proyectos fallidos.

10 Software essentials - Steve McConnell:

Overview: El libro habla sobre las cosas esenciales que se necesitan a la hora de desarrollar software.

10 Esencias:

1. Especificación de producto: Es la brújula del proyecto. Sin una buena dirección el trabajo de los individuales puede ir en la dirección equivocada y distintas personas pueden trabajar en propósitos enfrentados.
2. Prototipo de Interfaz de usuario: La documentación en papel generalmente no puede describir adecuadamente la apariencia solicitada del producto. Una ventaja es que el software “funcionalmente” visible es bueno para la moral de los desarrolladores y los clientes.
3. Programación Realista: Provee el fundamento esencial del planeamiento para un adecuado aseguramiento de calidad y el nivel apropiado de formalidad en los procesos del proyecto.
4. Prioridades Explícitas: Ayudan a evadir el problema de querer todas las características posibles con la mejor calidad en los tiempos más cortos con el menor esfuerzo. Hacen las decisiones difíciles más fáciles.
5. Gestión de riesgo activa: Si no atacas los riesgos activamente, ellos te atacaran a ti.
6. Plan de aseguramiento de calidad: Orienta el proyecto hacia la detección de defectos tempranamente, no permitiéndoles infectar el trabajo más tarde en el proyecto.
7. Lista detallada de actividades: Comparando la lista de actividades completadas con la lista de actividades planeadas indica cuando un proyecto está en tiempo o necesita ser rescatado.
8. Administración de la configuración de software: Ayuda a evitar riesgos. Al nivel más básico se utiliza código para automatizar la administración del código fuente. Al nivel más alto los proyectos también colocan, diseños, requerimientos, y materiales de planeamiento bajo la administración de configuración.
9. Arquitectura de SW: Promueve el diseño y las aproximaciones de implementación de forma consistente. Facilitando futuras correcciones y extensiones.
10. Plan de integración: ayuda evitar los problemas de integración.

No Silver Bullet: Overview: Habla sobre los problemas del desarrollo del software, y como conclusión dice que no hay una única solución, sino que hay distintas herramientas que se han desarrollado pero ninguna es la solución definitiva.

Propiedades Inherentes del SW:

- Complejidad
- Conformidad
- Cambiabilidad
- Invisibilidad

Herramientas(que no resolvieron el problema):

- Lenguajes de alto nivel
- Programación orientada a objetos
- Inteligencia artificial.
- Sistemas expertos
- Programación gráfica

Herramientas que podrían ser la solución

- Compra vs Construir: Capaz que la mejor solución para construir software, no es construirlo del todo, sino comprar los componentes que ya están hechos.
- Refinamiento de requerimientos y prototipado rápido: La parte más difícil de construir software es decidir precisamente que construir. Y muchas veces el usuario no sabe qué es lo que quiere o cómo responder a las preguntas. Por eso el refinamiento de los requerimientos es importantes. En cuanto a los prototipos estos son importantes ya que el software funcionando sube la moral de los clientes y los desarrolladores.
- Grandes diseñadores: Buscar grandes diseñadores para desarrollar el software, las características estan en la traducción de No Orphans Preferred.

Unidad 2: Gestión Lean-Ágil de productos de Software

Manifiesto Agil/Filosofía Lean

Introduction al Desarrollo Agil

Métodos Ágiles: Los métodos ágiles son adaptables en lugar de predictivos, y orientados a la gente. Son un subconjunto de los métodos iterativos. El modelado ágil no es un proceso completo, sino un conjunto de principios y prácticas para modelado y análisis de requerimientos, que complementa a la mayoría de los ciclos de vida iterativos e incrementales. Las metodologías ágiles son aplicables cuando los problemas a ser resueltos son complejos, es decir, que se está relativamente alejado del acuerdo y la certeza pero no tanto como para caer en una “zona de anarquía”.

Sommerville sobre métodos agiles:

En la actualidad la entrega y el desarrollo rápidos son por lo general el requerimiento fundamental de los sistemas de software. Debido a que dichos negocios funcionan en un entorno cambiante, a menudo es prácticamente imposible derivar un conjunto de software estable. Es posible que sea hasta después de entregar un sistema, y que los usuarios adquieran experiencia con este, cuando se aclaren los requerimientos reales.

Los procesos de desarrollo del software rápido se diseñan para producir rápidamente un software útil. El software no se desarrolla como una sola unidad, sino como una serie de incrementos, y cada uno de ellos incluye una nueva funcionalidad del sistema.

Características Fundamentales:

1. Los procesos de especificación, diseño e implementación están entrelazados. No existe una especificación detallada del sistema, la documentación del diseño se minimiza.
2. El sistema se desarrolla en diferentes versiones. Los usuarios finales y otros colaboradores del sistema intervienen en la especificación y evaluación de cada versión.
3. Los incrementos son mínimos.
4. Involucran a los clientes en el proceso de desarrollo para conseguir una rápida retroalimentación sobre los requerimientos cambiantes.
5. Minimizan la documentación con el uso de comunicaciones informales, en vez de reuniones formales con documentos escritos.

Los métodos ágiles son métodos de desarrollo incremental donde los incrementos son mínimos. Involucran a los clientes en el proceso de desarrollo para conseguir una rápida retroalimentación sobre los requerimientos cambiantes. Minimizan la cantidad de documentación con el uso de comunicaciones informales, en vez de reuniones formales con documentos escritos.

Los métodos ágiles se apoyan universalmente en el enfoque incremental para la especificación, el desarrollo y la entrega del software. Son más adecuados para el diseño de aplicaciones en que los requerimientos del sistema cambian, por lo general, rápidamente durante el proceso de desarrollo. Tienen la intención de entregar con prontitud el software operativo a los clientes, quienes entonces propondrán requerimientos nuevos y variados para incluir en posteriores iteraciones del sistema.

Valores del Manifiesto Ágil:

1. Individuos e interacciones sobre procesos y herramientas.
2. Software funcionando sobre documentación detallada.
3. Colaboración sobre negociación con el cliente.
4. Responder a cambios por sobre seguir un plan.

12 Principios del manifiesto ágil:

1. La prioridad es satisfacer al cliente a través de releases tempranos y frecuentes.
2. Recibir cambios de requerimientos, aun en etapas finales.
3. Releases frecuentes: Permite una evolución del modelo en espiral y un mejor manejo de los requerimientos. Deben hacerse de 2 semanas a un mes.
4. Técnicos y no técnicos trabajando juntos en todo el proyecto.
5. Hacer proyectos con individuos motivados.
6. El medio de comunicación por excelencia es cara a cara: El espacio físico debe favorecer la comunicación. Esto no significa reducir la documentación.
7. La mejor métrica de progreso es la cantidad de software funcionando, sí bien se colectan otras métricas.
8. El ritmo de desarrollo es sostenible en el tiempo: Más de 40 horas por semana no es sostenible. Debe hacerse con responsabilidad social y efectividad monetaria.
9. Atención continua a la excelencia técnica: mediante mejora continua del producto.
10. Maximización del trabajo no hecho: No implementar más de lo acordado. Tampoco debe confundirse con saltar directo a la codificación.
11. Las mejores arquitecturas, diseños y requerimientos emergen de equipos auto organizados: Se gana sinergia, se deben preservar los equipos de un proyecto a otro.
12. A intervalos regulares, el equipo evalúa su desempeño y ajusta la manera de trabajar.

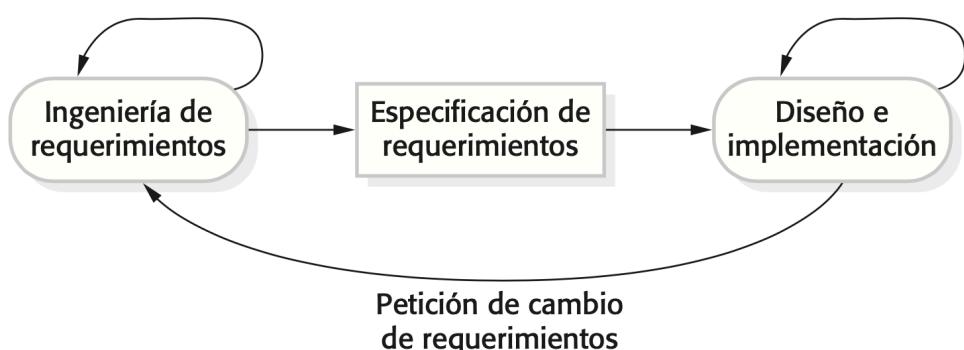
Los métodos ágiles deben apoyarse en contratos, en los cuales el cliente pague por el tiempo requerido para el desarrollo del sistema, en vez de hacerlo por el desarrollo de un conjunto específico de requerimientos.

Los apasionados de los métodos ágiles argumentan que escribir la documentación es una pérdida de tiempo y que la clave para implementar software es producir un código legible de alta calidad. De esta manera, las prácticas ágiles enfatizan la importancia de escribir un código bien estructurado y destinar el esfuerzo en mejorar el código.

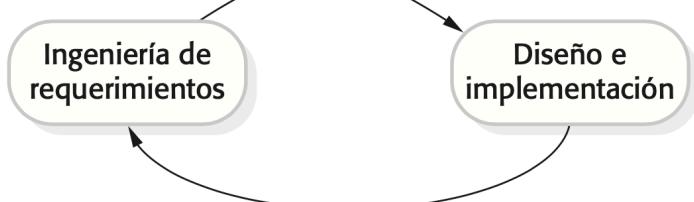
Desarrollo dirigido por un plan y desarrollo ágil

Los enfoques ágiles en el desarrollo de software consideran el diseño y la implementación como las actividades centrales en el proceso del software. Incorporan otras actividades en el diseño y la implementación, como la adquisición de requerimientos y pruebas.

Desarrollo basado en un plan



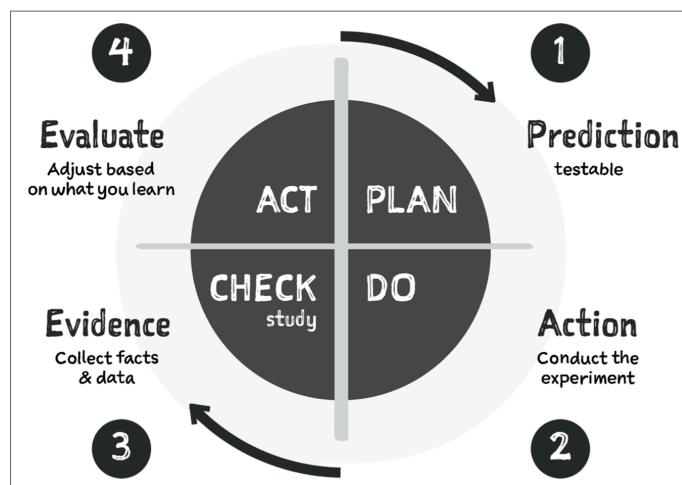
Desarrollo ágil



Donde han tenido éxito?

- Compañía desarrolla producto pequeño o mediano para su venta.
- Diseño de sistemas a la medida dentro de una organización, donde hay un claro compromiso del cliente por intervenir en el proceso de desarrollo.

Filosofia Lean



The Lean mindset is a management philosophy that embraces scientific thinking to explore how right our beliefs and assumptions are while improving a system. Lean practitioners use the deliberate practice of testing their hypotheses through action, observing what actually happens, and making adjustments based on the differences observed. It's how organizations set their course, learn by doing, and decide what to do next on their journey to achieve outcomes.

Lean is a management philosophy for improving any system that produces value—that is, any organization. Persistent improvement, high quality, and reduction in waste are some common characteristics of Lean management.

Principios Lean:



-Eliminar el desperdicio: Evitar re trabajo, reducir el tiempo removiendo lo que no agrega valor

-Amplificar el aprendizaje: Crear y mantener una cultura de mejora continua y solución del problemas. Un proceso localizado en crear conocimiento esperara que el diseño evolucione durante la codificación y no perderá tiempo definiéndolo en forma completa prematuramente.

-Embeber la integridad conceptual: Encastrar todas las partes del producto o servicio, que tenga coherencia y consistencia.

Integridad Percibida: el producto total tiene un balance entre función, uso, confiabilidad y economía que le gusta a la gente.

Integridad Conceptual: todos los componentes del sistema trabajan en forma coherente en conjunto.

El objetivo es construir con calidad desde el principio, no probar después.

Dos clases de inspecciones:

Inspecciones luego de que los defectos ocurren.

Inspecciones para prevenir defectos.

Si se quiere calidad no inspeccione después de los hechos!

Si no es posible, inspeccione luego de pasos pequeños.

-Diferir compromisos:

El último momento responsable para tomar decisiones (en el cual todavía estamos a tiempo). Si nos anticipamos tenemos información parcial.

- Se relaciona con el principio ágil: decidir lo más tarde posible pero responsablemente. No hacer trabajo que no va a ser utilizado. Enlaza con el principio anterior de aprendizaje continuo, mientras más tarde decidimos más conocimiento tenemos.

Entregar rápido: estabilizar ambientes de trabajo a su capacidad más eficiente y acotar los ciclos de desarrollo.

- Entregar rápidamente esto hace que se vayan transformando “n” veces en cada iteración. Incrementos pequeños de valor. Llegar al producto mínimo que sea valioso. Salir pronto al mercado.

- Relacionado con el principio Ágil de entrega frecuente.

- **-Dar poder al equipo:** ejemplo, vamos a comer a un restaurante y no nos metemos en la cocina del restaurante. Nos fijamos en el precio, pedimos y esperamos. Hay mucho micro management, el dueño no decide cuánta sal poner a la comida.

- Respetar a la gente • Entrenar líderes

- Fomentar buena ética laboral

- Delegar decisiones y responsabilidades del producto en desarrollo al nivel más bajo posible

- **Ágil:** El propio equipo pueda estimar el trabajo.

Ver el todo: tener una visión holística, de conjunto (el producto, el valor agregado que hay detrás, el servicio que tiene los productos como complemento).

Valores:

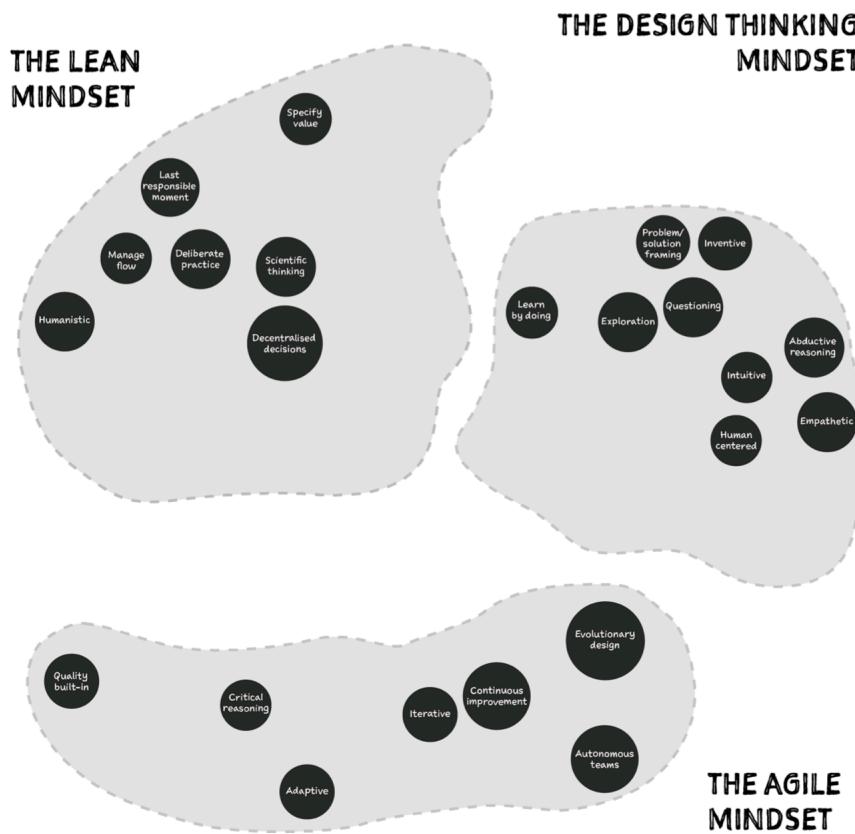
- Learning and adapting over analysis and prediction

- Empowered people are happier and achieve better outcomes

- Outcomes over outputs

- Manage flow to optimize value

- Quality is a result not an activity.



Commonalities of Lean and Agile

Agile and Lean are similar in the following ways:

- They embrace and adapt to change, regardless of how late it occurs.
- They produce value iteratively, in short cycles.
- Each is humanistic, that is, valuing people above process, and encouraging autonomy and collaboration.
- Both Agile and Lean focus on quality, which in turn improves efficiency.⁷
- They seek to eliminate wasted effort.
- They continuously improve through reflection and learning.

How Lean and Agile Are Different

- Agile optimizes software delivery; Lean optimizes systems of work
- Continuous flow versus time-based iterations
- Stable and repetitive versus always changing

Requerimientos en ambientes ágiles - User Stories:

User Stories Applied - Mike Cohn + A User Story Primer

User Story: Describe funcionalidad que tendrá valor para un usuario o un comprador del sistema o software. Las historias de usuario están compuestas de tres partes:

1. Una descripción escrita de la historia usada para planear como un recordatorio.
2. Conversaciones sobre la historia, que sirve para conocer los detalles de la misma.
3. Pruebas que documentan detalles que van a ser usados para determinar cuando una historia está completa.

Características(Sumario):

- Las cartas de historia son la parte visible de una historia. Pero la parte más importante es la conversación entre el cliente y los desarrolladores sobre la historia.
- Las historias son priorizadas según el valor para la organización.
- Si una historia no entra en una iteración, se puede dividir en dos.

- Son de gran ayuda ya que enfatizan la comunicación verbal, y pueden ser entendidas equivalentemente por los usuarios y los desarrolladores.

Ventajas de la US:

- Enfatizan lo verbal antes que la comunicación escrita.
- Son comprendidas por los usuarios y los desarrolladores.
- Son del tamaño ideal para planear.
- Funcionan para desarrollo iterativo.

Las historias de Usuario no son Requerimientos

Mientras que las user stories hacen la mayoría del trabajo previamente hecho por la especificación de requerimientos, casos de uso, etc, estas son materialmente diferentes en distintas formas:

- No no son especificaciones de requerimientos detalles(algo que el sistema debe hacer), en vez son expresiones negociables de intención(se necesita que haga algo como esto).
- Son cortas y fáciles de leer, entendibles por los devs, stakeholders y usuarios.
- Son relativamente fáciles de estimar.
- Representan pequeños incrementos de funcionalidad valuada.
- Se necesita poco o nada de mantenimiento y pueden ser seguramente descartadas después de la implementación.

The customer team: El equipo incluye aquellos que aseguran que el software va a cumplir las necesidades de sus usuarios. Esto quiere decir que el equipo debe incluir, testers, product manager, usuarios reales, y diseñadores de interacción.

Las historias pueden ser escritas en cualquier momento durante el proyecto. El customer team permanece altamente involucrado durante la iteración, hablando con los desarrolladores sobre las historias que están siendo desarrolladas durante la misma. Durante la iteración el equipo también especifica las pruebas y trabajos con los desarrolladores para automatizar y correr las pruebas. También se aseguran que el proyecto está en constante movimiento hacia la entrega del producto deseado.

Escribir Historias: Para escribir buenas historias nos enfocamos en seis atributos: **INVEST**

- **Independent:** Evitar la introducción de dependencias entre historias. Estas pueden llevar a problemas de priorización y planeamiento. Significa que la historia puede ser desplegada, testeada y potencialmente entregada por su cuenta.
- **Negotiable:** Las historias son negociables. No son contratos escritos o requerimientos que el software debe implementar. Las cartas de historia son cortas descripciones de funcionalidad, los detalles de los cuales tienen que ser negociados en una conversación entre el cliente y el equipo de desarrollo.
- **Valuable:** Deben tener valor para el usuario. Para lograr esto se puede hacer que el usuario escriba las historias. Es el atributo más importante en el modelo INVEST. Crear historias con valor requiere reorientar nuestras estructuras funcionales de horizontales a verticales(Ejemplo de la torta con el patrón layered). Creamos historias que corten a través de la arquitectura para que podamos presentar valor al usuario y encontrar su feedback lo más rápido posible.
- **Estimatable:** Es importante que los desarrolladores puedan estimar el tamaño de una historia o la cantidad de tiempo que le llevará transformar la historia en un código funcionando. Si el equipo no puede estimar una historia generalmente indica que la historia es muy larga o incierta. Si es muy larga de estimar podría ser dividida en historias más pequeñas. Si la historia es muy incierta para estimar, entonces una

spike técnica o funcional puede ser utilizada para reducir la incertidumbre, así una o más historias estimables se dan como resultado.

- **Small:** El tamaño de las historias importa, si son muy chicas o grandes no se pueden utilizar para planear. Las épicas deberían ser divididas en historias más chicas. Y cuando las historias son muy chicas deberían ser unidas. Las historias de usuario deberían ser lo suficientemente chicas para que sean completadas en una iteración, sino no pueden proveer valor alguno o ser consideradas terminadas a ese punto. Las historias pequeñas no solo van más rápido por su tamaño pero también por su complejidad disminuida, y la complejidad tiene una relación no lineal con el tamaño.
- **Testable:** Las historias deben ser escritas para que puedan ser probadas. Si esta no puede ser probada, como los desarrolladores saben cuando se finalizó el código. Las historias no testeables comúnmente se muestran como requerimientos no funcionales.

En algunos casos una historia puede ser muy larga o compleja, o la implementación de la misma es pobremente entendida. En este caso, construye una spike técnica o funcional para resolverlo. Y luego divídela en historias basadas en el resultado.

Spikes: Son un tipo especial de historia utilizados para manejar un riesgo o incertidumbre en una user story u otra parte del proyecto. Estas pueden ser utilizadas por una serie de razones:

1. El equipo no tiene conocimiento sobre un nuevo dominio. La spike podría ser utilizada para una investigación básica y familiarizarse con la nueva tecnología o dominio.
2. La historia puede ser muy grande para ser estimada apropiadamente, y el equipo utiliza la spike para analizar el comportamiento implicado, para que puedan dividir la historia en partes estimables.
3. La historia puede contener riesgo técnico significante, y el equipo tendrá que hacer investigación o prototipado para ganar confianza en una aproximación tecnológica que los permita construir la historia de usuario en el futuro.
4. La historia podrá contener **riesgo funcional** significante, no se tiene claro como el sistema debe interactuar con el usuario para obtener el beneficio implicado.

Spikes Técnicas y Funcionales:

Técnicas: Son utilizadas para investigar enfoques técnicos en el dominio de la solución.

Funcionales: Cuando hay una incertidumbre significativa de cómo un usuario interactuará con el sistema. Generalmente son mejores evaluadas mediante prototipos para obtener retroalimentación de los usuarios.

Guía para las Spikes: Estimables, Demostrables y aceptables

Como las otras historias las spikes son colocadas en el backlog, estimables y amoldadas para entrar en una iteración. Los resultados de la spike son diferentes de una historia, ya que generalmente producen información en vez de código funcionando. Una spike podrá resultar en una decisión, un prototipo, evidencia, concepto, o alguna solución parcial para ayudar la llegada a los resultados finales.

La salida de una spike es demostrable al equipo. Estas son aceptadas por el product owner cuando el criterio de aceptación es completado.

La excepción no la regla: Cada historia de usuario tiene incertidumbre y riesgo, es

I a naturaleza misma del desarrollo ágil.

Cada historia contiene actividades del nivel de una spike para eliminar los riesgos técnicos y funcionales. Una spike, por otro lado debe ser reservada para desconocidos más críticos y largos.

Backlog: Backlog de Producto: Contiene la lista de requerimientos(o user stories en agil). Se listan características, funciones y mejoras que se aplicarán al producto. Al inicio no está completo, pero solo se necesita lo suficiente para el primer Sprint. Debe estar priorizado por el product owner. Las estimaciones son un esfuerzo de las partes e iterativas. Si un ítem no puede ser debidamente estimado, se debe dividir en el backlog.

Modelado de roles de Usuario:

Rol de Usuario:Es una colección de atributos definidos que caracterizan una población de usuarios y sus interacciones con el sistema.

Técnicas Adicionales:

Persona: es una representación imaginaria de un rol de usuario. Debería ser descrita lo suficiente para que todos en el equipo sientan que conocen a esta persona. Las historias se vuelven mucho más expresivas cuando las colocamos en términos de un rol de usuario o persona.

Extreme Characters PDA: Los desarrolladores deberían considerar usuarios con personalidades exageradas. Estos pueden llevar a encontrar nuevas historias.

Usuarios Representantes (Proxies): Cliente,vendedores, capacitadores, etc.

Un proyecto debería incluir uno o más usuarios reales del customer team. Los user proxies, quienes no son usuarios pero lo son en el proyecto para representar usuarios.

Pruebas de Aceptación de Historias de Usuario:

La prueba está vista como un proceso de dos paso. Primero notas sobre futuras pruebas son escritas detrás de las cartas. Segundo, las notas de prueba son transformadas en pruebas completas utilizadas para demostrar que la historia ha sido correcta y completamente codificada. Las pruebas de aceptación proveen criterios básicos utilizados para determinar si una historia fue totalmente implementada. También pueden proveer gran información a los programadores que pueden usar antes de codear la historia.

Estimaciones en ambientes ágiles

Estimación de User Stories

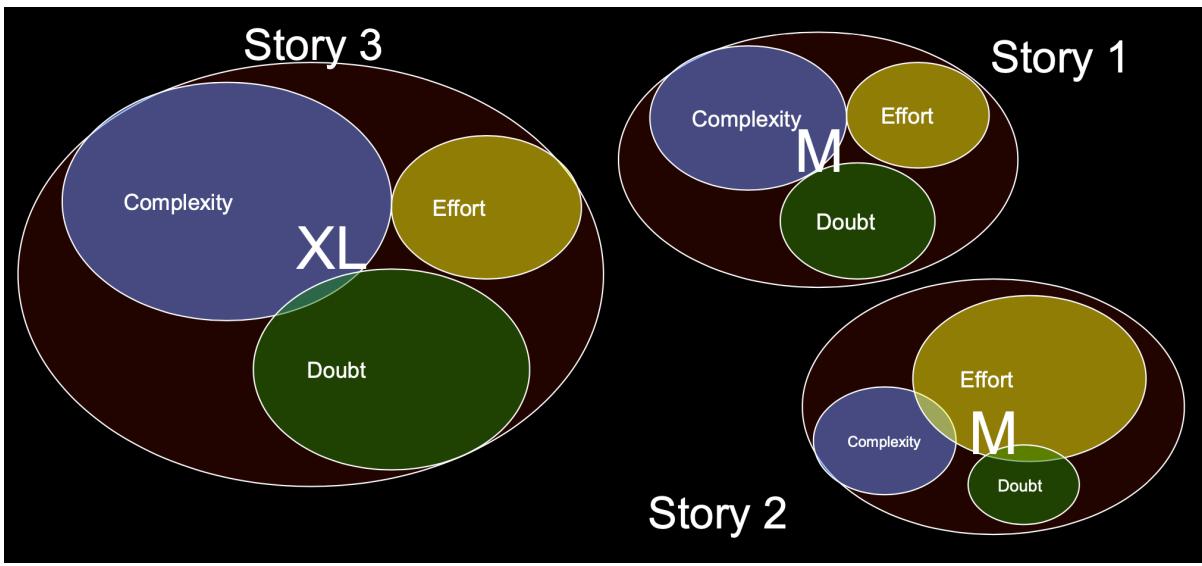
Story Points: Son una unidad de medida no basadas en tiempo. Son definidas por el equipo. Las medidas relativas no son absolutas. Los story points no son una medida basada en tiempo.

Estimación Relativa:

- Las personas no saben estimar en términos absolutos
- Comparar generalmente es mas rapido

Sumario:

- Estima las historias con story points, los cuales son estimaciones relativas de la complejidad, esfuerzo y duración de la historia.
- Las estimaciones deben ser realizadas por el equipo, y las estimaciones son propiedad del equipo en vez de los individuales.
- Triangular las estimaciones comparandolas con otras estimaciones.



Framework SCRUM

Somerville:

El enfoque estándar de la administración de proyectos es el basado en un plan. El enfoque SCRUM es un método ágil general, su enfoque esta en la administración iterativa del desarrollo, y no solo en enfoques técnicos específicos de la ingeniería de software ágil. Este proceso no prescribe el uso de las prácticas de programación. Por lo tanto puede utilizarse en enfoques ágiles más técnicos como XP para ofrecer al proyecto un marco administrativo.

Scrum Basics (Manifiesto)

Scrum: Un Marco de trabajo por el cual las personas pueden abordar problemas complejos adaptativos, a la vez entregar productos del máximo valor posible productiva y creativamente.

Scrum es liviano, fácil de entender y difícil de dominar.

La esencia de scrum es un pequeño equipo de personas. El equipo individual es altamente flexible y adaptativo.

Teoría de Scrum

Scrum se basa en la teoría de control de procesos empírica. El empirismo asegura que el conocimiento procede de la experiencia y de tomar decisiones basándose en lo que se conoce. Scrum emplea un enfoque iterativo e incremental para optimizar la predictibilidad y el control del riesgo.

Tres pilares soportan toda la implementación del control de procesos empírico:

Transparencia: Los aspectos significativos del proceso deben ser visibles para aquellos que son responsables del resultado

Inspección: Los usuarios de Scrum deben inspeccionar frecuentemente los artefactos de Scrum y el progreso hacia un objetivo para detectar variaciones indeseadas.

Adaptación

Si un inspector determina que uno o más aspectos de un proceso se desvían de límites aceptables y que el producto resultante será inaceptable, el proceso o el material que está siendo procesado deben ajustarse. Dicho ajuste debe realizarse cuanto antes para minimizar desviaciones mayores.

Scrum prescribe cuatro eventos formales, contenidos dentro del Sprint, para la inspección y adaptación:

Planificación del Sprint (Sprint Planning)

Scrum Diario (Daily Scrum)
Revisión del Sprint (Sprint Review)
Retrospectiva del Sprint (Sprint Retrospective)

Los valores del SCRUM

Compromiso
Coraje
Foco
Apertura
Respeto

Scrum Team

El equipo Scrum consiste en un Product Owner, el Development Team y un Scrum master. Los equipos de scrum son autoorganizados y multifuncionales. Los equipos autoorganizados eligen la mejor forma de llevar a cabo su trabajo y no son dirigidos por personas externas al equipo. Los equipos multifuncionales tienen todas las competencias necesarias para llevar a cabo el trabajo sin depender de otras personas que no son parte del equipo. El modelo de equipo en Scrum esta diseñado para optimizar la flexibilidad, la creatividad y la productividad.

Los equipos de Scrum entregan productos de forma iterativa e incremental, maximizando las oportunidades de obtener retroalimentacion.

Product Owner: Responsable de maximizar el valor del producto resultante del trabajo del equipo de desarrollo. Es la única persona responsable de gestionar el Product BackLog

Gestionar el BackLock incluye:

- Expresar claramente los elementos del mismo
- Ordenar los elementos del Backlog para alcanzar los objetivos y misiones de la mejor manera posible.
- Optimizar el valor del trabajo que el equipo de desarrollo realiza.
- Asegurar que el BackLog es visible, transparente y clara para todos y que muestra aquello en lo que se trabajara a continuación.
- Asegurar que el equipo de desarrollo entiende los elementos del Backlog al nivel necesario.

Development Team: El Equipo de Desarrollo consiste en los profesionales que realizan el trabajo de entregar un Incremento de producto “Terminado” que potencialmente se pueda poner en producción al final de cada Sprint. La organización es la encargada de estructurar y empoderar a los Equipos de Desarrollo para que estos organicen y gestionen su propio trabajo. La sinergia resultante optimiza la eficiencia y efectividad del Equipo de Desarrollo.
Caracteristicas:

- Son autoorganizados (ni siquiera el Scrum Master indica como convertir elementos del BackLog en Incrementos de funcionalidad)
- Multifuncionales: Como equipo cuentan con todas las habilidades necesarias para crear el incremento.
- Scrum no reconoce títulos para los miembros del equipo.
- Scrum no reconoce subequipos en los equipos de desarrollo.
- La responsabilidad recae en el equipo completo.

Tamaño del Equipo: El tamaño óptimo del Equipo de Desarrollo es lo suficientemente pequeño como para permanecer ágil y lo suficientemente grande como para completar una cantidad de trabajo significativa.

Scrum Master: Responsable de promover y apoyar scrum como se define en la guia.

El servicio al dueño del producto:

- Asegurar que los objetivos, el alcance y el dominio del producto sean entendidos por todos en el equipo.

- Encontrar técnicas para gestionar el backlog de manera efectiva.

- Ayudar al equipo a entender la necesidad de contar con elementos del BackLog claros y consisos.

- Entender la planificación del producto en un entorno empírico.

- Facilitar los eventos de Scrum según requiera o se necesite

- Entender y practicar la agilidad.

Al Equipo de desarrollo:

- Guia al equipo en ser autoorganizado y multifuncional.

- Ayudar al equipo a crear productos de alto valor

- Facilitar los eventos de Scrum según se necesite.

- Guia al equipo en el entorno organizacional del Scrum.

Al master de la organización:

- Liderar y guiar a la organización en la adopción de scrum.

- Planificar las implementaciones de scrum en la organización.

Eventos de SCRUM

En scrum existen eventos predefinidos con el fin de crear regularidad y minimizar la necesidad de reuniones no definidas. Todos los eventos son bloques de tiempo (time-boxes) de tal modo que todos tienen una duración máxima. Una vez que comienza un Sprint su duración es fija y no puede modificarse. Los temas eventos pueden terminar siempre que se alcance el objetivo del evento.

Ademas del propio Sprint que es un contenedor del resto de los eventos, cada uno de los eventos de Scrum constituye una oportunidad formal para la inspección y adaptación de algún aspecto.

Sprint: Es un time-box de un mes o menos durante el cual se crea un incremento de producto “terminado” utilizable y potencialmente desplegable.

Los sprints contienen y consisten en La planificación del Sprint (Sprint Planning), los Scrum Diarios (Daily Scrums), el trabajo de desarrollo, la Revisión del Sprint (Sprint Review) y la Retrospectiva del Sprint (Sprint Retrospective).

Durante el Sprint:

- No se realizan cambios que puedan afectar al objetivo del sprint (Sprint Goal).

- Los objetivos de calidad no disminuyen

- El alcance puede clarificarse y renegociarse entre el Product Owner y el Dev team.

Sprint Planning: El trabajo a realizar durante el sprint se planifica. Este plan se crea mediante el trabajo colaborativo del equipo completo. Tiene máximo de duración de ocho horas para un Sprint de un mes.

Se trabajan las siguientes preguntas:

- Que puede entregarse en el Incremento resultante del Sprint que comienza?

La entrada a esta reunión esta constituida por el BackLog, el ultimo incremento, la capacidad proyectada del Dev Team para el Sprint y el rendimiento pasado del Dev Team.

Durante la planificación del Sprint, el equipo también define el Sprint Goal. Este debería lograrse durante el sprint a través de la implementación del Product BackLog.

- Como se conseguirá hacer el trabajo necesario para entregar el incremento?

Sprint BackLog: Los elementos del BackLog seleccionados para el sprint.

Si el equipo de desarrollo determina que tiene demasiado trabajo o no suficiente puede renegociar los elementos del BackLog con el Product Owner.

Al final la SprintPlanning el Dev Team debería ser capaz de explicar al Product Owner y al Scrum Master como pretende trabajar como un equipo para lograr el Sprint Goal

Daily Scrum: Es una reunión de 15 minutos. En el cual se planea el trabajo para las siguientes 24 horas. El Daily Scrum se realiza a la misma hora y en el mismo lugar todos los días para reducir la complejidad. Se responden cosas como que se hizo ayer, que se hará hoy y si hay algún impedimento para lograr el objetivo.

Los scrums diarios mejoran la comunicación, eliminan la necesidad de realizar otras reuniones, identifican impedimentos a remover, resaltan y promueven la toma rápida de decisiones y mejoran el nivel de conocimiento del Dev Team.

Sprint Review: Se inspecciona el incremento y adapta el BackLog si es necesario. Durante la revisión el Scrum Team y los Stakeholders colaboran acerca de lo que se hizo durante el sprint. Se trata de una reunión informal y la presentación del incremento tiene como objetivo facilitar la retroalimentación de información. Suelen ser hasta 4 horas para sprints de un mes. El Scrum Master dirige el evento. Se incluyen los siguientes elementos:

-El equipo de Scrum y los Stakeholders clave

-El product owner explica que elementos del Backlog se han “terminado” y cuales no.

-El Dev Team hace una muestra del trabajo que ha terminado.

El resultado del Sprint Review es un Product Backlog revisado que define los elementos para el siguiente Sprint. Es posible que la lista reciba un ajuste general para enfocarse en nuevas oportunidades.

Sprint Retrospective: es una oportunidad para el Equipo Scrum de inspeccionarse a sí mismo y de crear un plan de mejoras que sean abordadas durante el siguiente Sprint.

La Retrospectiva de Sprint tiene lugar después de la Revisión de Sprint y antes de la siguiente Planificación de Sprint. Se trata de una reunión de, a lo sumo, tres horas para Sprints de un mes.

El propósito de la Retrospectiva de Sprint es:

Inspeccionar cómo fue el último Sprint en cuanto a personas, relaciones, procesos y herramientas;

Identificar y ordenar los elementos más importantes que salieron bien y las posibles mejoras; y,

Crear un plan para implementar las mejoras a la forma en la que el Equipo Scrum desempeña su trabajo.

Artefactos

Product BackLog: Es una lista ordenada de todo lo que se conoce que es necesario en el producto. Una Lista de Producto nunca está completa. El desarrollo más temprano de la misma solo refleja los requisitos conocidos y mejor entendidos al principio. La Lista de Producto evoluciona a medida de que el producto y el entorno en el que se usará también lo hacen. La Lista de Producto es dinámica; cambia constantemente para identificar lo que el producto necesita para ser adecuado, competitivo y útil.

La Lista de Producto enumera todas las características, funcionalidades, requisitos, mejoras y correcciones.

El refinamiento (refinement) de la Lista de Producto es el acto de añadir detalle, estimaciones y orden a los elementos de la Lista de Producto. Se trata de un proceso continuo en el cual el Dueño de Producto y el Equipo de Desarrollo colaboran acerca de los

detalles de los elementos de la Lista de Producto. Durante el refinamiento de la Lista de Producto, se examinan y revisan sus elementos.

Incremento

El Incremento es la suma de todos los elementos de la Lista de Producto completados durante un Sprint y el valor de los incrementos de todos los Sprints anteriores. Al final de un Sprint el nuevo Incremento debe estar “Terminado”, lo cual significa que está en condiciones de ser utilizado y que cumple la Definición de “Terminado” del Equipo Scrum.

Definición de “Terminado” (Definition of “Done”)

Cuando un elemento de la Lista de Producto o un Incremento se describe como “Terminado”, todo el mundo debe entender lo que significa “Terminado”. Aunque esto puede variar significativamente para cada Equipo Scrum, los miembros del Equipo deben tener un entendimiento compartido de lo que significa que el trabajo esté completado para asegurar la transparencia. Esta es la definición de “Terminado” para el Equipo Scrum y se utiliza para evaluar cuándo se ha completado el trabajo sobre el Incremento de product.

Metricas Agiles

Métricas(Velocity): Representa la cantidad de trabajo que puede ser realizada en una iteración. Una vez que conocemos la velocidad del equipo, podemos utilizarla para transformar los días ideales en días de calendario.

Hay tres formas de obtener el valor inicial de la velocidad:

1. Usar Valores históricos.
2. Correr una iteración inicial y utilizar la velocidad de la misma.
3. Asumir una velocidad.

Velocidad en Agile Estimating and Planning - Mike Cohn:

Usar Valores Históricos: estos son muy buenos, si los tienes. Son el mejor recurso cuando ha cambiado muy poco entre el proyecto viejo y el equipo y el nuevo proyecto y el equipo. Cualquier cambio de personal o tecnologías significantes reducirán la usabilidad de las medidas históricas de velocidad. Aun así estas velocidades deberían ser utilizadas con un rango.

Correr una iteración: La mejor forma de anticipar la velocidad es corriendo una o un par de iteraciones y luego estimar la velocidad de lo observado durante las iteraciones. Esta debería ser siempre la aproximación por defecto.

Asumir una velocidad: A veces no es posible correr iteraciones para poder medir la misma, el proyecto comienza en un año por ejemplo. Es aquí cuando debemos estimar la velocidad basándonos en distintos aspectos como la experiencia de los desarrolladores.

Grooming the product backlog (Engominando el backlog) - Roman Pichler

El backlog grooming, también llamado refinamiento del backlog, es la actividad de mantener el producto actualizado. Esto es necesario, a medida que el backlog del producto tiende a cambiar basándose en lo aprendido obtenido del desarrollo de software y la exposición del mismo a los clientes, usuarios, y otros stakeholders.

Ayuda a integrar las últimas percepciones dentro del backlog. Esto asegura que se desarrolla el producto correcto de la manera correcta. También se asegura que el backlog es trabajable, que hay suficiente items listos para empezar el siguiente sprint.

El refinamiento del backlog consiste de los siguientes pasos:

1. Analizar retroalimentación por los usuarios, stakeholders etc.
2. Integrar el aprendizaje
3. Decidir qué hacer después

4. Tener el backlog listo. Seleccionar el objetivo del sprint y escribir user stories detalladas que están listas.

Llevar a cabo estos pasos deberían resultar en un backlog que es DEEP: Detallado apropiadamente, emergente, estimado, y priorizado.

El refinamiento debería ser un esfuerzo colaborativo que incluya el product owner y los miembros del equipo de desarrollo. Esto ayuda a nivelar el conocimiento y creatividad del equipo, incluyendo tener en cuenta la factibilidad y los riesgos técnicos; Incrementa el entendimiento del equipo sobre los ítems del backlog; reduce el trabajo como product owner y ayuda a asegurarse que los ítems de alta prioridad esten listos.

El refinamiento debería llevarse a cabo cuando comienza el trabajo de desarrollo o durante el sprint.

Cuanto tiempo lleva el refinamiento? Es importante tener en cuenta la etapa del ciclo de vida del producto y la duración del sprint. Mientras más estable y maduro esté el producto, menos es el esfuerzo de refinamiento que hay en los sprints.

La mejor herramienta para llevar a cabo el refinamiento es el Canvas de Producto, un estructurado y multidimensional backlog de producto

Backlog Grooming Segundo paper:

Actividades que incluye el refinamiento:

1. Remover user stories que ya no son relevantes
2. Crear nuevas historias con nuevas necesidades descubiertas
3. Cambiar la prioridad relativa de las historias.
4. Corregir estimaciones
5. Dividir user stories

Beneficios esperados:

El refinamiento del backlog es para asegurar que el backlog permanece poblado con ítems que son relevantes, detallados y estimados a un nivel apropiados con su prioridad, y mantener el actual entendimiento del proyecto y sus objetivos.

Herramientas para gestión de productos

Lean UX

Los problemas de hoy en día no es con los diseñadores o los ingenieros, son los sistemas que usamos para crear compañías. Seguimos construyendo organizaciones lineales en un mundo que demanda cambio constante. Seguimos construyendo silos en un mundo que demanda colaboración. Y seguimos invirtiendo en análisis, discutiendo sobre las especificaciones, y produciendo eficientemente los deliverables en un mundo que demanda continua experimentación para obtener innovación continua.

Lean Start-up es una carpa grande. Se construye en ideas establecidas de muchas disciplinas, desde la manufactura lean hasta design thinking. La union de Lean Startup y User Experience-based design y su simbólica coexistencia es Lean UX.

Que es Lean-UX y como es diferente?

Los principios Lean debajo de Lean Startup aplican a Lean UX en tres formas. Primero, nos ayudan a remover el desperdicio de nuestro proceso de diseño de UX. Nos alejamos de documentos pesados a un proceso que crea solo el diseño de los artefactos que necesitamos para mover hacia adelante el aprendizaje del equipo.

Segundo, nos ayudan a armonizar nuestro sistema de diseñadores, desarrolladores, product managers, ingenieros de aseguramiento de calidad, etc. En una transparente y multifuncional colaboración que trae a los no diseñadores a nuestro proceso de diseño. Por ultimo y capaz que lo mas importante, es el cambio de mentalidad que ganamos por adoptar un modelo basado en experimentación. En vez de depender en un diseñador héroe que obtenga la mejor solución desde un solo punto de vista, usamos experimentación rápida y medición para aprender rápidamente qué tan bien o no nuestras ideas cumplen nuestros objetivos. En todo esto, el rol del diseñador comienza a evolucionar hacia la facilitación del diseño, y con eso tomamos un nuevo conjunto de responsabilidades.

A parte del Lean Startup, LeanUX tiene otras fundaciones: Design Thinking y filosofías de desarrollo agil. Design thinking toma una aproximación enfocada en resolver el problema, trabajando colaborativamente para iterar en un infinito, camino hacia la perfección. Trabaja hasta los objetivos del producto mediante ideación específica, prototipado, implementación y pasos de aprendizaje para traer una solución apropiada a la luz. Agile se reconcentra en el desarrollo de software en el valor. Parece buscar software funcionando para los clientes de forma rápida y ajustar regularmente nuevo aprendizaje a lo largo del camino.

Lean UX usa estas fundaciones para romper el estancamiento entre la velocidad de Agile y la necesidad de diseñar en el ciclo de vida del desarrollo del producto.

Las tres fundaciones de Lean UX

1)Design Thinking: Es importante porque toma la posición explícita de que cada aspecto de un negocio puede ser aproximado con métodos de diseño. Le da a los diseñadores permisos y precedentes para trabajar más allá de sus límites típicos. También anima a los no diseñadores a utilizar métodos de diseño para resolver los problemas que enfrentan en sus roles. Design Thinking es una fundación crítica que anima a los equipos a colaborar atravesando roles y considerar el diseño del producto desde una perspectiva holística.
2)Desarrollo de Software Agil: Lean UX aplica los cuatro principios de Agile para diseñar el producto:

1)Individuos y interacciones sobre procesos y herramientas: Para generar las mejores soluciones rápido debes relacionar el equipo completo. Las ideas deben ser intercambiadas libre y frecuentemente.

2)Software funcionando sobre documentación comprensiva: Cada negocio tiene soluciones infinitas y cada miembro del equipo tendrá una opinión de cuál es la mejor. El desafío es descubrir cuál opinión es la mejor. Construyendo software lo más rápido posible, las soluciones pueden ser asesoradas para un encaje en el mercado y viabilidad.

3)Colaboración con clientes sobre negociación del contrato: Colaborar con tus compañeros de equipo y clientes construye un compartido entendimiento del espacio del problema y las soluciones propuestas. El resultado son iteraciones más rápidas, menor dependencia en documentación pesada, etc.

4) Responder al cambio sobre seguir un plan: La asunción en Lean UX es que el diseño inicial del producto va a estar mal, entonces el objetivo debería ser encontrar que está mal con ellos lo más rápido posible.

3)Método Lean Startup: Basado en un bucle de retroalimentación llamado “construye-mide-aprende” para minimizar el riesgo del proyecto y tener a los equipos construyendo y aprendiendo rápidamente. El equipo construye MVPs (Minimum Viable Products) y los entrega lo más rápido posible para comenzar el proceso de aprendizaje lo más antes posible. En Lean UX cada diseño es una solución de negocio propuesta - una hipótesis. Tu objetivo es validar esta solución lo más eficientemente posible usando el feedback del cliente. La cosa más pequeña que puedes construir para probar cada hipótesis es el MVP. El MVP no

necesita estar hecho de código, puede ser una aproximación de la experiencia final. Colectas lo que aprendes del MVP y evolucionas tus ideas. Y luego lo haces de nuevo.

La práctica de Lean UX es: Es una práctica de traer la verdadera naturaleza de un producto a la luz de una manera rápida, colaborativa y cross-funcional que reduce la énfasis en la documentación mientras incrementa el enfoque en construir un entendimiento compartido de la experiencia actual del producto que está siendo diseñada.

Principios de Lean UX:

Equipos Cross-Functionals

Pequeño, Dedicado, Colocado -> Mantener equipos pequeños

Problem Focused Teams

Progresso = Outcomes, not output: Features y servicios son outputs. Los objetivos de negocio que se intentan conseguir son outcomes.

Remover Desperdicio

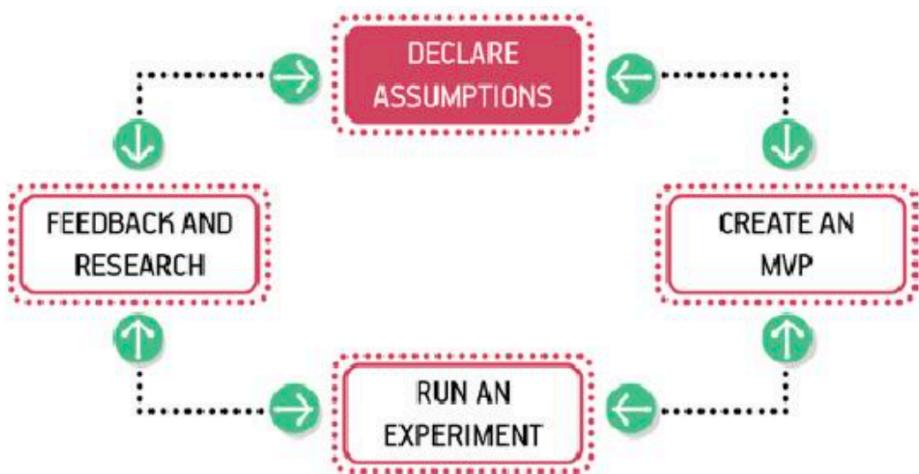
Entrega Continua

Making over analysis

Learning over Growth

Permission to fail

El proceso de Lean UX



Design Thinking

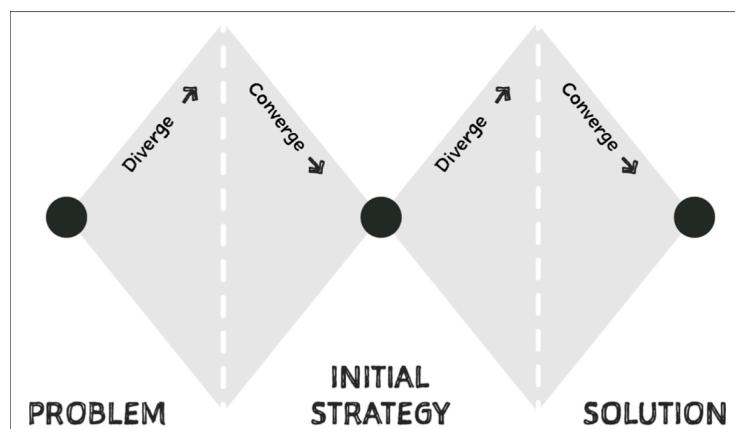


Figure 1-2. The divergence and convergence of Design Thinking

Design Thinking es una mentalidad así como un toolkit de técnicas para aplicar las formas de un diseñador que tiene para pensar y hacer. Podemos aplicarlo en cualquier contexto, dominio y problema. Nos ayuda a explorar un nuevo territorio y definir un rango de potenciales soluciones. Norman lo define como:

Los diseñadores no buscan una solución hasta que han determinado el problema real, incluso ahí, en vez de resolver el problema, se detienen para considerar un amplio rango de soluciones potenciales. Solo ahí convergerán en su propuesta. Este proceso es llamado Design Thinking.

Los componentes en la definición de Norman son:

- Determinar el problema
- Buscar soluciones
- Considerar muchas opciones
- Converger en una propuesta

Design thinking no es algo especial, que solo los diseñadores hacen. Todos diseñamos, sea conscientemente o no.

Unidad 3: Gestión del Software como producto (Gestión de la configuración)

Resumen del PPTN5:

Software Configuration Management SCM:
(Definición de la IEEE)

Una disciplina que aplica dirección y monitoreo administrativo y técnico a: identificar y documentar las características funcionales y técnicas de los ítems de configuración, controlar los cambios de esas características, registrar y reportar los cambios y su estado de implementación y verificar correspondencia con los requerimientos.

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

La configuración involucra:

- Identificarla en un momento dado.
- Controlar sistemáticamente sus cambios.
- Mantener su integridad y origen.

Integridad del producto: Satisface las necesidades del usuario, puede ser fácil y completamente rastreado durante su ciclo de vida. Satisface criterios de performance y cumple con sus expectativas de costo.

Problemas en el manejo de componentes:

- Pérdida de un componente
- Perdida de cambios
- Regresión de fallas
- Doble mantenimiento
- Superposición de cambios
- Cambios no validados

Ítem de configuración de software(SCI): Todos y cada uno de los artefactos que forman parte del producto o del proyecto, que pueden sufrir cambios o necesitan ser compartidos entre los miembros del equipo y sobre los cuales debemos saber su estado y evolución. Ejemplos: riesgos, plan de desarrollo, requerimientos, código fuente, gráficos, documento de despliegue, plan de CM, propuesta de cambio, etc.

Versión: La forma particular de un artefacto en un instante o contexto dado.

Variante: Es una versión de un ítem de configuración que evoluciona por separado.

Representan configuraciones alternativas.

Línea Base: Una configuración que ha sido revisada formalmente y sobre la que se ha llegado a un acuerdo. Sirve como base para desarrollos posteriores y puede cambiarse sólo a través de un procedimiento formal de control de cambios. Permiten ir atrás en el tiempo y reproducir un entorno de desarrollo en un momento dado del proyecto.

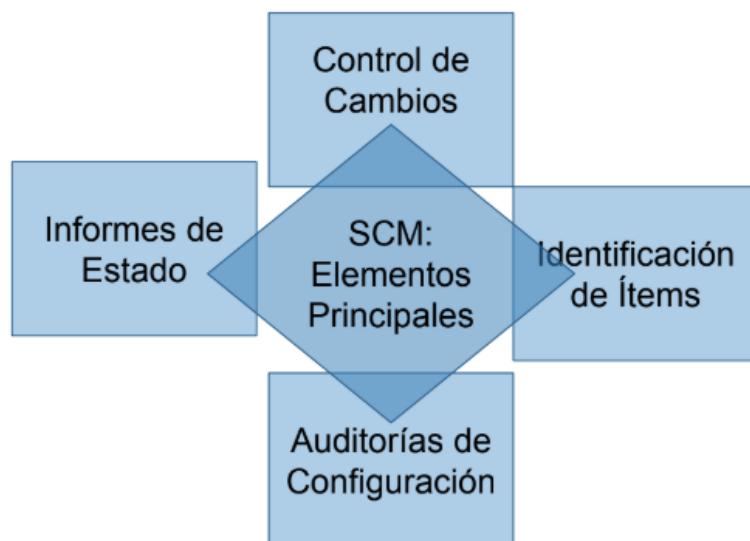
Ramas: Existe una rama principal. Sirven para bifurcar el desarrollo y pueden tener razones de creación con semántica. Pueden ser descartadas o integradas.

Repositorio: Contiene los ítems de configuración. Mantiene la historia de cada uno con sus atributos y relaciones.

Centralizados: Un solo servidor contiene todos los archivos con sus versiones.

Descentralizado: Cada cliente tiene una copia del repositorio.

Actividades Fundamentales de la Administración de Configuración de Software:



Identificación de Ítems de Configuración:

- Identificación única de cada ítem de configuración.
- Convenciones y reglas de nombrado
- Definición de la estructura del repositorio
- Ubicación dentro de la estructura del repositorio

Ítems de

Producto(ERS,arquitectura,código,manual de usuario),

Proyecto(plan de proyecto,cronograma),

Iteración(Plan de iteración, reporte de defectos).

Control de Cambios:

- Tiene su origen en un Requerimiento de Cambio a uno o varios ítems de configuración que se encuentran en una **línea base**.
- Es un procedimiento formal que involucra diferentes actores y una evaluación del impacto del cambio.
- **Comité de control de cambios:** Representantes del área de diseño, implementación, testing, etc.

Auditorías de configuración de Software:

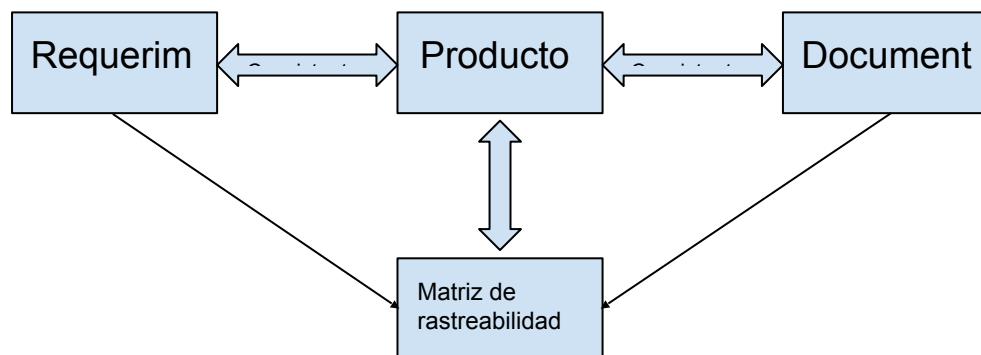
Auditoria Física de configuración(PCA): Asegura que lo que está indicado para cada ICS en la línea base o en la actualización se ha alcanzado realmente.

Auditoria Funcional de configuración (FCA): Evaluación independiente de los productos de software, controlando que la funcionalidad y performance reales de cada ítem de configuración sean consistentes con la especificación de requerimientos.

Auditoria de Gestión de configuración y V&V (Validación y Verificación):

Validación: El problema es resuelto de manera apropiada que el usuario obtenga el producto correcto.

Verificación: Asegura que un producto cumple con los objetivos preestablecidos, definidos en la documentación de línea base.



Informe de Estado:

- Se ocupa de 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 rastreabilidad de todos los cambios realizados a las líneas base durante el ciclo de vida.

Plan de Gestión de Configuración:

Debe Incluir:

- Reglas de nombrado de los CI
- Herramientas a utilizar para SCM
- Roles e integrantes del comité
- Procedimiento formal de cambios
- Plantillas de formularios
- Procesos de Auditoría

SCM en Agile

- Sirve a los practicantes y no viceversa.
- Hace seguimiento y coordina el desarrollo en lugar de controlar a los desarrolladores
- Responde a los cambios en lugar de tratar de evitarlos
- Coordinación y automatización frecuente y rápida
- Feedback continuo y visible sobre calidad, estabilidad e integridad

Tips:

- Es responsabilidad de todo el equipo.
- Automatizar lo más posible

Conceptos Introductorios de la Gestión de Configuración

Los sistemas de software siempre cambian durante su desarrollo y uso. Se descubren bugs y éstos deben corregirse. Los requerimientos del sistema cambian, y es necesario implementar dichos cambios en una nueva versión del sistema. Se dispone de nuevas versiones de hardware y plataformas de sistema, por lo que hay que adaptar los sistemas para que funcionen con ellos. Los competidores introducen nuevas características en sus sistemas que se deben igualar. Conforme se hacen cambios al software, se crea una nueva versión del sistema. En consecuencia, la mayoría de los sistemas pueden considerarse como un conjunto de versiones, cada una de las cuales debe mantenerse y gestionarse.

La administración de la configuración (CM, por las siglas de configuration management) se ocupa de las políticas, los procesos y las herramientas para administrar los sistemas cambiantes de software. Es necesario gestionar los sistemas en evolución porque es fácil perder la pista de cuáles cambios y versiones del componente se incorporaron en cada versión del sistema.

Si no se cuenta con procedimientos efectivos de administración de la configuración, se puede malgastar esfuerzo al modificar la versión equivocada de un sistema, entregar a los clientes la versión incorrecta de un sistema u olvidar dónde se almacena el código fuente del software para una versión particular del sistema o componente.

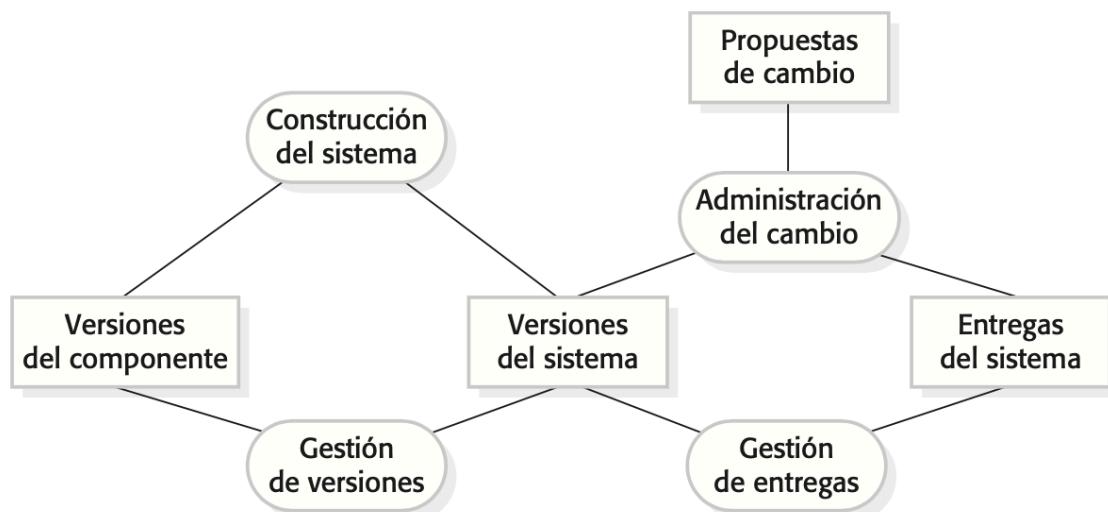
Actividades Relacionadas con la Gestión de Configuración

La administración de la configuración de un producto de sistema de software comprende cuatro actividades estrechamente relacionadas

-Administración del cambio Esto implica hacer un seguimiento de las peticiones de cambios al software por parte de clientes y desarrolladores, estimar los costos y el efecto de realizar dichos cambios, y decidir si deben implementarse los cambios y cuándo.

-Gestión de versiones Esto incluye hacer un seguimiento de las numerosas versiones de los componentes del sistema y garantizar que los cambios hechos por diferentes desarrolladores a los componentes no interfieran entre sí.

-Construcción del sistema Éste es el proceso de ensamblar los componentes del programa, datos y bibliotecas, y luego compilarlos y vincularlos para crear un sistema ejecutable.



-Gestión de entregas (release) Esto implica preparar el software para la entrega externa y hacer un seguimiento de las versiones del sistema que se entregaron para uso del cliente.

La administración de la configuración implica enfrentar un gran volumen de información, por lo que se han desarrollado numerosas herramientas de administración de la configuración para dar soporte a los procesos de CM. Éstos abarcan desde las simples herramientas que apoyan una sola tarea de administración de la configuración, como el rastreo de bugs, hasta complejos y costosos conjuntos de herramientas integradas que apoyan todas las actividades de administración de la configuración.

Las herramientas de administración de la configuración se usan para rastrear las propuestas de cambio, almacenar versiones de componentes del sistema, construir sistemas a partir de dichos componentes, y rastrear liberaciones de las versiones del sistema para los clientes.

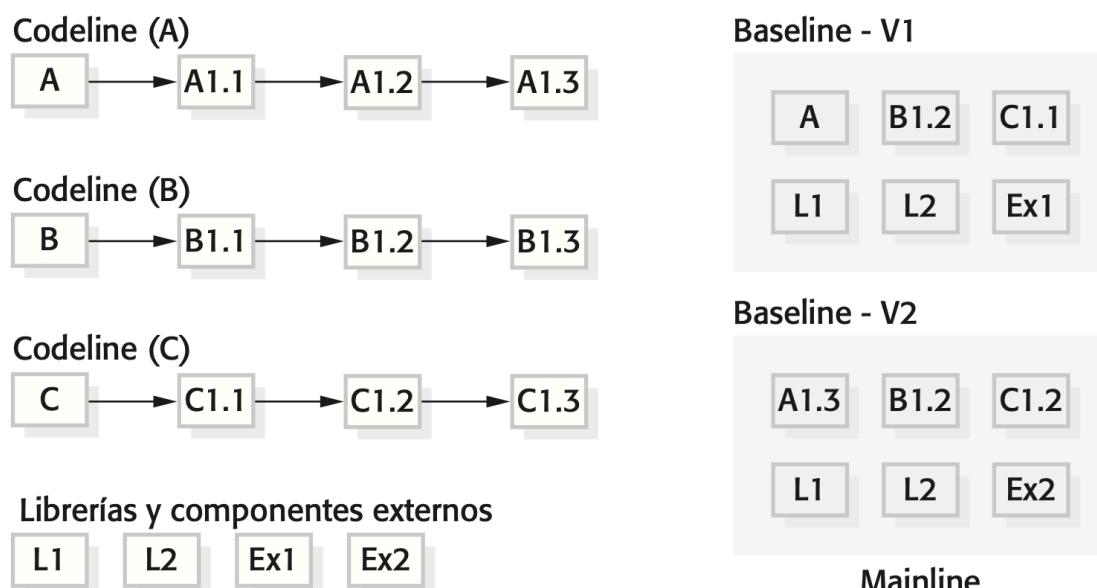
En ocasiones la administración de la configuración se considera parte de la gestión de calidad del software

Versión, Variantes, Release

Conceptos

Término	Explicación
Ítem de configuración o ítem de configuración de software (SCI, por las siglas de <i>Software Configuration Item</i>)	Cualquier aspecto asociado con un proyecto de software (diseño, código, datos de prueba, documento, etcétera) se coloca bajo control de configuración. Por lo general, existen diferentes versiones de un ítem de configuración. Los ítems de configuración tienen un nombre único.
Control de configuración	El proceso de asegurar que las versiones de sistemas y componentes se registren y mantengan de modo tal que los cambios se gestionen, y se identifiquen y almacenen todas las versiones de componentes durante la vida del sistema.
Versión	Una instancia de un ítem de configuración que difiere, en alguna forma, de otras instancias del mismo ítem. Las versiones siempre tienen un identificador único, que se compone generalmente del nombre del ítem de configuración más un número de versión.
Línea base (<i>baseline</i>)	Una línea base es una colección de versiones de componente que construyen un sistema. Las líneas base están controladas, lo que significa que las versiones de los componentes que conforman el sistema no pueden ser cambiadas. Por lo tanto, siempre debería ser posible recrear una línea base a partir de los componentes que lo constituyen.
Línea de código (<i>codeline</i>)	Una línea de código es un conjunto de versiones de un componente de software y otros ítems de configuración de los cuales depende dicho componente.
Línea principal (<i>mainline</i>)	Una secuencia de líneas base que representa diferentes versiones de un sistema.
Entrega, liberación (<i>release</i>)	Una entrega de un sistema que se libera para su uso a los clientes (u otros usuarios en una organización).
Espacio de trabajo (<i>workspace</i>)	Área de trabajo privada donde puede modificarse el software sin afectar a otros desarrolladores que estén usando o modificando dicho software.

Ramificación (<i>branching</i>)	La creación de una nueva línea de código a partir de una versión en una línea de código existente. La nueva línea de código y la existente pueden desarrollarse de manera independiente.
Combinación (<i>merging</i>)	La creación de una nueva versión de un componente de software al combinar versiones separadas en diferentes líneas de código. Dichas líneas de código pueden crearse mediante una rama anterior de una de las líneas de código implicadas.
Construcción de sistema	Creación de una versión ejecutable del sistema al compilar y vincular las versiones adecuadas de los componentes y las librerías que constituyen el sistema.



Actividades:

Administracion del Cambio

El cambio es un hecho en la vida de los grandes sistemas de software. Las necesidades y los requerimientos organizacionales cambian durante la vida de un sistema. El proceso de administración del cambio se ocupa de analizar los costos y beneficios de los cambios propuestos, aprobar aquellos que lo ameritan e indagar cuál o cuáles de los componentes del sistema se modificaron.

El proceso de administración del cambio se inicia cuando un “cliente” completa y envía una petición de cambio en que se describe el cambio requerido al sistema.

Después de enviar una petición de cambio, ésta se verifica para asegurarse de que sea válida. Esta verificación puede venir tanto del cliente como del equipo de soporte de la aplicación, o para peticiones internas de un miembro del equipo de desarrollo. La comprobación es necesaria porque no todas las peticiones de cambio requieren acción. Si la petición de cambio es un reporte de bug, tal vez éste ya haya sido reportado

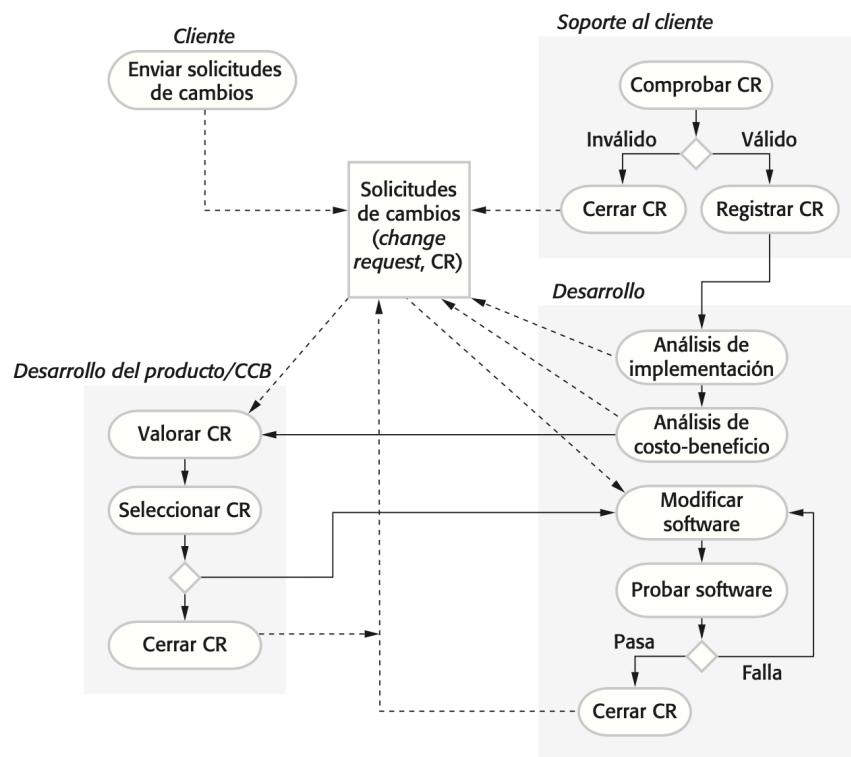


Figura 25.3 Proceso de administración del cambio

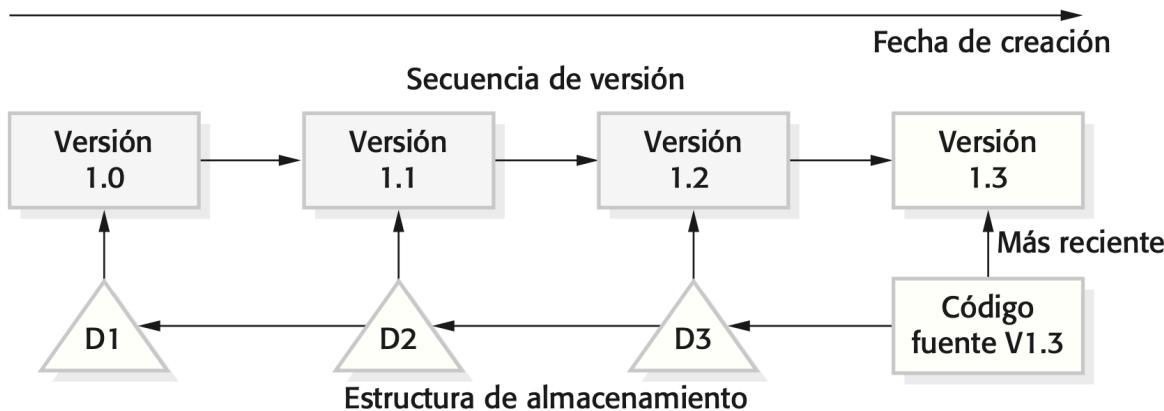
Continuando con este análisis, un grupo separado debe determinar si realizar el cambio al software es rentable desde una perspectiva empresarial. Para sistemas militares y gubernamentales este grupo se conoce usualmente como consejo de control del cambio (CCB, por las siglas de *change control board*). En la industria puede llamarse “grupo de desarrollo del producto”, el cual es el responsable de tomar las decisiones sobre cómo debe evolucionar el sistema de software. Este grupo debe revisar y aprobar todas las peticiones de cambio, a menos que los cambios impliquen simplemente corregir errores menores en pantallas de despliegue, páginas Web o documentos. Estas peticiones menores deben transmitirse al equipo de desarrollo sin un análisis detallado, pues esto podría ser más costoso que implementar el cambio.

El CCB o el grupo de desarrollo del producto consideran el efecto del cambio desde un punto de vista estratégico y organizacional más que técnico. Deciden si el cambio en cuestión está justificado económico y prioriza los cambios aceptados para su implementación. Los cambios aceptados se transmiten de regreso al grupo de desarrollo; las peticiones de cambio rechazadas se cierran y ya no se emprenden más acciones.

Gestión de Versiones

La gestión de versiones (VM, por las siglas de *version management*) es el proceso de hacer un seguimiento de las diferentes versiones de los componentes de software o ítems de configuración, y los sistemas donde se usan dichos componentes. También incluye asegurar que los cambios hechos a dichas versiones por los diferentes desarrolladores no interfieran unos con otros. Por lo tanto, se puede considerar a la gestión de versiones como el proceso de administrar líneas de código y líneas base.

Para soportar la gestión de versiones, siempre se deben usar herramientas de gestión de versiones (llamadas en ocasiones sistemas de control de versiones o sistemas de control de código fuente). Estas herramientas identifican, almacenan y controlan el acceso a las diferentes versiones de los componentes.



Los sistemas de gestión de versiones ofrecen a menudo varias características:

1. Identificación de versión y entrega (release)
1. Gestión de almacenamiento: En vez de conservar una copia completa de cada versión, el sistema almacena una lista de diferencias (deltas) entre una versión y otra.
2. *Registro del historial de cambios* Todos los cambios realizados al código de un sistema o componente se registran y enumeran.
3. Desarrollo independiente Es posible que diferentes desarrolladores trabajen en el mismo componente al mismo tiempo
4. *Soporte de proyecto* Un sistema de gestión de versiones puede soportar el desarrollo de varios proyectos que comparten componentes

Deltas: las deltas definen cómo recrear versiones anteriores del sistema.

Para apoyar el desarrollo independiente sin interferencia, los sistemas de gestión de versiones usan el concepto de repositorio público y un espacio de trabajo privado. Los desarrolladores sacan componentes del repositorio público hacia su espacio de trabajo privado y pueden cambiarlos como deseen en su mismo espacio. Cuando sus cambios están completos, ingresan los componentes al repositorio

Construcción del sistema

La construcción del sistema es el proceso de crear un sistema ejecutable y completo al compilar y vincular los componentes del sistema, librerías externas, archivos de configuración, etcétera.

Existe una gran cantidad de herramientas de construcción disponibles, y un sistema de construcción puede ofrecer algunas de las siguientes características o todas ellas:

1. Generación de rutinas (scripts) de construcción
2. Integración del sistema de gestión de versiones
2. Recompilación mínima
3. Creación de sistema ejecutable

Gestión de entregas de software (*release*)

Una entrega (*release*) de sistema es una versión de un sistema de software que se distribuye a los clientes. Para software de mercado masivo es posible identificar por lo general dos tipos de entregas: release mayor, que proporciona funcionalidad significativamente nueva, y release menor, que repara bugs y corrige problemas reportados por el cliente.

Una entrega de sistema no sólo es el código ejecutable del sistema; ésta también puede incluir:

- Archivos de configuración que definan cómo debe configurarse la entrega (release) para instalaciones particulares;
- archivos de datos, como los archivos de mensajes de error, necesarios para la operación exitosa del sistema;
- un programa de instalación para ayudar a instalar el sistema en el hardware objetivo;
- documentación electrónica y escrita que describa al sistema;
- empaquetado y publicidad asociada diseñados para dicha entrega.

La creación de entrega es el proceso de instaurar la colección de archivos y documentación que incluyen todos los componentes de la entrega del sistema

Resumen

- La administración de la configuración es la gestión de un sistema de software en evolución. Cuando se mantiene un sistema se establece un equipo CM para garantizar que los cambios se incorporen en el sistema de una forma controlada y que se mantienen registros con los detalles de los cambios que se implementaron.
- Los principales procesos de administración de la configuración se ocupan de la administración del cambio, gestión de versiones, construcción del sistema y gestión de entregas de software (release). Para apoyar todos estos procesos se dispone de herramientas de software.
- La administración del cambio implica valorar las propuestas de cambios por los clientes del sistema y otros interesados, así como decidir si es conveniente en términos de costos implementarlos en una nueva versión de un sistema.
- La gestión de versiones incluye hacer un seguimiento de las diferentes versiones de los componentes de software que se crean conforme se hacen cambios en ellos.
- La construcción del sistema es el proceso de ensamblar componentes de sistema en un programa ejecutable para que operen en un sistema de cómputo objetivo.
- El software debe reconstruirse frecuentemente y probarse de inmediato después de construir una nueva versión. Esto facilita la detección de bugs y problemas que se introdujeron desde la última construcción.
- Las entregas de sistema contienen código ejecutable, archivos de datos, archivos de configuración y documentación. La gestión de entregas incluye tomar decisiones referentes a las fechas de entrega del sistema, preparar toda la información para su distribución y documentar cada entrega del sistema.

Gestion de Configuracion en ambientes agiles

Agile SCM

SCM es crucial para el éxito de los proyectos ágiles. Un equipo sin prácticas efectivas de SCM se realentiza por el trabajo extra que cada desarrollador debe realizar cuando no hay una manera fácil de sincronizarse con sus compañeros. Un bien diseñado y liviano proceso de SCM acelera el proyecto.

Prácticas del SCM

Tu proyecto debería utilizar una estructura simple de ramificación. Las ramas de desarrollo para el código fuente de un ítem de configuración son generalmente codelines. Una o dos copetines son generalmente suficientes para proyectos ágiles que no requieren múltiples releases paralelos.

Los conceptos clave para la organización de la evolución del código son: una Mainline, una Active Development Line, y una Release Line.

Mainline: Es la única codeline central en la cual los artefactos de librería son reportados. Tener un único punto de integración hace más fácil de recrear el último estado de los ítems de configuración.

Active Development Line: Es la codeline que está configurada con una política de codeline que permite al trabajo en proceso ser incorporado sin ningún riguroso precheck de validación.

Release Line: Es la codeline para los estados que están disponibles para los clientes para arreglar problemas del sistema en preparación para el ciclo de release. Esta tiene políticas de codeline más estrictas que aseguran estabilidad.

Cada desarrollador debería trabajar en su Workspace privado que contiene el código en el cual está trabajando actualmente, completado desde un Repositorio. El desarrollador no necesita ejecutar pruebas exhaustivas antes de realizar los cambios. Pero debería crear una Construcción Privada y ejecutar un Smoke Test para asegurarse que el check in no causara problemas.

Control de Cambios

En un diálogo de dos lados con el usuario o los stakeholders el equipo licita los requerimientos en forma de solicitud de características y escenarios de usuario. Estas solicitudes pueden ser para nueva funcionalidad, o para cambios o adiciones a funcionalidad ya implementada. Los desarrolladores estiman cada pedazo de funcionalidad solicitada, y el cliente da las prioridades de las características para trabajar en la próxima iteración.

Los requerimientos son administrados mediante el seguimiento de la lista de características de la iteración actual, y el backlog actual de las características de solicitada pero aún no implementadas. Al final de cada iteración el actual Backlog junto a las nuevas solicitudes son prioridades nuevamente por el cliente para que el desarrollador pueda decidir qué conjunto de características trabajara en la siguiente iteración. Esta frecuente repriorización y proceso de selección es el equivalente más común de un CCB (Change Control Board).

Identificación y Almacenamiento

La identificación de los ítems de configuración generalmente es hecha con repositorios de texto y código fuente utilizando herramientas de control de versión como CVS.

Autorización de Cambio

El desarrollador de un proyecto ágil está autorizado a cambiar el código fuente en dos casos:

- 1) Para implementar una característica aceptada para la iteración actual.
- 2) Para mejorar la simplicidad (lectura y mantenimiento) del código fuente.

Los equipos ágiles están típicamente autorizados para realizar pequeños cambios de refactorización en cualquier momento sin la solicitud de aprobación.

El control de cambios de desarrollo generalmente es informal. El sistema de control de código fuente marca los cambios y puede llegar a notificar automáticamente al realizar un check in o check out.

Little Book of Configuration Management

El éxito de un proyecto es altamente dependiente de una administración de configuración efectiva. Es esencial a medida que el tamaño del software se incrementa. Es necesaria para permitir a un equipo grande trabajar juntos en un entorno estable.

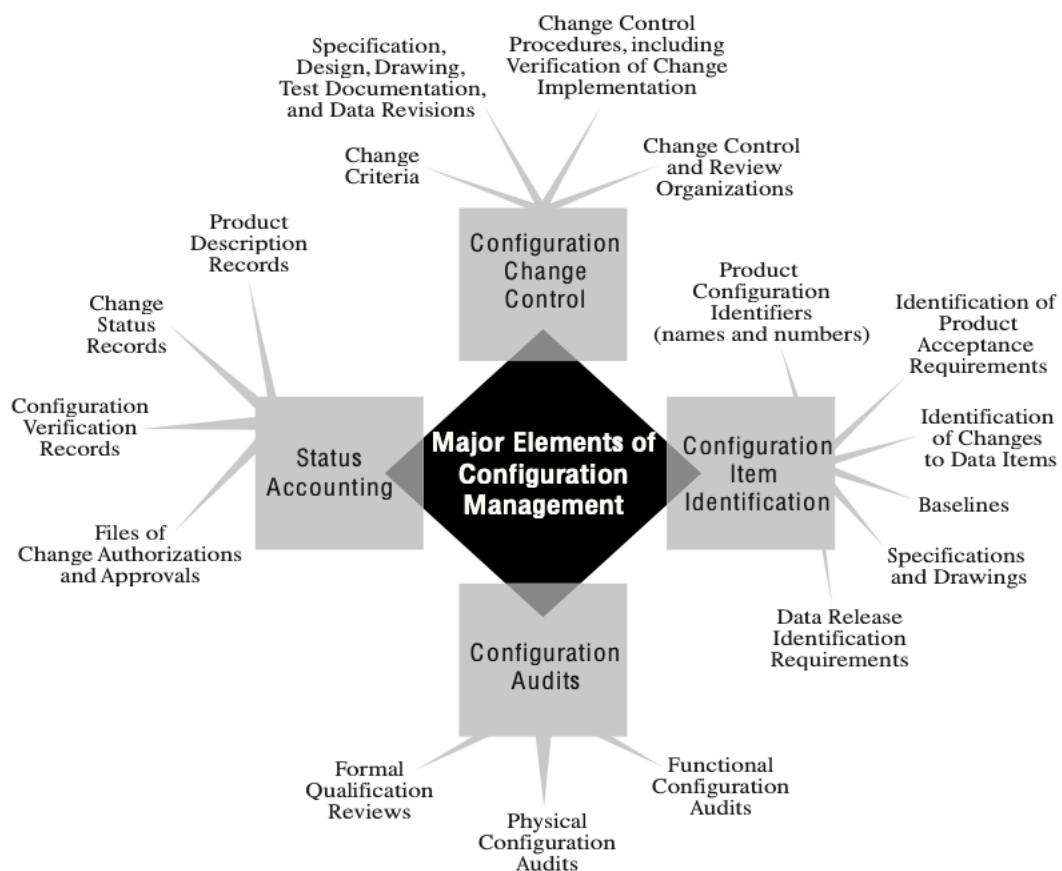
SCM: son los medios por el cual el contenido, cambios, o estados de la información compartida en un proyecto es administrada y controlada. Es una disciplina que aplica direcciones técnicas y administrativas para el control de cambio e integralidad de los datos y documentos del producto.

Propósito fundamental: Es establecer y mantener la integridad y control de los productos de software durante el ciclo de vida. Eso incluye productos como requerimientos de performance, atributos funcionales y físicos, y diseño e información de operación. CM es una disciplina aplicando direcciones técnicas y administrativas para el control del cambio e integridad de los datos y documentación del producto. CM incluye identificar la configuración del software, controlando sistemáticamente los cambios de la configuración, y manteniendo la integridad y trazabilidad de la configuración durante el ciclo de vida del proyecto.

El proceso de SCM esta compuesto por las siguientes actividades:

- Identificación de los artefactos usados o desarrollados por el proyecto.
- Control de cambios de configuración de la información, incluyendo impacto de los cambios, prácticas de administración, calendarios, presupuestos, actividades técnicas, testing, etc.
- Contabilidad del estado de los distintos artefactos utilizados en el proyecto.
- Auditorías y revisiones de la configuración.
- Procedimientos administrativos para la entrega y release del proyecto. Y la capacidad de monitorear el estado de la información del proyecto.

CM es el mecanismo de control de proyecto básico que establece y mantiene la integridad de los productos de software durante el ciclo de vida del proyecto.



CM nos provee:

-Identificación de la Configuración: La habilidad de identificar qué información ha sido aprobada para usar en el proyecto, quién es dueño de la información, como esta información fue aprobada y el ultimo release aprobado.

-Control de Configuration: Los procesos y procedimientos del control de configuración designando el nivel de control que cada producto de trabajo debe pasar, identificando las personas o grupos con autoridad para autorizar cambios y hacer cambios a cada nivel, los pasos a seguir para obtener la autorización requerida, el proceso de solicitudes de cambio, y el mantenimiento de versiones pasadas.

-Contabilidad de Estado: Registro y reportes formalizados para establecer los documentos de configuración, el estado de los cambios propuestos, y el estado de la implementación de los cambios aceptados.

-Revisiones y Auditorias: Evaluación frecuente de todo el contenido, la integridad de la linea base, y la integridad de todos los productos controlados para asegurar que son conformes a sus documentos de configuración.

CM es esencial para el éxito del proceso por las siguientes razones:

1) CM son los medios por el cual la información compartida es controlada y mantenida.

2) Los métodos CM proveen maneras de identificar, seguir y controlar el desarrollo del sistema desde la concepción del sistema hasta que es reemplazado o retirado.

3) La administración de Baselines y productos de ingeniería a través de CM proveen un control sostenido de la información a medida que las funciones de ingeniería trabajan a través del proceso.

4) CM provee visibilidad dentro de los indicadores de control de cambio del proyecto, incluyendo requerimientos cambiados por mes, numero de defectos que son abiertos y cerrados, etc.

Continous Integration

Continous Delivery

Continous Deployment

Continous Integration, Delivery and Deployment - Rosell Sander

Las Fundaciones de CI, Delivery y Deployment

Estas son relativamente nuevas prácticas de desarrollo que ganaron mucha popularidad en los últimos años. Continous Integration es todo sobre validar el software apenas es chequeado en el source control, garantizando que el software funciona y continua funcionando después que nuevo código ha sido escrito. Continous delivery sigue después de la Integración y hace que el software este a un click del deployment. Luego sigue Continous Deployment que automatiza el proceso entero de deployar software a los clientes o los propios servidores.

Si estas tres podrían ser reducidas en una palabra seria Automatización. Las tres prácticas son sobre automatizar el proceso de testar y deployar, minimizando (o eliminando completamente) la intervención humana, minimizando el riesgo de errores, y haciendo que la construcción y el despliegue de nuestro software sea mas fácil hasta el punto que cualquier desarrollador del equipo pueda hacerlo.

El problema de estas técnicas es que no es tan fácil de configurar y lleva mucho tiempo.

Continous Integration

CI es todo sobre tener el software en un estado desplegable en todo momento. Eso es, que el código compila y la calidad del código puede ser asumida como una cantidad razonable de buena calidad.

Source Control

CI comienza con un repositorio compartido, típicamente un sistema de Source Control.

Estos se aseguran que todo el código es guardado en un único lugar. Es importante mantener los check-ins pequeños. Si construyes una nueva característica y se cambian muchos archivos al mismo tiempo, se hace mas difícil encontrar los bugs que aparecen.

CI Server

Los builds son automatizados usando algún servidor CI. Estos monitorean el repositorio y comienzan la construcción con cada check in. Una sola construcción puede compilar el código, correr pruebas unitarias, calcular cobertura del código, checar el estilo, mitificar el código y mucho mas. Cada vez que una construcción falla, por ejemplo, porque un programador olvido un semi-colon el equipo debería ser notificado, esto lo realiza el server.

Continous Delivery

Con continuos delivery los artefactos que son producidos por el servidor CI son desplegados al server de producción con un solo click. Continuos Integration es un pre requisito para Continuos Delivery satisfactorio.

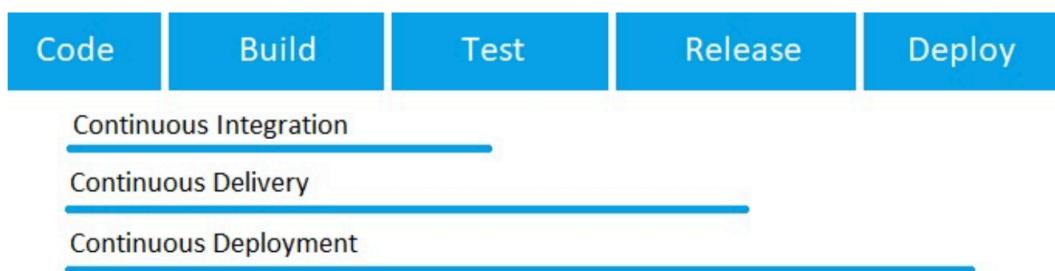
Los beneficios de tener un despliegue automatizado apenas se hace visible es que el despliegue automatizado lo hace mucho mas fácil al despliegue y también nos deja hacerlo con mas frecuencia.

Hacer el proceso de release mas fácil automatizandolo es importante, ya que ahora desplegaras cambios mas pequeños más seguido. Eso significa que es más fácil realizar un rollback en caso de falla, y también va a reducir el riesgo de que alguna falla ocurra en primer lugar. Despise de todo, los cambios relativos a la vieja versión del software permanecen pequeños.

Deberías poder desplegar tu software en un entorno de producción en todo momento. Para alcanzar esto, el software tiene que estar siempre en un estado entrañable, significando que la construcción es exitosa, el código compila, y las pruebas pasan.

Continous Deployment

La etapa final de automatizar el proceso de desarrollo de software es el Continous Deployment. Cuando practicamos esto, cada check in en el source control es desplegado a un entorno de producción en una construcción exitosa. La razón detrás de esto es que vas



a desplegar software antes o después de todas formas. Lo más temprano que lo hagas, mas serán las chances que arregles los bugs mas rápido.

Resumen:

Continuous Deployment ayuda en tener el software para nuestro cliente apenas es escrito. Continuos delivery es una buena alternativa si necesitamos mayor control sobre nuestros despliegues. Para minimizar el riesgo de bugs de despliegue, el software debería ser testado usando Continuous Integration. Este es sobre hacer que nuestro software es probado y desplegable.

Paper Continuous Integration vs Delivery vs Deployment

Continuous Integration:

Los desarrolladores practicando CI suben sus cambios al main branch lo antes posible. Los cambios de los desarrolladores son validados creando y corriendo pruebas automatizadas contra el build. CI pone un gran énfasis en la automatización del testing para checkear que la aplicación no esté rota cada vez que se nuevos comits son integrados al main branch.

Que necesitas:

- El equipo tendrá que escribir pruebas automatizadas para cada nueva característica, mejora o arreglo.
- Se necesita un server CI que puede monitorear el repositorio principal y correr pruebas automáticamente por cada nuevo commit que es pusheado.
- Los desarrolladores deben combinar sus cambios lo más rápido posible.

Que se obtiene:

- Menos bugs llevados a producción.
- La construcción de la release es fácil ya que las regresiones fueron capturadas por las pruebas automáticas
- Los costos de testing son reducidos drásticamente, el server puede correr cientos de pruebas en segundos.
- El QA team gasta menos tiempo testeando y se puede concentrar más en mejorar la cultura de calidad.

Continous Delivery:

Es una extensión de CI para asegurarse que se puede liberar nuevos cambios a los clientes de forma rápida de una manera sustentable. Esto significa en cima de tener testing automatizado, también hay que tener automatizado el proceso de release y se puede desplegar la aplicación en cualquier momento con un solo click. Sin embargo cuando realmente quieras obtener los beneficios de continuous delivery, deberías desplegar a producción lo más rápido posible para asegurarse que los bugs son fáciles de resolver en caso que aparezcan.

Que se necesita:

- Fuerte fundación en la integración continua
- Los despliegues deben ser automatizados. El gatillo sigue siendo manual pero una vez que comienza es automático y no se necesita intervención humana.

Que se gana:

- La complejidad de desplegar software se va. El equipo no debe gastar días preparando el próximo release.
- Se puede liberar más seguido acelerando el bucle de retroalimentación de los clientes.

Continuous Deployment:

Cada cambio que pasa todas las etapas de la producción son liberados a los clientes. No hay intervención humana, y solo los que fallaron alguna prueba prevendrán un nuevo cambio a ser desplegado a la producción.

CD es una excelente manera de acelerar el bucle de retroalimentación con los clientes y sacar presión sobre el equipo ya que no hay un dia de liberación. Ahora los desarrolladores pueden concentrarse en construir software, y ver su trabajo en funcionamiento a pocos minutos de haber terminadolo.

Que se necesita:

- La cultura del testing tiene que ser la mejor. La calidad del testing determinara la calidad de las liberaciones

- Los procesos de documentación deber ir a la par de los despliegues

Que se gana:

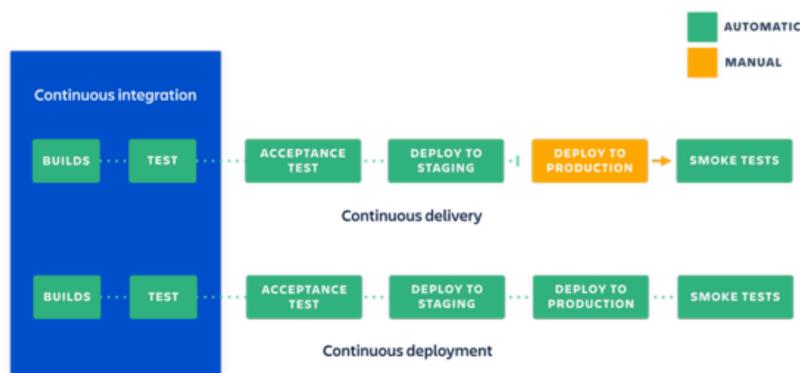
- Se puede desarrollar rápido ya que no hay que pausar el desarrollo de las release.

- Las release son menos riesgosas y fáciles de arreglar en caso de problemas.

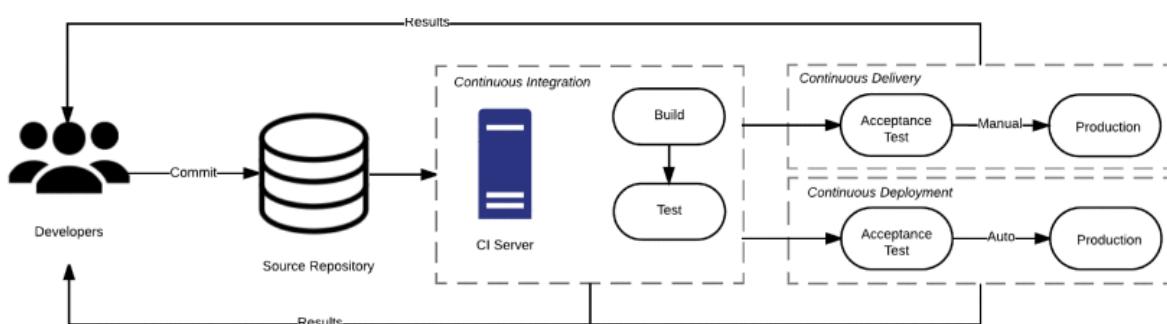
- Los clientes ven una continua mejora, y la calidad incrementa cada día, en vez de cada mes o cuarto de año.

Como las practicas se relacionan entre si?

Continous Integration es parte de CDelivery y CDeployment. Y continuos deployment es comocontinous delivery , excepto que las liberaciones son automaticas.



Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices”, IEEE Access, 2017.



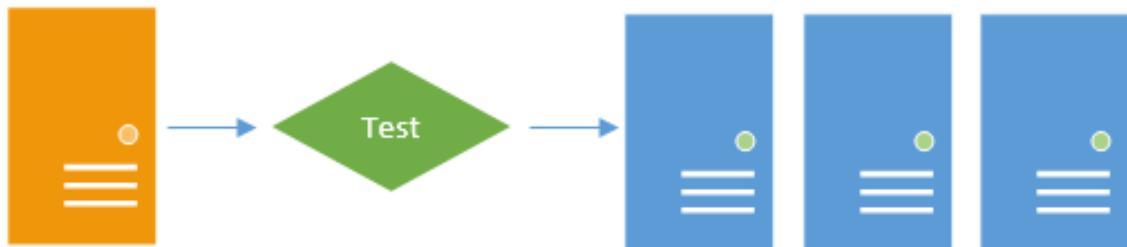
Continous Integration (CI): Es una practica de desarrollo ampliamente establecida en la industria del desarrollo de software, en la cual miembros de un equipo integran y combinan el trabajo de desarrollo frequentemente. CI habilita a las compañías de software a tener ciclos de liberación más cortos y frecuentes, mejorar la calidad del software, e incrementar la productividad del equipo. Esta practica incluye construcción y testing automatizado.

Continous Delivery (CDE): Esta dirigida a asegurar que la aplicación esta siempre en un estado de producción listo después de haber pasado satisfactoriamente las pruebas automatizadas. CDE emplea un conjunto de practicas, como por ejemplo CI y la automatización del despliegue para entregarlo de forma automática. Esta practica reduce el riesgo de despliegue, disminuye los costos y obtiene feedback de los clientes mas rápido.

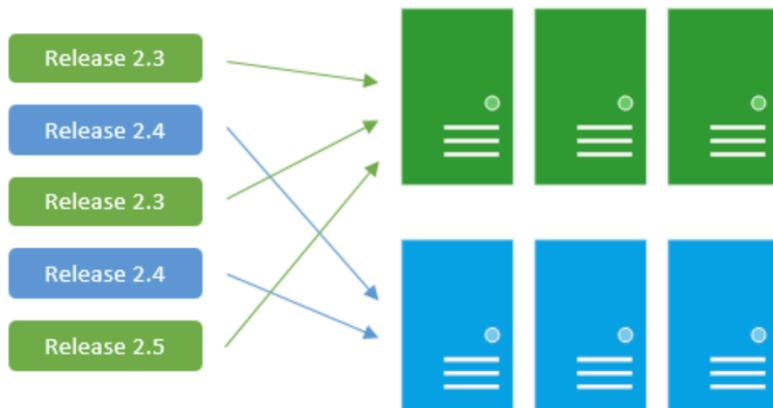
Continous Deployment (CD): La practica va un paso mas adelante y automatica y continuamente despliega la aplicación a producción o a los entornos del cliente.

Estrategias de Deployments - Canary Deployment - Blue/Green Deployment

Canary Deployment: Es un patron para largar liberaciones a un subconjunto de usuarios o servidores. La idea es primero desplegar el cambio a un pequeño conjunto de servidores, probarlo, y luego largarlo al resto de los servidores. El Canary deployment sirve como un indicador temprano de alerta con menor impacto en el downtime: si el Canary deployment falla, el resto de los servidores no son impactados.



Blue/Green Deployment: Es un patrón que reduce el downtime durante los despliegues de producción teniendo dos entornos de producción: verde y azul



En este modelo el modelo, el entorno de producción cambia con cada release. Así como reduce el downtime, también puede ser potente para usar hardware extra comparado a tener un entorno de staging dedicado:

Staging: cuando el azul esta activo, el verde se convierte en el staging environment para el siguiente deployment.

Rollback: Desplegamos azul y lo hacemos activo. Si un problema se descubre. Como el verde sigue corriendo la versión vieja podemos hacer un rollback fácilmente.

Recuperación del desastre: Después de desplegar al azul y estamos satisfechos con qué es estable, podemos desplegar el nuevo release al verde también. Teniendo un entorno de standby en caso de desastre.

Unidad Nro 4: Aseguramiento de Calidad de Proceso y de Producto

Conceptos Generales sobre calidad

Importancia de trabajar para y con Calidad: ventajas y desventajas

Somerville - Cap 24

Los problemas de calidad del software se descubrieron inicialmente en la década de 1960 con el desarrollo de los primeros grandes sistemas de software, y han continuado invadiendo la ingeniería de software a partir de esa década. El software entregado era lento y poco fiable, difícil de mantener y de reutilizar. El descontento con esta situación condujo a la adopción de técnicas formales de gestión de calidad del software, desarrolladas a partir de métodos usados en la industria manufacturera. Estas técnicas de gestión de calidad, en conjunto con nuevas tecnologías y mejores pruebas de software, llevaron a progresos significativos en el nivel general de calidad del software.

La gestión de calidad del software para los sistemas de software tiene tres intereses fundamentales:

1)A nivel de organización, la gestión de calidad se ocupa de establecer un marco de proceso y estándares de organización que conducirán a software de mejor calidad. Esto supone que el equipo de gestión de calidad debe tener la responsabilidad de definir los procesos de desarrollo del software a usar, los estándares que deben aplicarse al software y la documentación relacionada, incluyendo los requerimientos, el diseño y el código del sistema.

2)A nivel del proyecto, la gestión de calidad implica la aplicación de procesos específicos de calidad y la verificación de que continúen dichos procesos planeados; además, se ocupa de garantizar que los resultados del proyecto estén en conformidad con los estándares aplicables a dicho proyecto.

3)A nivel del proyecto, la gestión de calidad se ocupa también de establecer un plan de calidad para un proyecto. El plan de calidad debe establecer metas de calidad para el proyecto y definir cuáles procesos y estándares se usarán.

El aseguramiento de calidad (QA, por las siglas de quality assurance) es la definición de procesos y estándares que deben conducir a la obtención de productos de alta calidad y, en el proceso de fabricación, a la introducción de procesos de calidad. El control de calidad es la aplicación de dichos procesos de calidad para eliminar aquellos productos que no cuentan con el nivel requerido de calidad.

En la mayoría de las compañías, el equipo QA es el responsable de administrar el proceso de pruebas de liberación. Como se explicó en el capítulo 8, esto significa que se aplican las pruebas del software antes de que éste se libere a los clientes. El equipo es responsable de comprobar que las pruebas del sistema cubran los requerimientos y de mantener los registros adecuados del proceso de pruebas.

La gestión de calidad proporciona una comprobación independiente sobre el proceso de desarrollo de software. El proceso de gestión de calidad verifica los entregables del proyecto para garantizar que sean consistentes con los estándares y las metas de la organización (figura 24.1). El equipo QA debe ser independiente del equipo de desarrollo para que pueda tener una perspectiva objetiva del software. Esto les permite reportar la calidad del software sin estar influídos por los conflictos de desarrollo del software.

La planeación de calidad es el proceso de desarrollar un plan de calidad para un proyecto. El plan de calidad debe establecer las cualidades deseadas de software y describir cómo se valorarán. Por lo tanto, define lo que realmente significa software de “alta calidad” para un sistema particular.

Se sugiere un bosquejo de estructura para un plan de calidad. Éste incluye:

- 1)Introducción del producto Una descripción del producto, la pretensión de su mercado y las expectativas de calidad para el producto.
- 2)Planes del producto Indican las fechas de entrega críticas y las responsabilidades para el producto, junto con planes para distribución y servicio al producto.
- 3)Descripciones de procesos Describen los procesos y estándares de desarrollo y servicio que deben usarse para diseño y gestión del producto.
- 4)Metas de calidad Las metas y los planes de calidad para el producto, incluyendo una identificación y justificación de los atributos esenciales de calidad del producto.
- 5)Riesgos y gestión del riesgo Los riesgos clave que pueden afectar la calidad del producto y las acciones a tomar para enfrentar dichos riesgos.

Aunque los estándares y procesos son importantes, los administradores de calidad deben enfocarse también a desarrollar una “cultura de calidad” en la que todo responsable del desarrollo del software se comprometa a lograr un alto nivel de calidad del producto. Deben exhortar a los equipos a asumir la responsabilidad de la calidad de su trabajo y desarrollar nuevos enfoques para el mejoramiento de la calidad

Calidad del software

La calidad del software no es directamente comparable con la calidad en la fabricación. La idea de tolerancia no es aplicable a los sistemas digitales y es prácticamente

imposible llegar a una conclusión objetiva sobre si un sistema de software cumple o no su especificación, por las siguientes razones:

- 1) Es difícil escribir especificaciones de software completas y sin ambigüedades. Los desarrolladores y clientes de software pueden interpretar los requerimientos de diferentes formas y tal vez sea imposible llegar a acuerdos acerca de si el software se desarrolló conforme a su especificación.
- 2) Por lo general, las especificaciones integran requerimientos de varias clases de participantes. Dichos requerimientos son un compromiso ineludible y tal vez no incluyan los requerimientos de todos los grupos de participantes. Por lo tanto, las partes interesadas excluidas quizás perciban el sistema como uno de mala calidad, a pesar de que implementa los requerimientos acordados.
- 3) Es imposible medir de manera directa ciertas características de calidad (por ejemplo, mantenibilidad)

Debido a estos problemas, la valoración de calidad del software es un proceso subjetivo en que el equipo de gestión de calidad tiene que usar su juicio para decidir si se logró un nivel aceptable de calidad.

La calidad subjetiva de un sistema de software se basa principalmente en sus características no funcionales. Esto refleja la experiencia práctica del usuario: Si la funcionalidad del software no es lo que se esperaba, entonces los usuarios con frecuencia sólo le darán la vuelta a este asunto y encontrarán otras formas de hacer lo que quieren. Sin embargo, si el software no esiable o resulta muy lento, entonces es prácticamente imposible que los usuarios logren sus metas. Por consiguiente, la calidad del software no sólo se trata de si la funcionalidad de éste se implementó correctamente, sino también depende de los atributos no funcionales del sistema.

Una suposición que subyace en la gestión de la calidad del software es que la calidad del software se relaciona directamente con la calidad del proceso de desarrollo de software. Esto proviene de nuevo de los sistemas fabriles, donde la calidad del producto está estrechamente relacionada con el proceso de producción. Un proceso de fabricación incluye configurar, establecer y operar las máquinas implicadas en el proceso. Una vez que las máquinas operan correctamente, se sigue de manera natural la calidad del producto. Entonces se mide la calidad del producto y el proceso se modifica hasta que se logra el nivel de calidad necesario.

En la manufactura existe un claro vínculo entre el proceso y la calidad del producto, ya que el proceso es relativamente sencillo de estandarizar y monitorizar. Una vez calibrados los sistemas de fabricación, pueden operar una y otra vez para generar productos de alta calidad; sin embargo, el software no se manufactura, se diseña. Por lo tanto, en el desarrollo del software es más compleja la relación entre calidad de proceso y calidad del producto. El diseño del software es un proceso creativo más que mecánico, pues es significativa la influencia de las habilidades y la experiencia individuales. Factores externos, como la novedad de una aplicación o la premura por el lanzamiento comercial de un producto, también afectan la calidad de éste sin importar el proceso usado.

No hay duda de que el proceso de desarrollo utilizado tiene una influencia importante sobre la calidad del software, y que los buenos procesos tienen más probabilidad de conducir a software de buena calidad. La gestión de la calidad y el mejoramiento del proceso pueden conducir a menores defectos en el software a desarrollar. Sin embargo, es difícil valorar los atributos de calidad del software, como la mantenibilidad, sin usar el software durante un largo periodo. En consecuencia, es difícil decir cómo las características del proceso influyen en dichos atributos. Más aún, debido al papel del diseño y la creatividad en el proceso de software, la estandarización del proceso en ocasiones puede extinguir la creatividad, lo cual, lejos de elevar la calidad, conducirá a un software de calidad inferior.

Estándares de software

Los estándares de software tienen una función muy importante en la gestión de calidad del software. Como se indicó, un aspecto importante del aseguramiento de calidad es la definición o selección de estándares que deben aplicarse al proceso de desarrollo de software o al producto de software.

Los estándares de software son importantes por tres razones:

1. Los estándares reflejan la sabiduría que es de valor para la organización. Se basan en conocimiento sobre la mejor o más adecuada práctica para la compañía
2. Los estándares proporcionan un marco para definir, en un escenario particular, lo que significa el término “calidad”. Como se dijo, la calidad del software es subjetiva, y al usar estándares se establece una base para decidir si se logró un nivel de calidad requerido
3. Los estándares auxilian la continuidad cuando una persona retoma el trabajo iniciado por alguien más. Los estándares aseguran que todos los ingenieros dentro de una organización adopten las mismas prácticas. En consecuencia, se reduce el esfuerzo de aprendizaje requerido al iniciarse un nuevo trabajo.
 - Existen dos tipos de estándares de ingeniería de software relacionados que pueden definirse y usarse en la gestión de calidad del software:

1. *Estándares del producto* Se aplican al producto de software a desarrollar. Incluyen estándares de documentos (como la estructura de los documentos de requerimientos), estándares de documentación (como el encabezado de un comentario estándar para una definición de clase de objeto) y estándares de codificación, los cuales definen cómo debe usarse un lenguaje de programación.
2. *Estándares de proceso* Establecen los procesos que deben seguirse durante el desarrollo del software. Deben especificar cómo es una buena práctica de desarrollo. Los estándares de proceso pueden incluir definiciones de especificación, procesos de diseño y validación, herramientas de soporte de proceso y una descripción de los documentos que deben escribirse durante dichos procesos.

Los estándares deben entregar valor, en la forma de calidad aumentada del producto.

El marco de estándares ISO 9001

ISO 9001, el más general de dichos estándares, se aplica a organizaciones que diseñan, desarrollan y mantienen productos, incluido software

La revisión principal del estándar ISO 9001 reorientó en 2000 el estándar hacia nueve procesos centrales (figura 24.5). Si una organización quiere estar conforme con el estándar ISO 9001, debe documentar cómo se relacionan sus procesos con dichos procesos centrales. También deberá definir y mantener registros que demuestren que se siguieron los procesos organizacionales establecidos. El manual de calidad de la compañía tiene que describir los procesos relevantes y los datos de proceso que deben recopilarse y conservarse.

Para estar de conformidad con ISO 9001, una compañía debe especificar los tipos de proceso que se muestran en la figura 24.5 y tener procedimientos que demuestren que se siguen sus procesos de calidad.

Algunos clientes de software demandan que sus proveedores tengan la certificación ISO 9001. Así, los clientes podrán estar seguros de que la compañía que desarrolla el software tiene un sistema de gestión de calidad aprobado. Autoridades de acreditación independiente examinan los procesos de gestión de calidad y la documentación de proceso, y deciden si dichos procesos abarcan todas las áreas especificadas en ISO 9001. Si es así, certifican que los procesos de calidad de una compañía, definidos en el manual de calidad, concuerdan con el estándar ISO 9001.

Los métodos ágiles, que evitan la documentación y se enfocan en el código a desarrollar, tienen poco en común con los procesos de calidad formal que se examinan en ISO 9001. Se ha hecho cierto trabajo para reconciliar estos enfoques (Stalhane y Hanssen, 2008), pero la comunidad de desarrollo ágil en general se opone a lo que considera una carga burocrática de la conformidad con los estándares. Por esta razón, las compañías que usan métodos de desarrollo ágil se preocupan pocas veces por la certificación ISO 9001.

Revisiones e inspecciones

Las revisiones e inspecciones son actividades QA que comprueban la calidad de los entregables del proyecto. Esto incluye examinar el software, su documentación y los registros del proceso para descubrir errores y omisiones, así como observar que se siguieron los estándares de calidad. Como se estudió en los capítulos 8 y 15, revisiones e inspecciones se usan junto con las pruebas del programa como parte del proceso general de verificación y validación del software.

Durante una revisión, un grupo de personas examinan el software y su documentación asociada en busca de problemas potenciales y la falta de conformidad con los estándares. El equipo de revisión realiza juicios informados sobre el nivel de calidad de un entregable de sistema o de proyecto. Entonces los administradores de proyecto pueden usar dichas valoraciones para tomar decisiones de planeación y asignar recursos al proceso de desarrollo.

Las revisiones de calidad se basan en documentos que se elaboraron durante el proceso de desarrollo del software. Al igual que las especificaciones, el diseño o el código del software, también pueden revisarse los modelos de proceso, planes de prueba, procedimientos de gestión de configuración, estándares de proceso y manuales de usuario. La revisión debe comprobar la coherencia e integridad de los documentos o el código objeto de prueba, y asegurarse de que se han seguido las normas de calidad. Las conclusiones de la revisión deben registrarse formalmente como parte del proceso de gestión de calidad.

El propósito de las revisiones e inspecciones es mejorar la calidad del software, no de valorar el rendimiento de los miembros del equipo de desarrollo. La revisión es un

proceso público de detección de errores, comparado con el proceso más privado de prueba de componentes. Es necesario que los errores cometidos por los individuos se revelen a todo el equipo de programación. Para garantizar que todos los desarrolladores participen constructivamente con el proceso de revisión, los administradores de proyecto tienen que ser sensibles a las preocupaciones individuales. Deben desarrollar una cultura de trabajo que brinde apoyo y no culpar cuando se descubran errores.

El proceso de revisión

1. *Actividades previas a la revisión* Se trata de actividades preparatorias esenciales para que sea efectiva la revisión. Por lo general, las actividades previas a la revisión se ocupan de la planeación y preparación de la revisión. La planeación de la revisión incluye establecer un equipo de revisión, organizar un tiempo, destinar un lugar para la revisión y distribuir los documentos a revisar. Durante la preparación de la revisión, el equipo puede reunirse para obtener un panorama del software a revisar. Miembros del equipo de revisión leen y entienden el software o los documentos y estándares relevantes. Trabajan de manera independiente para encontrar errores, omisiones y distanciamiento de los estándares.
2. *La reunión de revisión* Durante la reunión de revisión un autor del documento o programa a revisar debe repasar el documento con el equipo de revisión. La revisión en sí debe ser relativamente corta, dos horas a lo sumo. Un miembro del equipo debe dirigir la revisión y otro registrar formalmente todas las decisiones y acciones de revisión a tomar.
3. *Actividades posteriores a la revisión* Despues de terminada una reunión de revisión, deben tratarse los conflictos y problemas surgidos durante la revisión. Esto puede implicar corregir bugs de software, refactorizar el software de modo que esté conforme con los estándares de calidad, o reescribir los documentos

Por lo general, el proceso de revisión en el desarrollo de software ágil es informal. En Scrum, por ejemplo, hay una junta de revisión después de completar cada iteración del software (una revisión rápida), en la que pueden exponerse los conflictos y problemas de calidad. En la programación extrema, como se estudiará en la siguiente sección, la programación en grupos de dos personas garantiza que el código se examine y revise constantemente por otro miembro del equipo

Inspecciones del programa

Las inspecciones del programa son “revisiones de pares” en las que los miembros del equipo colaboran para encontrar bugs en el programa en desarrollo

Complementan las pruebas, puesto que no requieren la ejecución del programa

Las inspecciones del programa incluyen a miembros del equipo con diferentes antecedentes que realizan una cuidadosa revisión, línea por línea, del código fuente del programa. Buscan defectos y problemas, y los informan en una reunión de inspección.

Durante una inspección, con frecuencia se usa una lista de verificación de errores comunes de programación para enfocar la búsqueda de bugs. Esta lista de verificación se basa en ejemplos de libros, o bien, en el conocimiento de defectos normales en un dominio de aplicación común

Es posible detectar más del 60% de los errores en un programa mediante inspecciones informales de programación

Los procesos ágiles pocas veces usan procesos de inspección formal o revisión de pares.

En vez de ello, se apoyan en los miembros del equipo que cooperan para comprobar mutuamente el código y en lineamientos informales, tales como “comprobar antes de

ingresar”, lo que sugiere que los programadores deben comprobar su propio código. Los profesionales de la programación extrema argumentan que la programación en parejas es un sustituto efectivo de la inspección, ya que, en efecto, se trata de un proceso de inspección continuo. Dos personas observan cada línea de código y la comprueban antes de aceptarla.

Medición y métricas del software

La medición del software se ocupa de derivar un valor numérico o perfil para un atributo de un componente, sistema o proceso de software. Al comparar dichos valores unos con otros, y con los estándares que se aplican a través de una organización, es posible extraer conclusiones sobre la calidad del software, o valorar la efectividad de los procesos, las herramientas y los métodos de software.

Resumen

- La gestión de calidad del software se ocupa de garantizar que el software tenga un número menor de defectos y que alcance los estándares requeridos de mantenibilidad, fiabilidad, portabilidad, etcétera. Incluye definir estándares para procesos y productos, y establecer procesos para comprobar que se siguieron dichos estándares.
- Los estándares de software son importantes para el aseguramiento de la calidad, pues representan una identificación de las “mejores prácticas”. Al desarrollar el software, los estándares proporcionan un cimiento sólido para diseñar software de buena calidad.
- Es necesario documentar un conjunto de procedimientos de aseguramiento de la calidad en un manual de calidad organizacional. Esto puede basarse en el modelo genérico para un manual de calidad sugerido en el estándar ISO 9001.
- Las revisiones de los entregables del proceso de software incluyen a un equipo de personas que verifican que se siguieron los estándares de calidad. Las revisiones son la técnica usada más ampliamente para valorar la calidad.
- En una inspección de programa o revisión de pares, un reducido equipo comprueba sistemáticamente el código. Ellos leen el código a detalle y buscan posibles errores y omisiones. Entonces los problemas detectados se discuten en una reunión de revisión del código

Principales Modelos de Calidad Existentes (CMMI-SPICE-ISO) y sus métodos de evaluación.

Somerville Cap - 26 Mejora de Procesos

En la actualidad existe una constante demanda de la industria por un mejor y más barato software, que debe entregarse en plazos cada vez más cortos. Por consiguiente, numerosas compañías de software han dirigido la atención hacia la mejora de procesos de software como una forma de aumentar la calidad de su software, reducir sus costos o acelerar los procesos de desarrollo. La mejora de procesos significa comprender los procesos existentes y cambiarlos para incrementar la calidad del producto o reducir los costos y el tiempo de desarrollo.

Se usan dos enfoques muy diferentes para la mejora y el cambio de procesos:

1. El enfoque de madurez de procesos, que se ha orientado en mejorar el proceso y la gestión del proyecto e introducir en una organización buenas prácticas de ingeniería de software. El nivel de madurez del proceso refleja la medida en que se adoptan buenas prácticas técnicas y administrativas en los procesos de desarrollo de software organizacional. Las metas principales de este enfoque consisten en mejorar la

calidad del producto y la previsibilidad del proceso.

2. El enfoque ágil, orientado al desarrollo iterativo y la reducción de las sobrecargas en el proceso de software. Las características primarias de los métodos ágiles son la entrega rápida de funcionalidad y la capacidad de respuesta ante los cambiantes requerimientos del cliente.

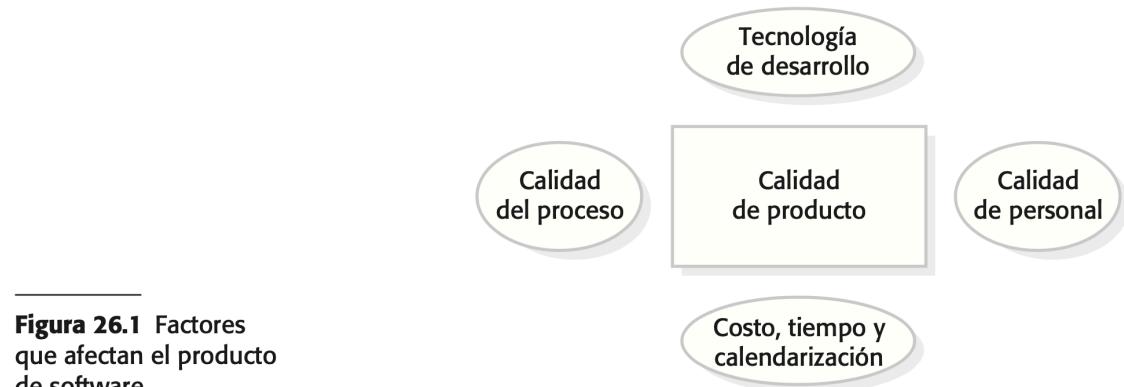


Figura 26.1 Factores que afectan el producto de software

La relación calidad de proceso/calidad de producto es menos evidente cuando el producto es intangible y depende, en alguna medida, de procesos intelectuales que no pueden automatizarse. La calidad del software no está influida por sus procesos de fabricación, sino por su proceso de diseño, en el que las habilidades y la experiencia de la gente son significativas. En ciertos casos, el proceso utilizado puede ser el determinante más significativo de la calidad del producto. Sin embargo, para aplicaciones innovadoras en particular, el personal que interviene en el proceso tiene mayor influencia sobre la calidad que el proceso utilizado.

La influencia de cada uno de estos factores depende del tamaño y tipo del proyecto. Para sistemas muy grandes que incluyen subsistemas separados, desarrollados por equipos que pueden trabajar en diferentes lugares, el factor principal que afecta la calidad del producto es el proceso de software. Los mayores problemas con los proyectos grandes son la integración, la gestión del proyecto y las comunicaciones. Por lo general, existe una mezcla de habilidades y experiencia entre los miembros del equipo y, como el proceso de desarrollo a menudo tiene lugar a lo largo de varios años, el equipo de desarrollo es volátil. Puede cambiar por completo durante la vida del proyecto.

Sin embargo, para proyectos pequeños, en los que sólo existen en el equipo algunos miembros, la calidad del equipo de desarrollo es más importante que el proceso de desarrollo utilizado. Por consiguiente, el manifiesto ágil proclama la importancia de la gente por encima de los procesos. Si el equipo tiene un alto nivel de habilidad y experiencia, es probable que la calidad del producto sea alta, independientemente del proceso utilizado. Si el equipo carece de experiencia y habilidad, un buen proceso puede limitar el daño, pero, en sí mismo, no llevará a un software de alta calidad.

Donde los equipos son reducidos, la buena tecnología de desarrollo es particularmente importante. El equipo pequeño no puede dedicar mucho tiempo a tediosos procedimientos administrativos. Los integrantes del equipo pasan la mayor parte de su

tiempo diseñando y programando el sistema; por lo tanto, las buenas herramientas afectan significativamente su productividad. Para proyectos grandes es esencial un nivel básico de tecnología de desarrollo para la gestión de la información.

Sin considerar los factores de personal, procesos o herramientas, si un proyecto tiene un presupuesto inadecuado o se planea con un calendario de entregas poco realista, la calidad del producto se verá afectada. Un buen proceso requiere de recursos para una implementación efectiva. Si dichos recursos son insuficientes, el proceso no puede ser realmente efectivo. Si los recursos son inadecuados, sólo personal de excelencia puede salvar el proyecto. Aun así, si el déficit es demasiado grande, la calidad del producto se degradará. Al no haber suficiente tiempo para el desarrollo, es probable que el software entregado tenga funcionalidad reducida o niveles más bajos de fiabilidad o rendimiento.

Con demasiada frecuencia, la causa real de los problemas en la calidad del software no es una gestión deficiente, procesos inadecuados o capacitación de escasa calidad. Más bien, es el hecho de que las organizaciones deben competir para sobrevivir. Para ganar un contrato, una compañía puede subestimar el esfuerzo requerido o prometer la entrega rápida de un sistema. Con la intención de cumplir estos compromisos, tal vez acuerde un calendario de desarrollo poco realista. En consecuencia, la calidad del software se ve afectada de manera negativa.

El marco de trabajo para la mejora de procesos CMMI

Modelo de Madurez de Capacidades de Software

Dicho modelo ha influido enormemente para convencer a la comunidad de la ingeniería de software de tomar con seriedad la mejora de procesos

Otras organizaciones han desarrollado también modelos comparables de madurez de proceso. El enfoque SPICE a la valoración de capacidades y la mejora de procesos (Paultk y Konrad, 1994) es más flexible que el modelo SEI. Incluye niveles de madurez comparables con los niveles CMM, pero también identifica procesos, como los procesos cliente-proveedor, que atraviesan dichos niveles. Conforme aumenta el nivel de madurez, también debe mejorar el rendimiento de dichos procesos transversales.

El modelo CMMI (Ahern *et al.*, 2001; Chrissis *et al.*, 2007) tiene la intención de ser un marco para la mejora de procesos con amplia aplicabilidad a través de varias compañías

El modelo CMMI es muy complejo, con más de 1,000 páginas de descripción. Aquí se simplificó principalmente para su análisis. Los principales componentes del modelo son:

1. Un conjunto de áreas de proceso que se relacionan con las actividades de proceso del software
2. Algunas metas, las cuales son descripciones abstractas de un estado deseable que debe lograr una organización
3. Un conjunto de buenas prácticas, las cuales son descripciones de formas para lograr una meta.

Una valoración CMMI implica examinar los procesos en una organización y clasificar dichos procesos o áreas de proceso en una escala de seis puntos que se relacionan con el nivel de madurez en cada área de proceso. La idea es que, cuanto más maduro sea el proceso, mejor será. La siguiente es la escala de seis puntos que asigna un nivel de madurez a un área de proceso:

1. *Incompleto* Al menos no se satisface una de las metas específicas asociadas con el área de proceso. No hay metas genéricas en este nivel, pues no tiene sentido la institucionalización de un proceso incompleto.

2. *Realizado* Las metas asociadas con el área de proceso están satisfechas, y para todos los procesos el alcance del trabajo a realizar se estableció de manera explícita y se comunicó a los miembros del equipo.
3. *Gestionado* En este nivel se satisfacen las metas asociadas con el área de proceso y se establecen políticas organizacionales que determinan cuándo debe usarse cada proceso. Tiene que haber planes de proyecto documentados que establezcan las metas del proyecto. En la institución debe haber procedimientos para la gestión de recursos y la monitorización de procesos.
4. *Definido* Este nivel se enfoca en la estandarización organizacional y el despliegue de procesos. Cada proyecto tiene un proceso gestionado que se adapta a los requerimientos del proyecto desde un conjunto definido de procesos organizacionales. Deben recopilarse activos y mediciones de proceso, además de usarse para futuras mejoras de proceso.
5. *Gestionado cuantitativamente* En este nivel hay una responsabilidad organizacional cuya finalidad es usar métodos estadísticos y cuantitativos para controlar los subprocesos; esto es, deben utilizarse mediciones recopiladas de proceso y producto en la gestión del proceso.
6. *Optimizado* En este nivel superior, la organización debe usar las mediciones de proceso y producto para impulsar la mejora de los procesos. Hay que analizar las tendencias y adaptar los procesos a las necesidades cambiantes de la empresa.

Calidad de Producto: Planificación de pruebas para el software, Niveles y tipos de prueba para el software. Técnicas y herramientas para probar software. Técnicas y herramientas para la realización de revisiones técnicas del software.

Testing de Software (PPT)

Testing en el contexto:

Asegurar la Calidad vs Controlar la Calidad

- Una vez definidos los requerimientos de calidad, tengo que tener en cuenta que:
 - La calidad no puede “inyectarse” al final
 - La calidad del producto depende de tareas realizadas durante todo el proceso
 - Detectar errores en forma temprana ahorra esfuerzos, tiempo, recursos

La calidad no solamente abarca aspectos del producto sino también del proceso y cómo éstos se pueden mejorar, que a su vez evita defectos recurrentes.

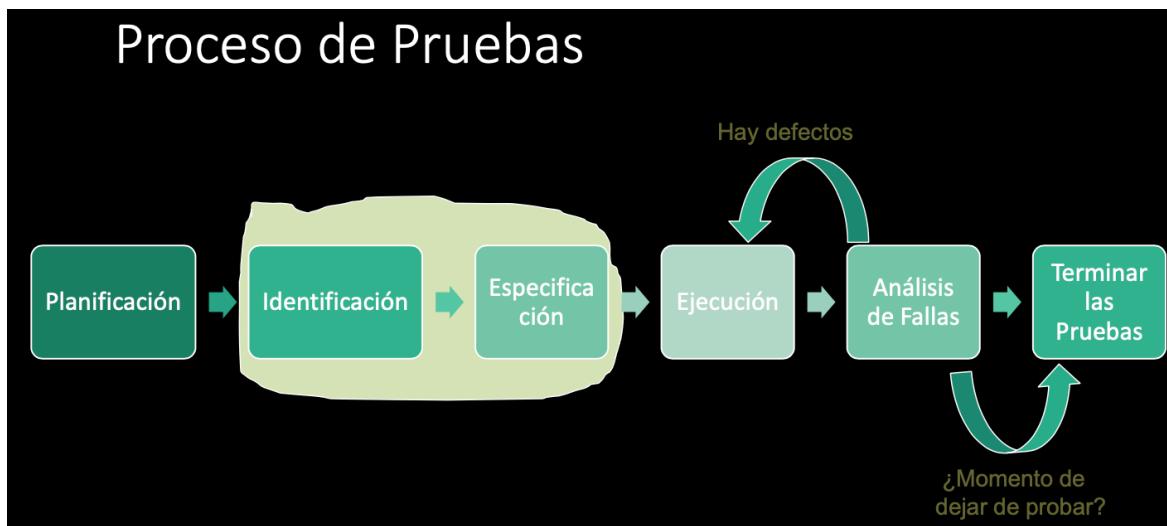
El testing NO puede asegurar ni calidad en el software ni software de calidad

¿Cuánto testing es suficiente?

- El testing exhaustivo es imposible.
- Decidir cuánto testing es suficiente depende de:
 - Evaluación del nivel de riesgo
 - Costos asociados al proyecto
 - Usamos los riesgos para de terminar:
 - Que testear primero
 - A qué dedicarle más esfuerzo de testing
 - Que no testear (por ahora)
 - 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.
 - Puede ser definido en términos de:
 - Costos

- % de tests corridos sin fallas
- Fallas predichas aún permanecen en el software
- No hay defectos de una determinada severidad en el software

Proceso del Testing



Proceso del Testing

Planificación y Control

- La Planificación de las pruebas es la actividad de verificar que se entienden las metas y los objetivos del cliente, las partes interesadas (stakeholders), el proyecto, y los riesgos de las pruebas que se pretende abordar.

Construcción del Test Plan:

- Riesgos y Objetivos del Testing
- Estrategia de Testing
- Recursos
- Criterio de Aceptación

Controlar:

- Revisar los resultados del testing
- Test coverage y criterio de aceptación
- Tomar decisiones

Identificación y Especificación

- Revisión de la base de pruebas
- Verificación de las especificaciones para el software bajo pruebas
- Evaluar la testeabilidad de los requerimientos y el sistema
- Identificar los datos necesarios
- Diseño y priorización de los casos de las pruebas
- Diseño del entorno de prueba

Ejecución

- Desarrollar y dar prioridad a nuestros casos de prueba

Crear los datos de prueba
Automatizar lo que sea necesario
Creación de conjuntos de pruebas de los casos de prueba para la ejecución de la prueba eficientemente.
Implementar y verificar el ambiente.
Ejecutar los casos de prueba
Registrar el resultado de la ejecución de pruebas y registrar la identidad y las versiones del software en las herramientas de pruebas.
Comparar los resultados reales con los resultados esperados.

Evaluación y Reporte

Evaluar los criterios de Aceptación
Reporte de los resultados de las pruebas para los stakeholders.
Recolección de la información de las actividades de prueba completadas para consolidar.
Verificación de los entregables y que los defectos hayan sido corregidos.
Evaluación de cómo resultaron las actividades de testing y se analizan las lecciones aprendidas.

Conceptos: Ciclo de Test

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

Conceptos: Regresión

- Al concluir un ciclo de pruebas, y reemplazarse la versión del sistema sometido al mismo, debe realizarse una verificación total de la nueva versión, a fin de prevenir la introducción de nuevos defectos al intentar solucionar los detectados.

El Testing en el Ciclo de Vida del Software

Objetivos de involucrar las actividades de Testing de manera temprana:

- Dar visibilidad de manera temprana al equipo, de cómo se va a probar el producto.
- Disminuir los costos de correcciones de defectos

The Art of Software Testing - Cap 2-6

Cuando pruebas un programa, quieres agregar mas valor al mismo. Agregar valor mediante el testing significa incrementar la calidad o confiabilidad del programa. Incrementando la confiabilidad del sistema significa encontrar y remover errores.

Entonces, no pruebas un programa para mostrar qué funciona, en cambio, deberías empezar con la asunción que el programa contiene errores y luego probar el programa para encontrar la mayor cantidad de errores posibles.

Si nuestro objetivo es demostrar que el programa no tiene errores, entonces inconscientemente nos dirigiremos hacia ese objetivo, entonces tenderemos a seleccionar los datos para probar que tienen una poca probabilidad de causar una falla en el programa. En la otra mano, si nuestro objetivo es demostrar que el programa tiene errores, nuestros datos de prueba tendrán una probabilidad mas alta de encontrar errores.

El testing es un proceso destructivo

A nuestra forma de pensar, una buena construida y ejecutada prueba de una pieza de software es exitosa cuando encuentra errores que pueden ser reparados. La única prueba que es insatisfactoria es la que no examina apropiadamente el software, y en la mayoría de los casos, la prueba que no encuentra errores será considerada insatisfactoria, ya que el concepto de un programa sin errores es básicamente irrealista.

Dos de las mas prevalentes estrategias incluyen Black-Box testing y White-box testing.

Metodologías/ Estrategias utilizadas para testing

Testing de Caja Negra

Black Box	White Box
Equivalence partitioning	Statement coverage
Boundary-value analysis	Decision coverage
Cause-effect graphing	Condition coverage
Error guessing	Decision-condition coverage
	Multiple-condition coverage

El objetivo de este testing no se preocupa por el comportamiento interno y estructura del programa. En cambio, se concentra en encontrar circunstancias en las cuales el programa no se comporta de acuerdo a sus especificaciones.

Si se quiere usar esta aproximación para encontrar todos los errores en el programa, el criterio es testing exhaustivo de inputs, probando cada posible condición de input como caso de prueba. Para asegurarse de encontrar todos los errores se tienen que probar no solo todos los input validos pero también todos los posibles input.

Metodos:

Basados en Especificaciones

Partición de Equivalencias: Se basa en que si un caso de prueba de una clase de equivalencia detecta un error, todos los otros casos de prueba en la clase de equivalencia deberían encontrar el mismo error. Consiste en identificar las clases de equivalencia validas y no validas. Pueden ser un rango de valores continuos, valores discretos, selección múltiple etc. Luego se identifican los casos de prueba:

- 1) Asignar un numero unico a cada clase de equivalencia
- 2) Hasta que todas las clases de equivalencia validas han sido cubiertas incorporar casos de prueba, escribir un nuevo caso de prueba que cubra la mayor cantidad de clases de equivalencia validas.

3)Hasta que todos los casos de prueba han cubierto las clases de equivalencia

External condition	Valid equivalence classes	Invalid equivalence classes

invalida, escribir un caso de prueba que cubra una.y solo una de las clases de equivalencia invalidas.

Analisis de Valores Limite

Condiciones limite son esas situaciones donde se usa arriba, por debajo y las esquinas de las clases de equivalencia de entrada y de salida. Se usan valores extremos.

Basados en Experiencia

Adivinanza de Errores (Error Guessing)

Dado un programa particular, utilizando la intuición y la experiencia, se piensan distintos tipos de errores probables y se los escribe en casos de prueba para exponer esos errores. La idea básica es enumerar la lista de posibles errores y luego escribir casos de prueba basados en dichos errores.

Testing de caja blanca

Nos permite examinar la estructura interna del programa. Esta estrategia deriva en probar datos desde la examinacion de la lógica del programa.

Lo ideal consiste en causar que se ejecute cada sentencia del programa al menos una vez pero esto resulta imposible, ya que puede haber millones de alternativas.

Statement Coverage:

Consiste en ejecutar cada sentencia del programa al menos una vez. Generalmente es un criterio débil para el testing de caja blanca.

Decision Coverage: Este consiste en que se debe crear la suficiente cantidad de casos de prueba tal que cada decisión tiene una salida true y false al menos una vez.

Condition Coverage: En este caso se debe escribir suficientes casos de prueba para asegurarse que cada condición en una decisión toma todas las posibles salidas al menos una vez.

Decision/Condition Coverage:

Cada condición en una decisión toma todas las posibles salidas al menos una vez, y cada decisión toma todas las posibles salidas al menos una vez, y cada punto de entrada es invocado al menos una vez.

Multiple-Condition Coverage:

Todas las posibles combinaciones de las salidas de una condición en cada decision, y todos los puntos de entrada son invocados al menos una vez.

Principios del Testing de Software

- 1 - Una parte necesaria del caso de prueba es la definición de la salida o resultado esperado.
- 2 - Un programador debería evitar probar su propio código.
- 3 - Una organización debería evitar probar sus propios programas.
- 4 - Inspeccionar completamente los resultados de cada prueba
- 5 - Los casos de prueba deberían estar escritos para condiciones de entrada que son inválidas y no esperadas, así también como las cuales son válidas y esperadas.
- 6 - Examinar un programa para ver si no hace lo que debería es la mitad de la batalla, la otra es ver si el programa hace lo que no se supone que haga.
- 7 - Evitar desechar casos de pruebas a menos que el programa sea realmente desecharable.
- 8 - No planear un esfuerzo de prueba bajo la suposición que no se encontrarán problemas.
- 9 - La probabilidad de existencia de más errores en una sección de un programa es proporcional al número de errores ya encontrados en esa sección.
- 10 - Testing es una tarea creativa e intelectualmente desafiante.

Inspecciones técnicas, Walkthroughs y Revisiones

Otra de las formas de probar software son con pruebas no basadas en computadoras (Human Testing). Las técnicas de Human Testing son bastante efectivas en encontrar errores, tanto que cada proyecto de programación debería utilizar una o más de estas técnicas. Estos métodos deberían ser aplicados entre el tiempo que el programa es codeado y el tiempo que el testing basado en computadora comienza.

Primero, es generalmente reconocido que lo más temprano que los errores son encontrados, más bajos eran los costos de corrección y más alta la probabilidad de corregirlos correctamente. Segundo, los programadores tienden a experimentar un cambio psicológico cuando comienza un testing basado en computadora.

Inspecciones y Walkthroughs

Los dos métodos primarios de testing humano son las inspecciones y los walkthroughs. Estas incluyen un equipo de personas leyendo o inspeccionando visualmente un programa. Con cualquier método, los participantes deben conducir un trabajo de preparación. El climax es una reunión. El objetivo de esta es encontrar errores pero no encontrar soluciones a los mismos. Es decir probar pero no hacer debugging.

En un walkthrough un grupo de desarrolladores con tres o cuatro miembros es el número óptimo para llevar a cabo la revisión. Uno de los participantes es el autor del programa. Entonces la mayoría de la prueba del programa es conducida por el resto, quienes siguen el principio que dice que es inefectivo cuando uno prueba su propio código.

Otra ventaja de los walkthroughs, resultando en un costo menor de debugging, es el hecho que cuando un error es encontrado generalmente es localizado precisamente en el código. Estos métodos generalmente son efectivos encontrando desde el 30 al 70% de los errores de diseño lógico y errores de código típicos. Sin embargo no son efectivos en detectar

errores de diseño de alto nivel, como los errores hechos en procesos de análisis o requerimientos.

Por supuesto que una critica posible de estas estadísticas es que los procesos humanos solo encuentran los errores “fáciles” y los difíciles solo pueden ser encontrados por los testing basados en computadoras.

La conclusion es que las inspecciones/walkthroughs y testing basado en computadores son complementarios, la eficiencia de la detección de errores sufrirá si una o otra no esta presente.

Inspecciones de Código

Una inspección de código es un conjunto de procedimientos y técnicas de detección de errores para lectura de código en grupo.

El equipo de inspección generalmente consiste de cuatro personas. One de las cuatro juega el rol de Moderador. Se espera que el moderador sea un programador competente, pero que el no es el autor del código y no tiene que saber nada sobre los detalles del programa.

Las tareas del moderador incluyen:

- Distribuir los materiales para organizar la sesión de inspección
- Liderar la sesión
- Registrar todos los errores encontrados
- Asegurarse que todos los errores son corregidos

El segundo miembro del equipo es el programador. El resto del equipo generalmente son diseñadores del programa y un especialista en testing.

El moderador distribuye la lista del programa y las especificaciones de diseño a los otros participantes varios días antes para la sesión de inspección. Se espera que los los participantes se familiaricen con el material previamente a la sesion. Durante la sesión dos actividades ocurren:

- 1) Los programadores narran, linea por linea, la lógica del programa. Durante el discurso, los otros participantes pueden hacer preguntas y deberían intentar determinar si existen errores.
- 2) El programa es analizado con respecto a una check List de errores comunes de programación.

El moderador es responsable de asegurarse que las discusiones proceden de manera productiva y que los participantes colocan su atención en encontrar errores, no encontrarlos.

Luego de la sesión al programador se le entrega una lista con los errores encontrados. Como se dijo previamente, el proceso de inspección generalmente se concentra en encontrar errores, no corregirlos.

La inspección también tiene varios efectos beneficiosos en adición a su efecto principal de encontrar errores. Por un lado, él programador generalmente recibe feedback sobre su estilo de programación, su elección de algoritmos, y técnicas de programación. Por otro lado los participantes también ganan en una forma similar al ser expuestos a los errores y el estilo de programación de otro programador. Finalmente, el proceso de inspección es una forma de identificar de manera temprana las secciones mas susceptibles a errores del

programa, ayudando a concentrar más la atención en esos sectores durante el testing basado en computadora.

Walkthroughs

Tiene mucho en común con el proceso de inspección, pero los procedimientos son un poco diferentes, una técnica de detección de errores diferente es empleada.

Como la inspección, el walkthrough es una reunión ininterrumpida de una o dos horas de duración. El equipo consiste de 3 a 5 personas. Una de estas cumple un rol similar al del moderador, otra el rol de secretario (el que registra todos los errores encontrados) y una tercera persona cumple el rol de tester. Las sugerencias para el resto de las personas suele variar. Por supuesto el programador es una de ellas. Otras sugerencias son: un programador experimentado, un experto en lenguaje de programación, la persona que generalmente mantiene el sistema, alguien de otro proyecto, etc.

El procedimiento inicial es idéntico al de inspección: Se les entrega a los participantes el material varios días antes para permitirles que se familiaricen con el programa. Sin embargo los procedimientos de la reunión son diferentes. En vez de leer el programa simplemente o usar check List de errores, los participantes hacen de computadora. La persona designada como el tester llega a la reunión con un conjunto de casos de pruebas en papel, con conjuntos de entradas representativas y salidas para el módulo o programa. Durante la reunión, cada caso de prueba es ejecutado mentalmente. Así es, los datos de prueba son llevados a lo largo de la lógica del programa. El estado del programa es monitoreado en papel o una pizarra.

Desk Checking

Otro proceso de detección de errores humano es la antigua práctica del desk checking. Esta puede ser vista como un walk through de una persona. La persona lee el programa, y la chekea con respecto a una lista de errores, o prueba datos de prueba en él. El problema principal de desk checking es que no cumple con uno de los principios del testing, el que dice que el código no debe ser probado por uno mismo.

Niveles de Testing:

Testing Unitario

Es el proceso de testar los subprogramas individuales, subrutinas, o procedimientos en un programa. En vez de probar el sistema como un todo, primero se concentra en pequeños bloques del sistema. Los motivos para hacer son:

- 1) El testing unitario es una forma de administrar los elementos combinados del testing, ya que la atención está enfocada inicialmente en pequeñas unidades del programa.
- 2) Facilita la tarea del debugging, ya que cuando un error es encontrado se sabe que existe en un modelo particular.
- 3) Introduce paralelismo al proceso del testing del programa presentándonos una oportunidad de probar múltiples modelos simultáneamente.

Diseño de Caso de Prueba para Testing Unitario

Se necesitan dos tipos de información cuando se diseña un caso de prueba para una prueba unitaria: una especificación del módulo y el código del mismo. La especificación típicamente define los parámetros de entrada y salida del módulo como una función. El testing unitario es orientado a testing de caja blanca.

Testing de Integración (Incremental Testing)

Test orientado a verificar que las partes de un sistema que funcionan bien aisladamente, también lo hacen en conjunto.

Cualquier estrategia de prueba de versión o de integración debe ser incremental, para lo que existen dos esquemas principales:

- Top-Down

Empieza con el modulo inicial o el que se encuentra mas arriba en el programa y se van integrando los que se encuentran debajo.

- Bottom-Up

Comienza con el modulo que se encuentra mas abajo ese realiza hacia arriba la integración.

Lo ideal es una combinación de ambos esquemas.

Tener en cuenta que los módulos críticos deben ser probados lo más tempranamente posible.

Los puntos clave del test de integración son simples:

- Conectar de a poco las partes más complejas
- Minimizar la necesidad de programas auxiliares

Testing de Sistema (PPT)

Es la prueba realizada cuando una aplicación esta funcionando como un todo (Prueba de la construcción Final).

- Trata de determinar si el sistema en su globalidad opera satisfactoriamente (recuperación de fallas, seguridad y protección, stress, performance, etc.)
- El entorno de prueba debe corresponder al entorno de producción tanto como sea posible para reducir al mínimo el riesgo de incidentes debidos al ambiente específicamente y que no se encontraron en las pruebas.
- Deben investigar tanto requerimientos funcionales y no funcionales del sistema.

(Segun Art of Software Testing) Se divide en dos partes

Function Testing: Es un proceso que intenta encontrar las discrepancias entre el programa y la especificación externa. La especificación externa es una descripción precisa del comportamiento del programa desde el punto de vista del usuario final.

System Testing: No es un proceso de probar las funciones del sistema completo, porque esto seria redundante con el función testing. El testing de sistema tiene un propósito particular: comparar el sistema con sus objetivos originales, dado este propósito surgen las siguientes implicaciones:

1. System testing no esta limitado a los sistemas. Si el producto es un programa, este tipo de testing debe demostrar como el programa como un todo no cumple sus requerimientos.

2. Este tipo de testing es imposible si no hay un conjunto de objetivos escritos y medibles para el programa.

Testing de Aceptación

Es el proceso de comparar el programa a sus requerimientos iniciales y las necesidades actuales de los usuarios finales. Es un tipo inusual de prueba en el que generalmente es llevado a cabo por el cliente o un usuario final.

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

La meta en 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 en las pruebas de aceptación.

- Comprende tanto la prueba realizada por el usuario en ambiente de laboratorio (pruebas alfa), como la prueba en ambientes de trabajo reales (pruebas beta).

Tipos de Pruebas

Testing Funcional

- Las pruebas se basan en funciones y características (descripta en los documentos o entendidas por los testers) y su interoperabilidad con sistemas específicos
 - Basado en Requerimientos
 - Basado en los procesos de negocio

Testing No Funcional

- Es la prueba de “cómo” funciona el sistema

Performance Testing

Pruebas de Carga

Pruebas de Stress

Pruebas de usabilidad

Pruebas de mantenimiento

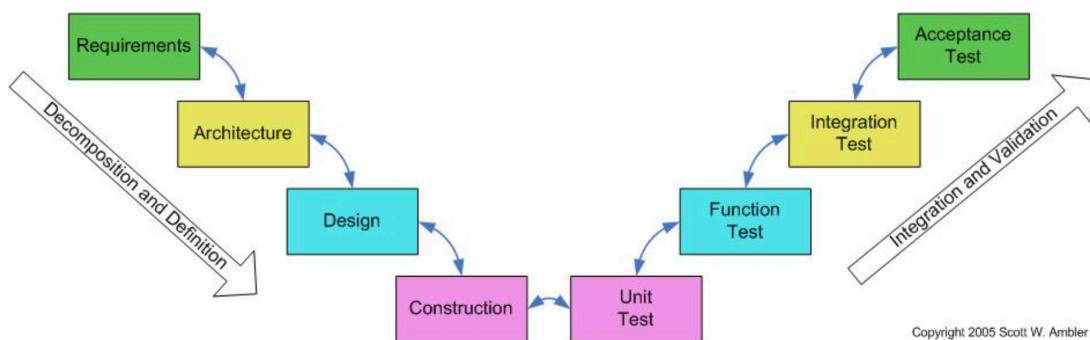
Pruebas de fiabilidad

Pruebas de portabilidad

Testing en Ambientes Agiles

Agile testing and Quality strategies - Paper

El sistema tradicional de desarrollo



Problemas:

-Arreglar defectos es mas caro: Mientras mas largo sea el ciclo de feedback mas grande sera el costo de reparar un defecto encontrado. El modelo en V promociona ciclos de feedback muy largos.

-Tiempo incrementado para el valor: El modelo en V alarga el tiempo que lleva entregar funcionalidad en producción.

Como agil es diferente?

-Mejor Colaboración: Los desarrolladores ágiles trabajan juntos, apoyando la comunicación directa sobre el pasaje de documentación.

-Ciclo de Feedback mas corto: El tiempo entre especificar un requerimiento en detalle y validar ese requerimiento puede ser ahora en el orden de los minutos.

TDD (Test-Driven Development)

Es una técnica de desarrollo ágil que combina:

-Refactorización: Es una técnica en la que se hace un pequeño cambio al código existente para improvisar el diseño sin cambiar las semánticas.

-TFD(Test first Development): Se escribe una prueba y se escribe suficiente código para probar.

Agile Testing Overview

Principles

Testing Moves the Project Forward

On traditional projects, testing is usually treated as a quality gate, and the QA/Test group often serves as the quality gatekeeper. It's considered the responsibility of testing to prevent bad software from going out to the field. The result of this approach is long, drawn out bug scrub meetings in which we argue about the priority of the bugs found in test and whether or not they are sufficiently important and/or severe to delay a release.

On Agile teams, we build the product well from the beginning, using testing to provide feedback on an ongoing basis about how well the emerging product is meeting the business needs.

Testing is NOT a Phase...

...on Agile teams, testing is a way of life.

Agile teams test continuously. It's the only way to be sure that the features implemented during a given iteration or sprint are actually done.

Continuous testing is the only way to ensure continuous progress.

Everyone Tests

On traditional projects, the independent testers are responsible for all test activities. In Agile, getting the testing done is the responsibility of the whole team. Yes, testers execute tests.

Developers do too.

Shortening Feedback Loops

How long does the team have to wait for information about how the software is behaving?

Measure the time between when a programmer writes a line of code and when someone or something executes that code and provides information about how it behaves. That's a feedback loop.

Shorter feedback loops increase Agility. Fortunately, on Agile projects the software is ready to test almost from the beginning. And Agile teams typically employ several levels of testing to uncover different types of information.

Lightweight Documentation

Instead of writing verbose, comprehensive test documentation, Agile testers:

- ! Use reusable checklists to suggest tests

- ! Focus on the essence of the test rather than the incidental details
- ! Use lightweight documentation styles/tools
- ! Capturing test ideas in charters for Exploratory Testing
- ! Leverage documents for multiple purpose

“Done Done,” Not Just Done