Ingeniería de Software 3

Alumno: Santiago Vietto

Docente: Fernando Bono

Institución: UCC

Año: 2022

Herramientas de control de versiones

Introducción

Terminología

Item de configuración o item de configuración de software (SCI, por las siglas de Software Configuration Item): 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 items 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 item de configuración más un número de versión.

Línea base (baseline): 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 (codeline): es un conjunto de versiones de un componente de software y otros ítems de configuración de los cuales depende dicho componente.

Línea principal (mainline): es una secuencia de líneas base que representa diferentes versiones de un sistema.

Entrega, liberación (release): es una entrega de un sistema que se libera para su uso a los clientes (u otros usuarios en una organización).

Espacio de trabajo (workspace): área de trabajo privada donde puede modificarse el software sin afectar a otros desarrolladores que estén usando o modificando dicho software.

Ramificación (branching): es 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 (merging): es 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: es la creación de una versión ejecutable del sistema al compilar y vincular las versiones adecuadas de los componentes y las librerías que constituyen el sistema.

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. 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.

Administración de la configuración (CM - Configuration Management)

_ CM 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.

_ Es esencial para los proyectos de equipo en los que muchos desarrolladores trabajan al mismo tiempo en un sistema de software. CM define como una organización construye y libera (builds and releases) sus productos, identifica y gestiona los cambios.

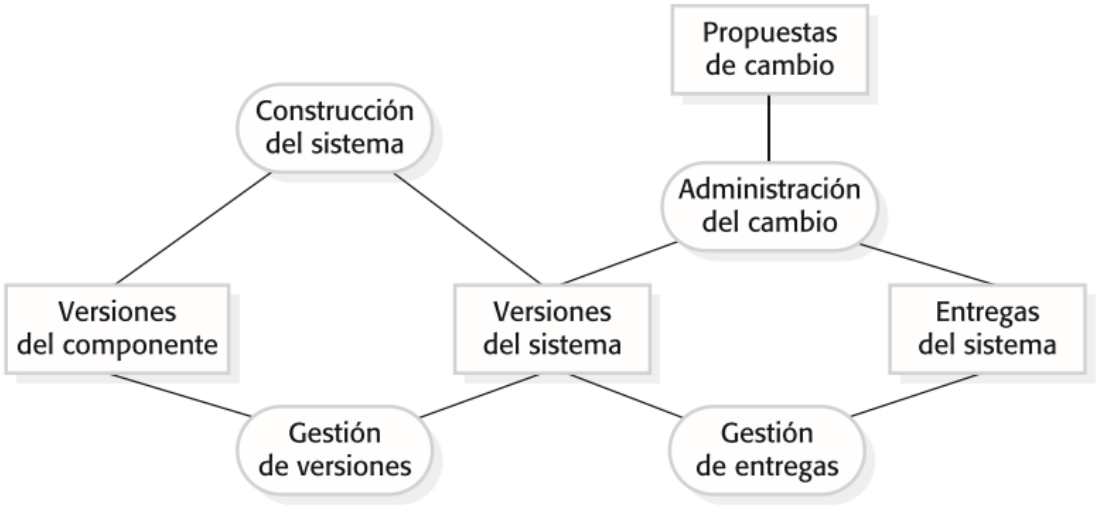
_ La administración de la configuración de un producto de sistema de software comprende cuatro actividades estrechamente relacionadas:

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 (build): es el proceso de ensamblar los componentes del programa, datos y librerías, 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.

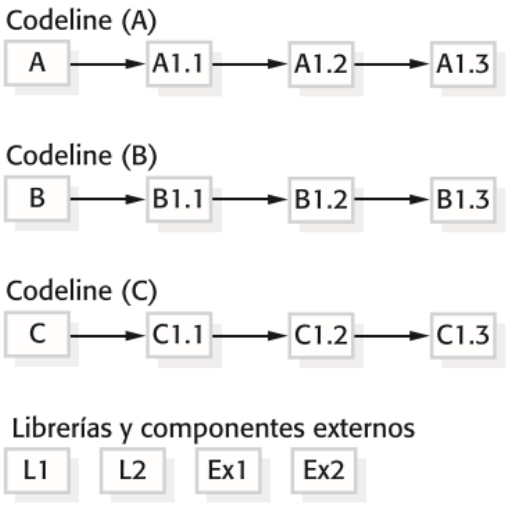


Gestión de versiones

Gestión de versiones (VM - Version Management)

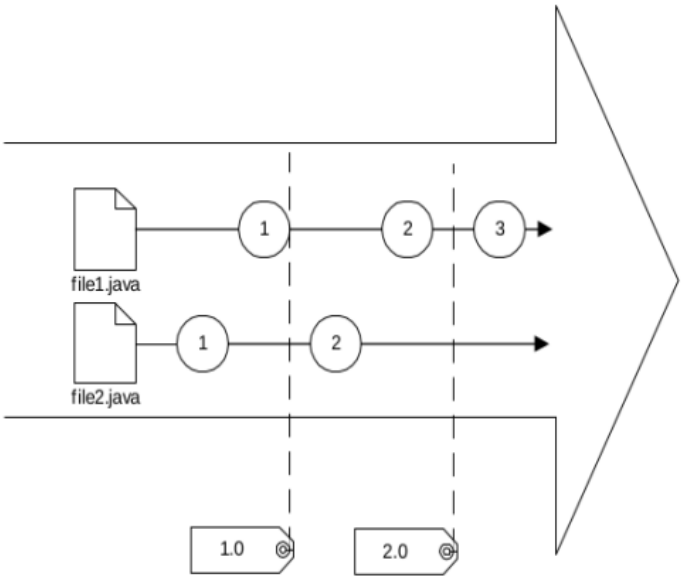
_ VM es el proceso de hacer un seguimiento de las diferentes versiones de los componentes de software o ítems de configuración, y los sistemas donde se usan dichos componentes. Por ende, es el proceso por el cual se gestionan los codelines y las baselines:

Codeline: es una secuencia de versiones de código fuente con las versiones más recientes en la secuencia derivadas de las versiones anteriores. Se aplican regularmente a componentes de sistemas, de manera que existen diferentes versiones de cada componente. Un codeline contiene todas las revisiones de todos los items a lo largo de un camino evolutivo de un componente. Además, es un conjunto de archivos fuente y otros ítems de configuración que conforman un componente de software a lo largo del tiempo mientras cambian. Cuando se cambia un item se crea una nueva revisión de ese ítem. Un producto puede ser construido a partir de un solo codeline o más de uno.

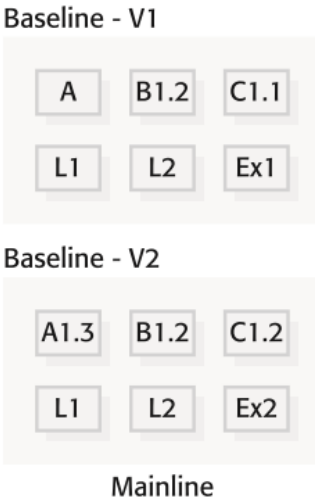


- Snapshot: contine una revisión específica de cada ítem en un codeline. Un snapshot de un codeline se dice que es una versión.
- Versión: se puede identificar con una etiqueta.

_ A continuación, vemos un ejemplo de un codeline compuesto por dos archivos de configuración (file1 y file2). Cada vez que hacemos commit, check in o guardamos dichos archivos en el sistema de control de versiones, creamos una nueva versión del archivo (1, 2, etc). Esto es “in codeline”, es decir, un conjunto de todas las versiones de un ítem de configuración mientras cambia a lo largo del tiempo.



Baseline: es la definición de un sistema especifico. Por consiguiente, un baseline especifica las versiones del componente que se incluyen en el sistema más una especificación de las librerías usadas, archivos de configuración, etc. Sirven para recrear una versión específica de un sistema, por ejemplo, para diferentes clientes, cuando se reporta un bug o se pide una nueva funcionalidad. No deben cambiar, ya que sirven como base para construir nuevas versiones.



Herramientas de versionado

Sistema de control de versiones (VCS - Version Control System)

_ Son sistemas que identifican, almacenan y controlan el acceso a las diferentes versiones de los items de configuración. Estos son una categoría de herramientas de software que ayudan a un equipo de software a gestionar los cambios en el código fuente a lo largo del tiempo. El software de control de versiones realiza un seguimiento de todas las modificaciones en el código. Si se comete un error, los desarrolladores pueden ir atrás en el tiempo y comparar las versiones anteriores del código para ayudar a resolver el error al tiempo que se minimizan las interrupciones para todos los miembros del equipo.

Funcionalidades:

- Identificación de Versiones y Releases: crea versiones de los items cuando estos se envían al sistema.
- Registro del historial de cambios: todos los cambios realizados al código de un sistema o componente se registran y enumeran. En algunos sistemas, dichos cambios pueden usarse para seleccionar la versión de un sistema en particular. Esto implica etiquetar componentes con palabras clave que describan los cambios realizados. Entonces se pueden usar dichas etiquetas (tags) para seleccionar los componentes a incluir en una línea base.
- Soporte para el desarrollo Independiente: es posible que diferentes desarrolladores trabajen en el mismo componente al mismo tiempo. El sistema de gestión de versiones hace un seguimiento de los componentes que se marcaron para la edición y se asegura de que no interfieran los cambios hechos a un componente por diferentes desarrolladores.
- Soporte de proyecto: permite hacer check in y check out de grupos de ítems.
- Gestión del almacenamiento: para reducir el espacio de almacenamiento requerido por múltiples versiones de los componentes que difieren sólo ligeramente, los sistemas de gestión de versiones ofrecen, por lo general, facilidades de 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.

Ventajas: el control de versiones permite que los desarrolladores se muevan más rápido y posibilita que los equipos de software mantengan la eficacia y la agilidad a medida que el equipo se escala para incluir más desarrolladores. Los sistemas de control de versiones han experimentado grandes mejoras en las últimas décadas y algunos son mejores que otros. Una de las herramientas de los VCS más populares hoy en día se llama Git, un VCS distribuido, una categoría conocida como DVCS. Las principales ventajas que deberías esperar del control de versiones son las siguientes:

- Historial completo de cambios a largo plazo de todos los archivos: los cambios incluyen la creación y la eliminación de los archivos, así como los cambios de

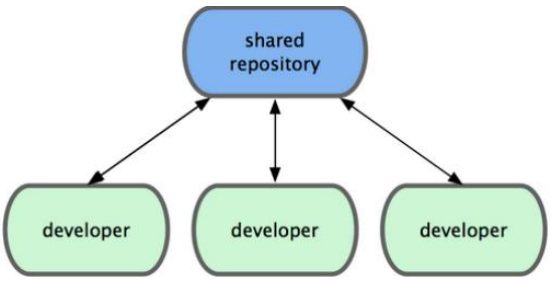
sus contenidos. Este historial también debería incluir el autor, la fecha y notas escritas sobre el propósito de cada cambio. Tener el historial completo permite volver a las versiones anteriores para ayudar a analizar la causa raíz de los errores y es crucial cuando se tiene que solucionar problemas en las versiones anteriores del software.

- Creación de ramas y fusiones (merge): la creación de una “rama” o “Branch” en las herramientas de VCS mantiene múltiples flujos de trabajo independientes los unos de los otros al tiempo que ofrece la facilidad de volver a fusionar ese trabajo, lo que permite que los desarrolladores verifiquen que los cambios de cada rama no entran en conflicto.
- Trazabilidad: ser capaz de trazar cada cambio que se hace en el software y conectarlo con un software de gestión de proyectos y seguimiento de errores, además de ser capaz de anotar cada cambio con un mensaje que describa el propósito y el objetivo del cambio, no solo te ayuda con el análisis de la causa raíz y la recopilación de información.

_ Aunque se puede desarrollar software sin utilizar ningún control de versiones, hacerlo somete al proyecto a un gran riesgo que ningún equipo profesional debería aceptar.

Conceptos

Repositorio: tenemos el ejemplo de un proyecto que será desarrollado por un equipo de desarrolladores sobre un mismo módulo, y acá surge el concepto de repositorio, ya que es el lugar donde están guardados los ítems de configuración y sus respectivas versiones. Luego, cada desarrollador podrá sacar una copia para llevar a su área de trabajo privada (workspace), y una vez concluido su cambio, será subido al espacio común (repositorio).



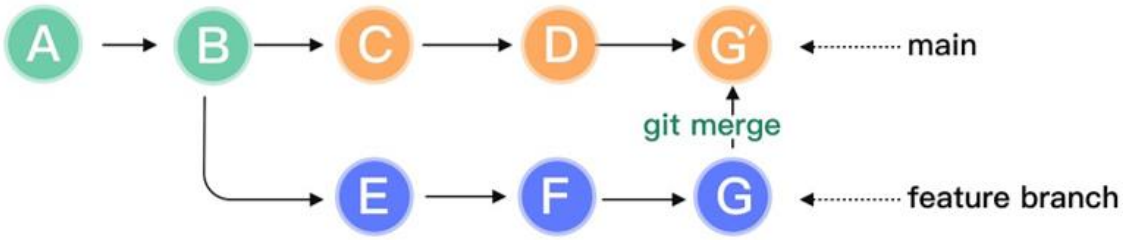
Workspace: es un área de trabajo privada, donde el desarrollador mantiene todos los ítems de configuración que necesita para realizar una tarea. Generalmente es un directorio en el disco. Este espacio contiene versiones específicas de los items (aunque depende de si el VCS es centralizado o distribuido). Además, debería contener un mecanismo para construir ejecutables a partir de su contenido, como Source code, source code for tests, libraries, scripts to build. Y también puede ser manejado en el contexto de una IDE.

Branch (rama): cuando comenzamos en el repositorio, tenemos una sola branch “master”, que es el codeline principal (mainline). Muchas veces, en lugar de tener una sola secuencia de versiones que reflejen los cambios a un componente en el tiempo, suele haber varias secuencias independientes (varias branches). Esto es normal durante el desarrollo del sistema, en dónde diferentes usuarios trabajan de manera independiente en diferentes versiones del código, a través de las branches creadas, y por lo tanto, el mismo cambia de manera diferente.

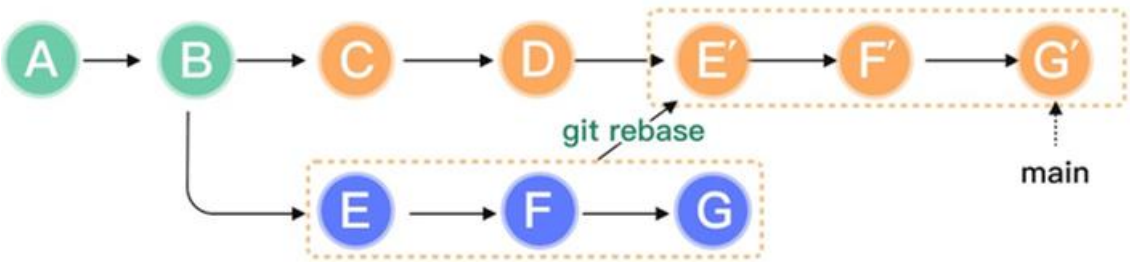
_ También puede suceder que una versión del componente esté en operación, mientras que otra versión esté en desarrollo. En este caso, puede ser necesario fixear bugs críticos en la versión en operación antes de que se entrega nueva versión.

Merge: en algún momento, puede ser necesario crear una nueva versión de un componente, que incluya todos los cambios que se hayan hecho de manera independiente (en las distintas branches), y para esto realizamos una operación merge en donde se integran todas las versiones en una sola rama. Si los cambios hechos involucran diferentes partes del código, las versiones de los componentes pueden mergearse de manera automática combinando los deltas que aplican a cada código. También puede ser necesaria la intervención de un desarrollador para resolver conflictos de merge.

_ En este ejemplo, a través de merge, incorporamos los últimos cambios realizados en la rama feature (feature branch) a la rama principal (main branch). También se puede realizar en el caso inverso. Básicamente es como intentar unir las dos ramas con un nudo.



Rebase (reorganización): es el proceso de mover o combinar una secuencia de commits en un nuevo commit base. Consiste en cambiar la base de un commit a otra, haciendo que parezca que hemos creado la rama o branch, desde un commit diferente. En este ejemplo, rebase cambia la base de nuestra feature branch, a la última confirmación o commit en la main branch, y luego reproduce nuestros cambios desde allí. Esto proporciona un historial de confirmaciones limpio, sencillo y lineal.

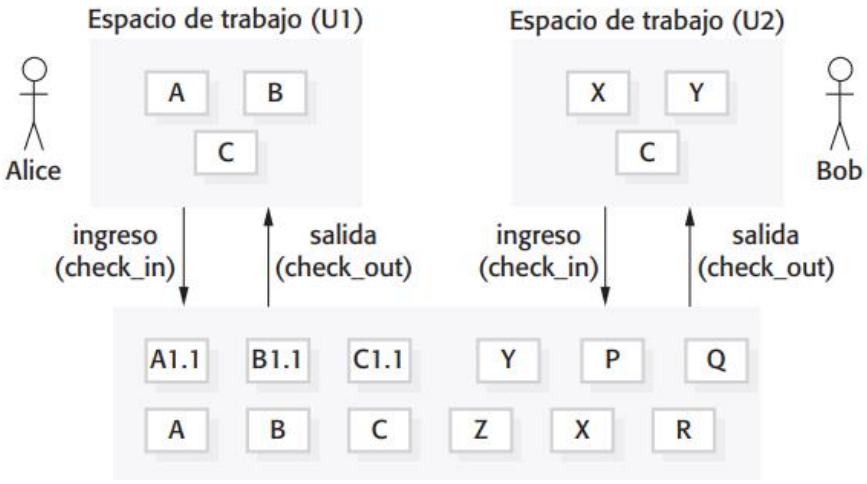


Características de los VCS

_ Según la arquitectura:

Centralizado: en este caso, existe un único repositorio central que contiene todos los ítems de configuración y su historia. Si queremos trabajar sobre este, debemos sacar una copia de una versión específica, trabajar en nuestro espacio de trabajo privado, y cuando terminamos hacemos un check in. Dicho de otra forma, cuenta con un solo punto de error, que es la instancia remota del VCS central. Si se pierde dicha instancia, puede producir la pérdida de datos y de la productividad, y se deberá sustituir por otra copia del código fuente (pero la historia se pierde). Asimismo, si se vuelve temporalmente no disponible, evitará que los desarrolladores envíen, fusionen o reviertan código.

- Desventajas:
  - Hay un unico punto de fallo: si el repositorio falla, y no se tiene backup, detiene el trabajo.
  - Operación solo en modo conectado: si estoy offline solo puedo trabajar en mi workspace, pero no puedo hacer check in ni check out de una versión que necesite, ni comparar la historia.
  - En la primera generación, solo operaban de manera local.
  - En la segunda generación, se soporta modelo cliente-servidor.
- Ejemplo: Subversion.

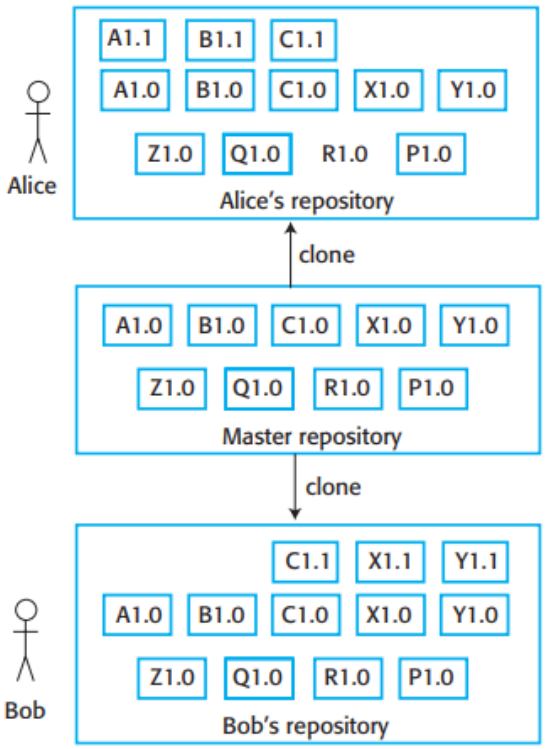


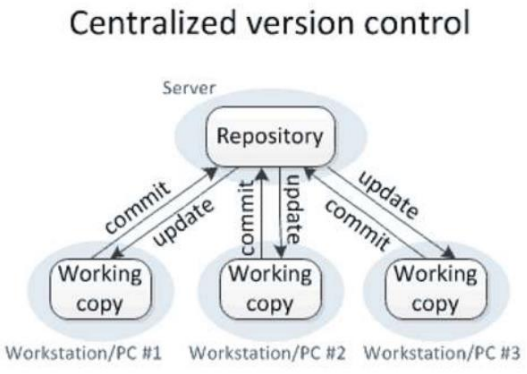
Distribuido: existen copias del mismo repositorio en varias máquinas, trabajo sobre distintas copias de este y después los combino. De otra forma, se mantiene una copia total del código fuente en cada instancia de VCS. Si en el modelo distribuido se produce cualquiera de los casos de error centralizados antes mencionados, se puede designar una instancia del VCS como principal mitigando cualquier caída grave de productividad, ya que todas las instancias son clones completos entre sí.

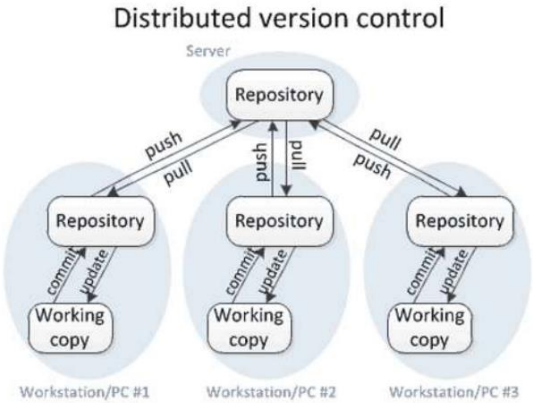
_ Los pasos son los siguientes:

1. El desarrollador crea un clon del repositorio principal, instalado en la máquina del desarrollador de manera local.
2. Luego, trabaja en los archivos y mantienen las nuevas versiones en su repositorio privado en su computadora.
3. Hace un commit de los cambios y actualiza su repositorio privado.
4. Finalmente hace un push de los cambios a un repositorio remoto y, por ejemplo, se sincroniza con el servidor principal del equipo.

- Ventajas:
  - Tenemos varias copias del repositorio, ideal como backup por si surge un inconveniente. Una copia principal en el servidor, que mantiene el código producido por el equipo de desarrollo.
  - Nos permite trabajar de forma offline.
- Ejemplo: Git.







_ Según la atomicidad, y hace referencia a si el sistema de control de versiones permite hacer un commit por un conjunto de archivos como una sola cosa, de manera atómica, o si trabaja por archivos individuales.

Operaciones por archivo: en las primeras generaciones, cada file tiene su propio master de cambios.

- Desventajas:
  - Cambios que afectan varios files al mismo tiempo no se guardan de manera atómica. para resolver un bug necesito buscar la versión de cada uno de los archivos que fueron modificados para solucionarlo.
  - Los comentarios no se pueden asociar a un conjunto de archivos.

Operaciones por conjunto de archivos: aquellos cambios que requieren la modificación de varios archivos se tratan como unidad. Se tiene una historia y comentarios asociados a un cambio y no a archivos individuales. Es importante para migrar cambios o retirarlos.

_ Según la resolución de conflictos, es decir, cómo resuelven conflictos cuando más de un desarrollador modifican el mismo archivo:

Locking: en los primeros VCS centralizados, uno podía sacar los archivos del repositorio para verlos de manera “read only”, pero en el momento que quisiéramos modificarlo se solicitaba al VCS lo que se llama un lock. Solamente la persona que tenía ese lock podía tener una copia para escritura. No podía haber más de una persona con ese lock al mismo tiempo. Cuando se hace check-in se liberaba. El flujo esperado es:

- Develop1 adquiere el lock de un archive foo.c y comienza a modificarlo.
- Developer2, trata de modificar el archive foo.c. Es notificado que developer1 tiene el lock del archivo y no puede hacerle check out.
- Developer2 está bloqueado, no puede continuar y se va a tomar un café.
- Develper1 termina los cambios en foo.c, hace check-in y libera el lock en foo.c.
- Developer2 regresa de tomar el café, hace check out de foo.c y adquiere el lock.

Merge-before-commit: si hay dos personas trabajando en la misma versión, al mismo tiempo, el VCS lo permite, pero al momento de hacer un commit verifica que la versión base de los cambios sea la misma versión que está en el repositorio. Avisa que el archivo ha sido modificado desde el check out y hay que mergear los cambios antes de poder hacer commit. El flujo esperado es:

- Developer1 hace check out del archivo foo.c y comienza a modificarlo.
- Developer2 hace check out del archivo foo.c y comienza a modificarlo.
- Developer1 termina los cambios y hace check in.
- Developer2 termina los cambios y cuando intenta hacer check in, el VCS le dice que la versión en el repositorio ha cambiado y que debe resolver los conflictos antes de proceder.

- Developer2 mergea los cambios que hizo con los cambios que hizo developer1 y que ya están en el repositorio.
- Los cambios de Developer1 y de Developer2 no se superponen.
- El merge es exitoso, y el VCS le permite a Developer2 hacer commit de la versión mergeada.

Commit-before-merge: en este caso el VCS nunca bloquea un commit. Si la versión del repositorio ha cambiado desde el check out se crea una nueva branch y, si luego se desea, permite mergearlos. En el repositorio queda registro que hubo dos commits al mismo tiempo sobre la misma versión. Esto permite llevar un desarrollo fluido con un historial sobre qué está pasando durante este proceso. También da lugar a la experimentación por la disponibilidad de los branches y la posibilidad de mergear cuando se crea conveniente.

Generaciones de VCS

_ Todas las características de los VCS que vimos antes dieron lugar a lo que llamamos generaciones de sistemas de control de versiones.

Generation	Networking	Operations	Concurrency	Examples
First	None	One file at a time	Locks	RCS, SCCS
Second	Centralized	Multi-file	Merge before commit	CVS, SourceSafe, Subversion, Team Foundation Server
Third	Distributed	Changesets	Commit before merge	Bazaar, Git, Mercurial

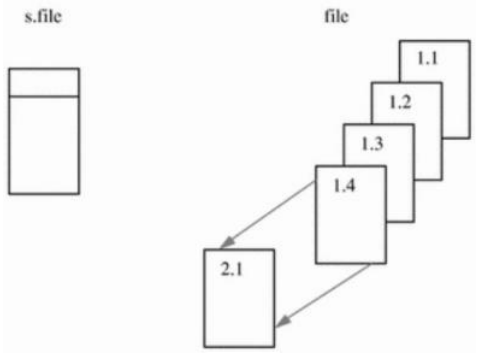
SCCS: en 1970 se inventó el concepto de control de versiones, llamado “Interleaved deltas” como solución al problema de espacio de copias. El algoritmo guardaba cada una de las versiones y reconstruía los archivos a partir de los deltas y la versión base.

_ Ventajas:

- Podía reconstruir cualquier versión a partir de esos deltas en un tiempo proporcional al tamaño del archivo.
- Más rápido para acceder a versiones más recientes.
- Interfaz de comando mejorada: inventó la terminología actual.

_ Desventaja:

- Deltas por archivo y no por conjunto de archivos.



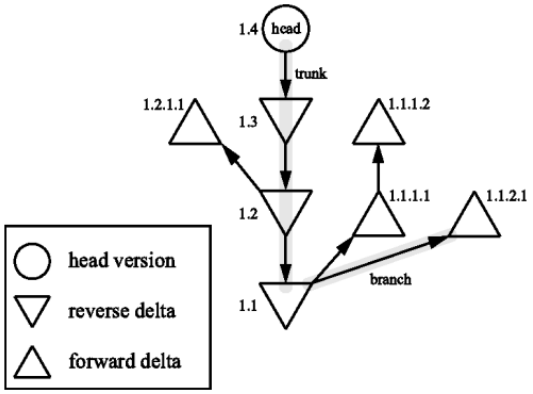
RCS: buscaba mejorar el problema de SCCS en el que cuantos más cambios había en el archivo, se volvía más lento construir las diferentes versiones. Ahora bien, no nos interesa que 10 versiones atrás sean rápidas en acceder, pero si las ultimas 5, por ejemplo. Por ello, se inventó el algoritmo de deltas invertidas que mantenía la última versión, y los deltas hacia atrás.

_ Ventajas:

- Las versiones más recientes eran más rápidas de construir que las versiones más antiguas.
- Liviano, fácil de instalar y usar.

_ Desventaja:

- Si teníamos un branch, no podíamos hacer reverse delta, sino que teníamos que hacer un forward delta. Entonces, debíamos ir hacia atrás y después hacia adelante.
- No soporta networking (esperable para la época).
- Locking (esperable también).
- Deltas por archivo y no por conjunto de archivos.



Subversion: las características son:

- Merge-before-commit.
- Operación por conjunto de archivos.
- Centralizado.
- Client-server model

_ Ventajas:

- Commits files-set based and atomic.
- Merge-before-commit.

_ Desventaja:

- El modelo centralizado imposibilita el trabajo desconectado.

Git: decimos que es distinto a los anteriores porque no se centra en ahorrar espacio utilizando deltas.

_ Características:

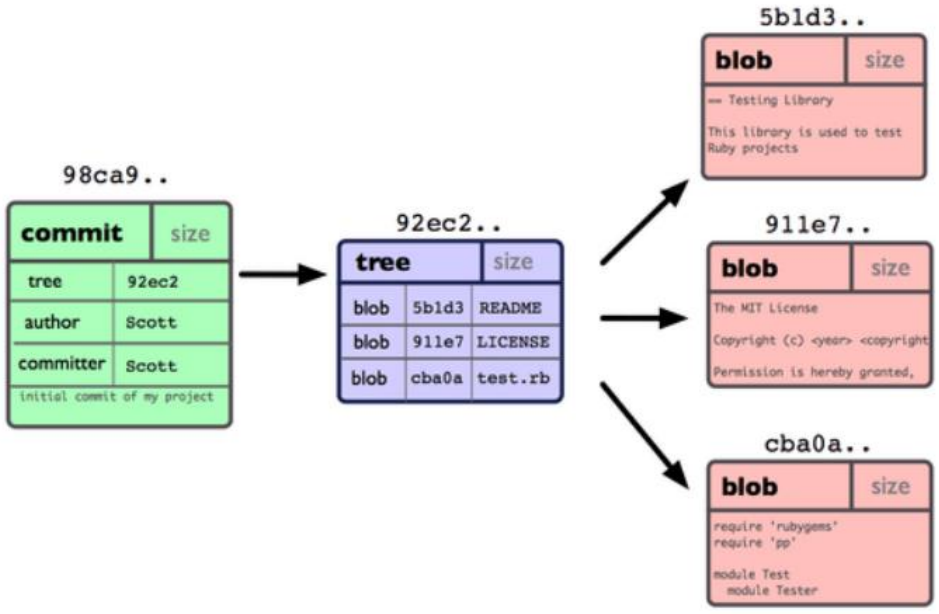
- Commit-before-merge.
- Operación por conjunto de archivos.
- Distribuido.

_ Ventajas:

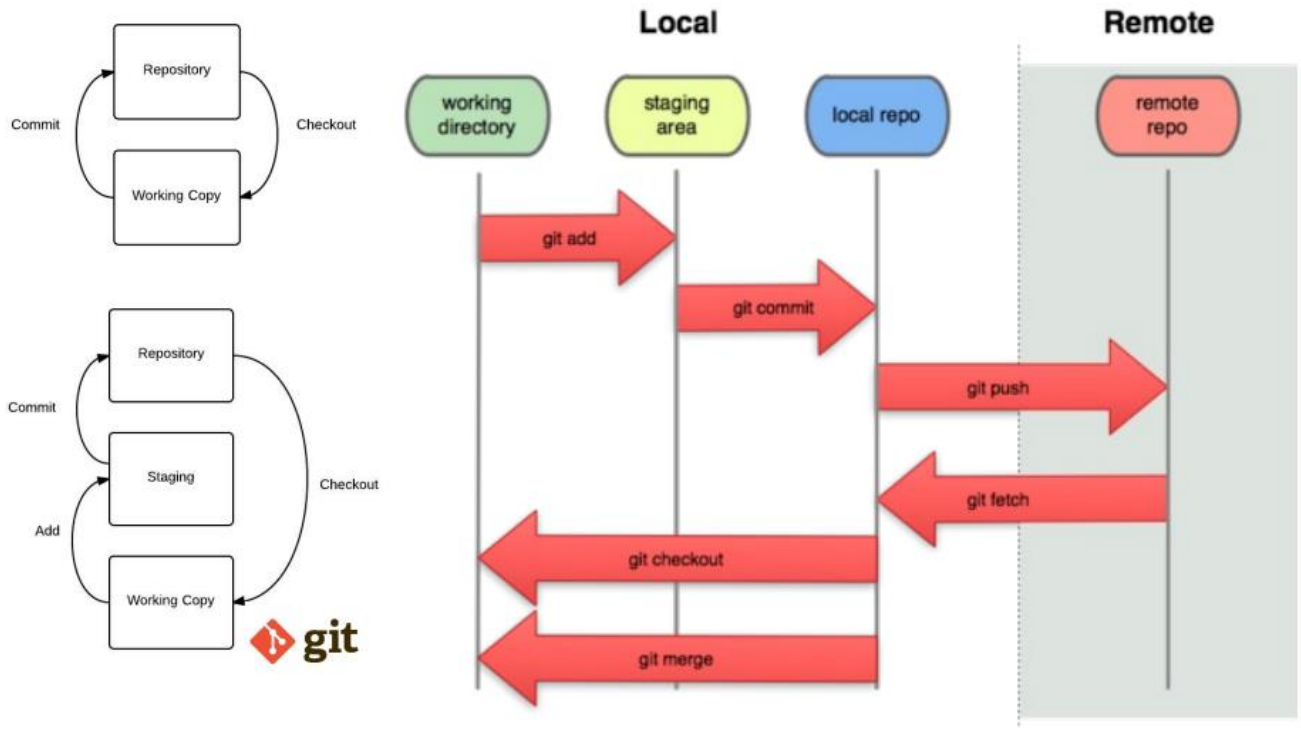
- Commits files-sets and atomic.
- Commit-before-merge.
- Posibilita el trabajo desconectado.

_ Estructura: un commit se compone por tres partes:

- Commit: contiene información del autor, un comentario sobre el cambio y un puntero al objeto de tipo tree.
- Tree: dice cuáles son los archivos que están en el árbol de directorio u otros directorios (pueden haber más de un tree, anidados), y cada uno de esos archivos apuntaran al contenido que tenía en ese momento.
- Archivos de tipo blob: son el contenido de los archivos que estaban en el directorio en ese momento. Es decir, ni siquiera estos blobs tienen un nombre de archivo, ya que son simplemente contenido. Git toma el contenido, lo mete en un archivo binario, un compress file, calcula un hash para identificarlo unívocamente y luego lo agrega en el árbol. Separa el nombre del contenido, porque solo le interesa indexar el contenido del file. Puede haber dos archivos que tengan el mismo contenido. Esto le permite ahorrar espacio, y no guarda deltas sino que reutiliza los contenidos.



Git 2-tree vs 3-tree architecture



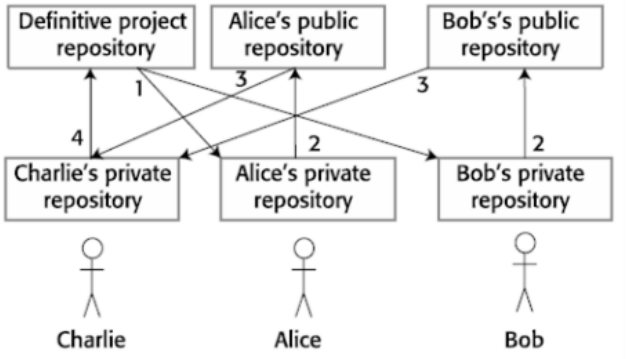
- Git add: mueve los cambios del directorio de trabajo al área del entorno de ensayo. Así puedes preparar una instantánea antes de confirmar en el historial oficial.
- Git commit: captura una instantánea de los cambios preparados en ese momento del proyecto. Las instantáneas confirmadas pueden considerarse como versiones "seguras" de un proyecto. Git no las cambiará nunca a no ser que se lo pidas expresamente.
- Git fetch: descargará el contenido remoto sin modificar el estado del repositorio local.
- Git pull: descargará el contenido remoto y tratará inmediatamente de cambiar el estado del repositorio local para reflejar ese contenido. De modo accidental, esto puede provocar que el repositorio local entre en conflicto. Del remote repo va directo al working directory o al local repo, mediante git pull.
- Git checkout: permite desplazarnos entre las ramas creadas por git branch. Al extraer una rama, se actualizan los archivos en el directorio de trabajo para reflejar la versión almacenada en esa rama y se indica a Git que registre todas las confirmaciones o commits nuevos en dicha rama.

Desarrollo open source

_ Los VCS distribuidos son esenciales para el desarrollo open source/código abierto. Varios desarrolladores pueden estar trabajando simultáneamente en el mismo proyecto sin una coordinación central. Así como cada desarrollador mantiene copias privadas del repositorio del proyecto en su propia computadora, los desarrolladores

también mantienen un repositorio público al cuál todos hacen push (en realidad pull request) de las nuevas versiones de los componentes que han modificado.

_ Es el administrador del proyecto el encargado de decidir cuándo hacer pull de esos cambios para incorporarlos definitivamente al sistema.



VCS Flows - Branching Models

Gitflow: es una forma de estructurar el proceso de desarrollo, en términos de cómo se utilizan los branches en el repositorio. Tenemos distintos tipos de branches:

- Develop: se utiliza para hacer las features que se están desarrollando para el próximo release. Para cada una de estas features se crea una "Feature branch" donde se trabajará, y luego mediante pull request lo unimos al Develop.

- Releases: una vez que tenemos todas las features que van a ir juntas a un release, se promueve dicho código a una “Release branch”. Probablemente, en esta branch se realice testing, validaciones para descubrir bugs, etc.
- Master: una vez corregidos los bugs, movemos el código a una Master branch, y ese es nuestro release.
- Hotfixes: existen esta branch para arreglar algún error en producción y luego volvemos a releasear.

_ Ventajas:

- Da mayor control al separar en niveles de estabilidad en el producto.

_ Desventajas:

- El proceso se hace pesado, por ende más lento.
- Tenemos muchas versiones de nuestro producto. Los branches están alejados unos de otro y los cambios se unifican en momentos determinados. No esta continuamente integrado.

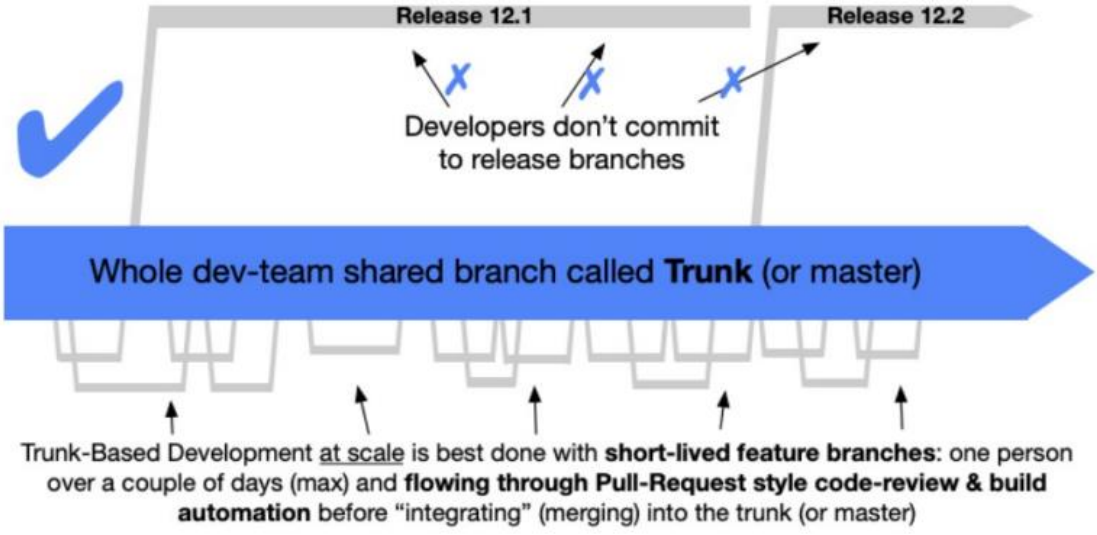
_ Recomendado cuando:

- El equipo no tiene experiencia o posee muchos junior developers.
- El producto es estable.

_ No recomendado cuando:

- En los Startups (empresas emergentes gracias a un modelo de negocio escalable y al uso de las nuevas tecnologías).
- Se requiere iterar rápidamente.
- Los desarrolladores son en su mayoría seniors.

Trunk based development: hace referencia a realizar todos los cambios en master/main.



_ Características:

- Más flexible.
- Escala mejor.
- Permite hacer integración continua.

_ Recomendado cuando:

- En los startups.
- Se requiere iterar rápidamente.
- Los desarrolladores son en su mayoría senior.

_ No recomendado cuando:

- El equipo no tiene experiencia, o hay muchos junior developers.
- El producto es estable.

Diseño de sistemas distribuidos

Introducción

_ Prácticamente, todos los grandes sistemas basados en computadoras son ahora sistemas distribuidos, y queda demostrado cada vez que vemos una película on demand, compramos algo de manera online, pedimos un taxi usando nuestro smartphone, o buscamos algo online. Todas estas empresas utilizan sistemas altamente escalables, altamente disponibles y distribuidos para manejar millones de usuarios al mismo tiempo, cantidades masivas de datos (petabytes) y ofrecer una experiencia de usuario consistente. Incluso, el sitio web más simple corriendo en la nube está siendo ejecutado en un sistema distribuido.

- La nube (Azure, AWS, GCP, etc) es un sistema distribuido diseñado para las compañías y desarrolladores de software.

Sistemas distribuidos

Concepto

_ Un sistema distribuido es aquel que implica numerosas computadoras, en contraste con los sistemas centralizados en que todos los componentes de sistema se ejecutan en una sola computadora.

_ Es un sistema que consiste de varios procesos ejecutándose en computadoras diferentes que se comunican entre sí a través de una red y comparten un estado o trabajan en conjunto para lograr un objetivo común. Conjunto de computadoras independientes que aparecen ante el usuario como una única computadora.

_ Algunos conceptos importantes en la definición son:

Concurrencia: todas las computadoras operan al mismo tiempo.

Independencia: las computadoras funcionan y fallan de manera independiente.

No hay reloj global: las computadoras no comparten un reloj global.

_ Los sistemas distribuidos tienen las siguientes ventajas:

- Compartición de recursos: permiten compartir los recursos de hardware y software, tales como discos, impresoras, archivos y compiladores, que se asocian con computadoras en una red.
- Apertura: por lo general, son sistemas abiertos, lo cual significa que están diseñados en torno a protocolos estándar que permiten la combinación de equipo y software de diferentes proveedores.
- Concurrencia: en estos sistemas, grandes procesos pueden ejecutarse al mismo tiempo en computadoras independientes en red. Dichos procesos pueden (pero no es necesario) comunicarse uno con el otro durante su operación normal.
- Escalabilidad: al menos en principio, los sistemas distribuidos son escalables cuando las capacidades del sistema pueden aumentarse al agregar nuevos recursos para enfrentar nuevas demandas del sistema. En la práctica, la red que vincula las computadoras individuales en el sistema puede limitar la escalabilidad del sistema.
- Tolerancia a fallas: la disponibilidad de muchas computadoras y el potencial de reproducir información significa que los sistemas distribuidos pueden tolerar algunas fallas de hardware y software. En la mayoría de los sistemas distribuidos, puede darse un servicio degradado al ocurrir fallas; la pérdida completa de servicio sólo sucede cuando hay una falla de red.

_ Desventajas:

- Los sistemas distribuidos son inherentemente más complejos que los sistemas centralizados. Esto los hace más difíciles de diseñar, implementar y poner a prueba.
- Es más complicado entender las propiedades emergentes de los sistemas distribuidos debido a la complejidad de las interacciones entre los componentes y la infraestructura del sistema.

Escalabilidad

_ La escalabilidad es una medida que indica como se ve afectada la performance cuando se agregan nuevos recursos. Pensemos en un sitio de comercio electrónico básico donde se nos permita comprar y dejar reviews. Los usuarios pueden acceder vía browser desde la PC y desde el teléfono. Con relativamente pocos usuarios el tiempo de respuesta es aceptable, pero cuando la base de usuarios crece, el tiempo de respuesta se degrada considerablemente, y el servidor tiene que soportar una carga mucho mayor.

- Tiempo de respuesta (RT): tiempo que un sistema requiere para procesar un pedido visto desde fuera.
- Carga: es una medida de cuan estresado están un sistema. Usado generalmente como contexto de otra medida de performance.
    - Ejemplo: el RT = 0,5s con 10 usuarios y 2s con 20 usuarios.

Escalabilidad vertical (Scale Up): se refiere a cuando compramos el servidor más avanzado, con el máximo de memoria y de poder de procesamiento posibles, pero también el más costoso. Algunas características son:

- Consiste en agregar más o mejores recursos a un servidor, cambiar el hardware por uno más potente (más memoria, procesador más rápido, más disco, etc).
- Demora el problema si la base de usuarios sigue creciendo.
- Tiene un límite cercano.

_ La escalabilidad de un sistema refleja su disponibilidad para entregar una alta calidad de servicio conforme aumentan las demandas al sistema. Identificamos tres dimensiones de la escalabilidad:

- Tamaño: debe ser posible agregar más recursos a un sistema para enfrentar el creciente número de usuarios.
- Distribución: debe ser posible dispersar geográficamente los componentes de un sistema sin reducir su rendimiento.
- Manejabilidad Debe ser posible administrar un sistema conforme aumenta en tamaño, incluso si las partes del sistema se ubican en organizaciones independientes.

_ En términos de tamaño, hay una distinción entre:

- Expandir (scaling up): significa sustituir recursos en el sistema con recursos más poderosos. Por ejemplo, es posible expandir o aumentar la memoria de un servidor de 16 GB a 64 GB.
- Ampliar (scaling out o escalabilidad horizontal): significa agregar recursos adicionales al sistema (por ejemplo, un servidor Web adicional para trabajar junto a un servidor existente, o agregar más servidores). Es a menudo más efectivo en costo que expandir, aun cuando, por lo general, representa que el sistema debe diseñarse para que sea posible el procesamiento concurrente.

_ En cuanto a la seguridad, cuando se distribuye un sistema, el número de formas en que éste puede ser atacado aumenta considerablemente, en comparación con los sistemas centralizados. Si una parte del sistema es atacada con éxito, entonces el atacante podrá usar esto como una “puerta trasera” en otras partes del sistema.

Problema de los sistemas centralizados

_ Estos son los problemas que dan origen a los sistemas distribuidos:

- Escalabilidad: tanto vertical como limitada.
- Un único punto de fallo:
  - Error: si algo se rompe o surge un error, se detienen el resto de las actividades.
  - Mantenimiento: si tenemos que actualizar el software, debemos parar el resto de las actividades.
- Localización: aumenta la latencia con la distancia, porque los saltos en la red van a ser más. Por ende, la experiencia de usuario se degrada para usuarios remotos, debido a una mayor latencia (tiempo mínimo requerido para obtener cualquier tipo de respuesta), y mayor ancho de banda.

Problemas de los sistemas distribuidos

_ Los sistemas distribuidos son más complejos que los sistemas que se ejecutan en un solo procesador. Esta complejidad surge porque es prácticamente imposible tener un modelo descendente de control de dichos sistemas. Los nodos en el sistema que entregan funcionalidad con frecuencia son sistemas independientes sin una sola autoridad encargada de ellos. La red que conecta dichos nodos es un sistema de gestión independiente. Es un sistema complejo por derecho propio y no puede controlarse por los propietarios de los sistemas que usan la red. Por lo tanto, existe una imprevisibilidad inherente en la operación de los sistemas distribuidos que debe considerar el diseñador del sistema.

_ Algunos de los conflictos de diseño más importantes que deben considerarse en la ingeniería de sistemas distribuidos son:

- Transparencia: ¿En qué medida el sistema distribuido debe aparecer al usuario como un solo sistema? ¿Cuándo es útil para los usuarios entender que el sistema es distribuido?.
- Apertura: ¿Un sistema debe diseñarse usando protocolos estándar que soporten interoperabilidad o deben usarse protocolos más especializados que restrinjan la libertad del diseñador?.
- Escalabilidad: ¿Cómo puede construirse el sistema para que sea escalable? Esto es, ¿cómo podría diseñarse un sistema global para que su capacidad se pueda aumentar en respuesta a demandas crecientes hechas sobre el sistema?.
- Seguridad: ¿Cómo pueden definirse e implementarse políticas de seguridad útiles que se apliquen a través de un conjunto de sistemas administrados de manera independiente?.
- Calidad de servicio: ¿Cómo debe especificarse la calidad del servicio que se entrega a los usuarios del sistema y cómo debe implementarse el sistema para entregar una calidad de servicio aceptable para todos los usuarios?.

- Gestión de fallas: ¿Cómo pueden detectarse las fallas del sistema, contenerse (de modo que tengan efectos mínimos sobre otros componentes del sistema) y repararse?

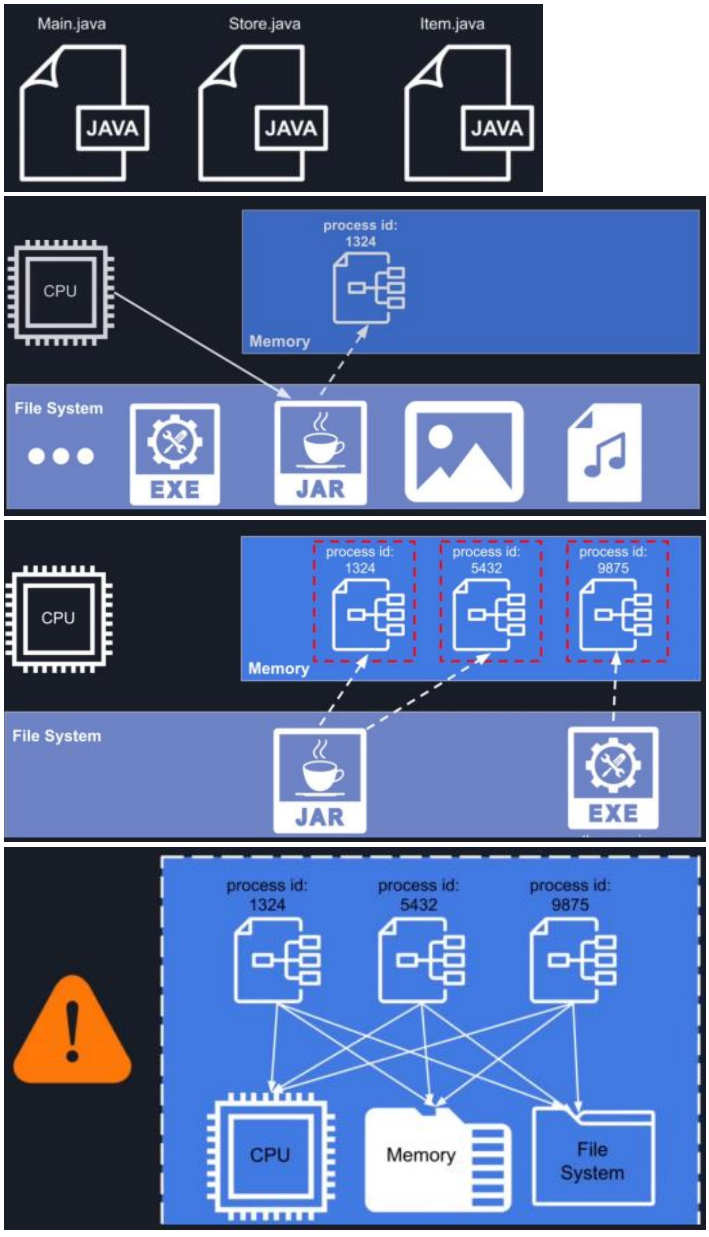
Capacidad de los sistemas distribuidos

_ Un sistema distribuido bien diseñado:

- Escala horizontalmente: tanto para manejar el aumento de carga, como para manejar el aumento de datos.
- Es altamente escalable: idealmente de manera lineal, en donde, cuando agregamos una cantidad de recursos, aumentamos la capacidad en misma proporción. Aumentamos y disminuimos la capacidad según la demanda.
- Es altamente disponible: no tiene un solo punto de falla.
- Experiencia de usuario: la latencia es independiente de dónde se encuentra el usuario.

Procesos

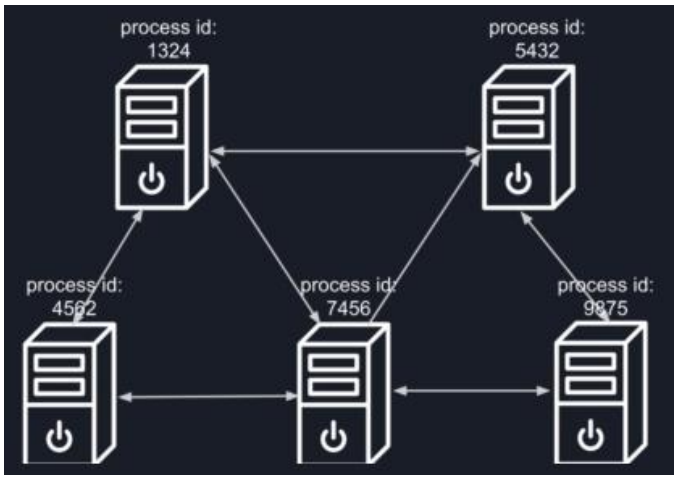
_ Cuando hablamos de un proceso hablamos de una instancia de una aplicación en memoria, en donde todos los procesos están aislados entre si.



_ Los procesos no se refieren a un sistema distribuido, sino que estos están en la misma computadora y comparten los mismos recursos. No pueden escalar más allá de la capacidad de la computadora dónde se ejecutan. La comunicación entre estos se lleva a cabo en la red (network), sistema de archivos (filesystem) y la memoria.

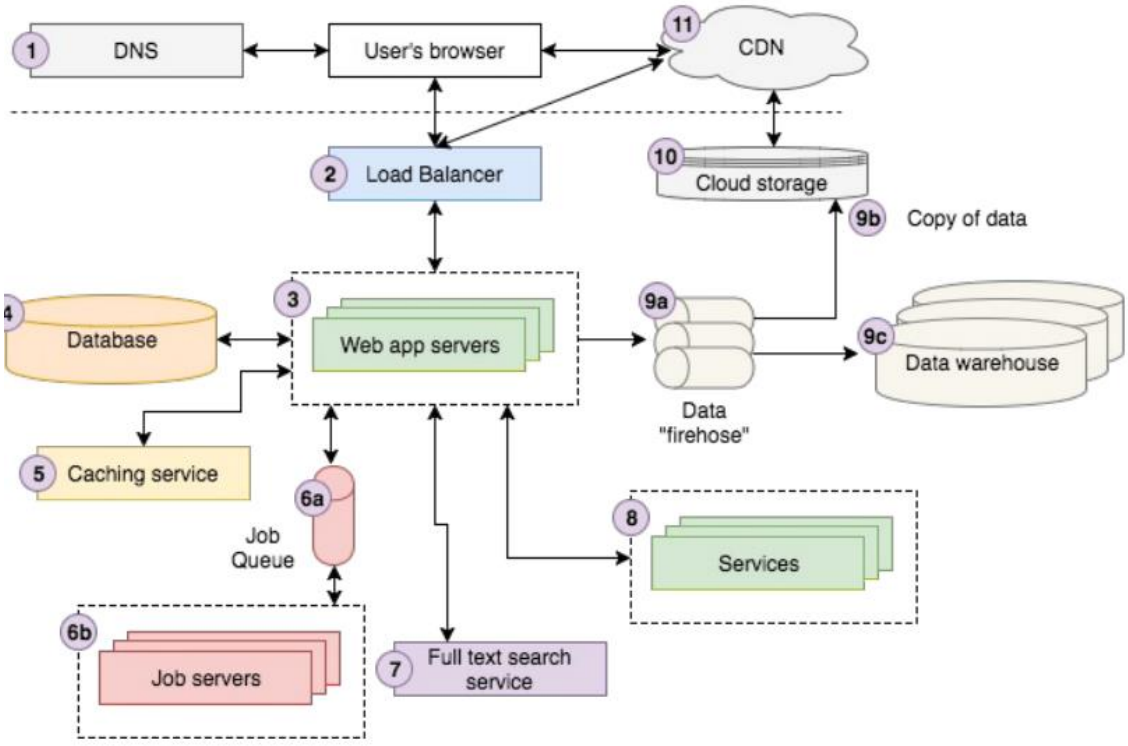
Procesos en computadoras diferentes: podemos poner los procesos en distintas computadoras para desacoplarlos en el uso de recursos.

- Escalabilidad horizontal: podemos agregar más computadoras para extender la cantidad de memoria y el poder de procesamiento.
  - Tolerancia a fallos: aunque algunas computadoras fallen, otros procesos siguen corriendo en las computadoras que continúan funcionando.
  - Manteamiento: se puede evitar “downtime” haciendo un upgrade progresivo.
- Comunicación por red: la única opción de comunicación para los procesos corriendo en diferentes computadoras es utilizar la red. La red no es confiable por definición, ya que puede no estar disponible, puede tener una gran latencia, o puede estar congestionada.



- Tienen una meta común: tienen que operar en conjunto para lograr un objetivo específico:
  - Mantienen una vista común del mundo.
  - Trabajan juntos para lograr un objetivo común.
  - Aparecen ante el usuario como una única entidad.
  - Si no tuvieran una meta común serían una colección de procesos separados.

Ejemplo de sistema distribuido



Diseño de sistemas distribuidos

Patrones

_Soluciones conocidas y reusables a problemas recurrentes en el software

Patrón de diseño: es una descripción del problema y la esencia de su solución, de modo que la solución puede reutilizarse en diferentes configuraciones. El patrón no es una especificación detallada sino que se puede considerar como una descripción de sabiduría y experiencia acumuladas, una solución bien probada a un problema común

_ Básicamente los patrones de diseño son una solución bien probada a un problema común que conjunta experiencia y buena práctica en una forma que pueda reutilizarse. Es una representación abstracta que puede ejemplificarse en varias formas.

_ Los elementos son las clases, y en si los patrones resuelven un problema común. Algunos ejemplos son abstract factory, state, strategy, proxy, monads, etc.

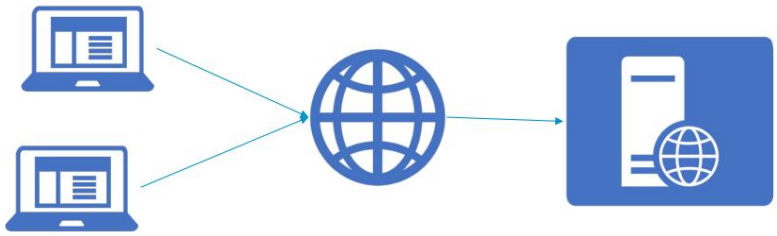
Patrón arquitectónico: se puede considerar como una descripción abstracta de una arquitectura de software que se ensayó y puso a prueba en algunos sistemas de software distintos. La descripción del patrón incluye información acerca de dónde es adecuado usar el patrón, cuándo es y cuándo no es adecuado usar dicho patrón, así como sobre las fortalezas y debilidades del mismo, y la organización de los componentes de la arquitectura.

_ Estos patrones deben describir una organización de sistema que ha tenido éxito en sistemas previos. Un patrón arquitectónico es una descripción de una organización del sistema y captan la esencia de una arquitectura que se usó en diferentes sistemas de software. Estilo y patrón llegaron a significar lo mismo.

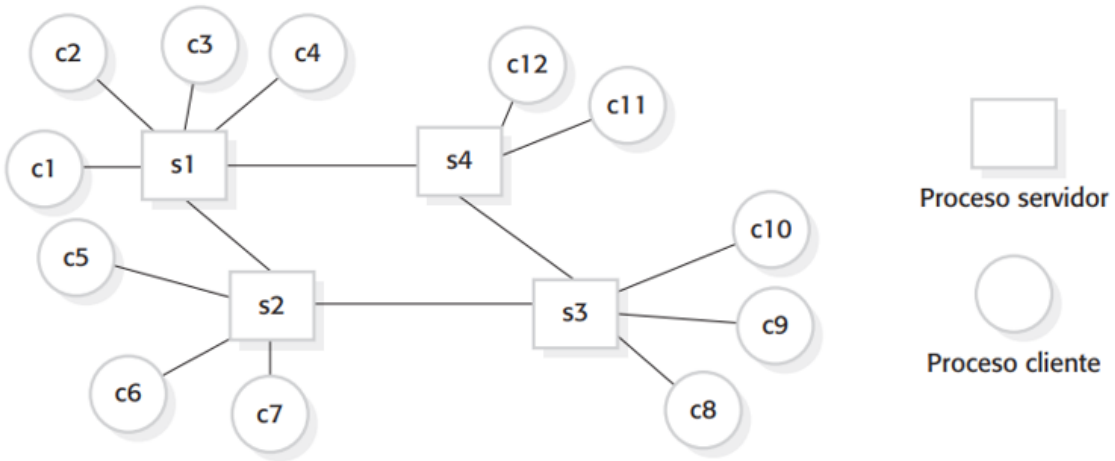
_ Está formada por componentes, tratan de entender como encajan las principales partes, como fluye la información, y definen otros aspectos estructurales. Algunos ejemplos son en capas, service oriented, microservices, microkernel, etc.

Estilos arquitectónicos

Arquitectura cliente-servidor: los sistemas distribuidos, a los que se accede por Internet, se organizan normalmente como sistemas cliente-servidor. En un sistema cliente-servidor, el usuario interactúa con un programa que se ejecuta en su computadora local (por ejemplo, un navegador Web o una aplicación basada en telefonía). Éste interactúa con otro programa que se ejecuta en una computadora remota (por ejemplo, un servidor Web). La computadora remota proporciona servicios, como acceso a páginas Web, que están disponibles a clientes externos.



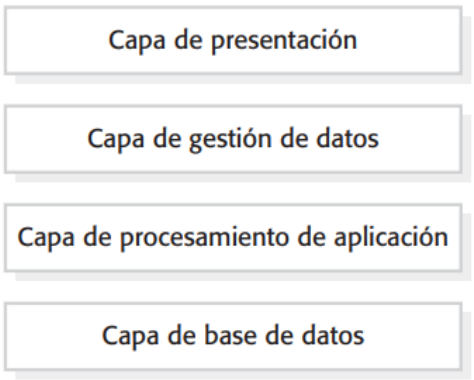
_ Los clientes deben estar al tanto de los servidores que están disponibles, pero no conocen la existencia de otros clientes. Clientes y servidores son procesos separados, como se muestra en la siguiente imagen, la cual ilustra una situación donde existen cuatro servidores (s1-s4), que entregan diferentes servicios. Cada servicio tiene un conjunto de clientes asociados que acceden a dichos servicios:



_ Varios procesos servidor diferentes pueden ejecutar en el mismo procesador, pero, con frecuencia, los servidores se implementan como sistemas multiprocesador en los

que una instancia independiente del proceso servidor se ejecuta en cada máquina. El software de balanceo de carga distribuye las peticiones de servicio de los clientes a diferentes servidores, de modo que cada servidor realiza la misma cantidad de trabajo. Esto permite el manejo de mayor volumen de transacciones con los clientes, sin degradar la respuesta a clientes individuales.

Arquitectura en capas: los sistemas cliente-servidor dependen de que exista una separación clara entre la presentación de información y los cálculos que crea y procesa esa información. En consecuencia, se debe diseñar la arquitectura de los sistemas distribuidos cliente-servidor para que se estructuren en varias capas lógicas, con interfaces claras entre dichas capas. Esto permite que cada capa se distribuya en diferentes computadoras. La siguiente imagen ilustra este modelo y muestra una aplicación estructurada en cuatro capas:



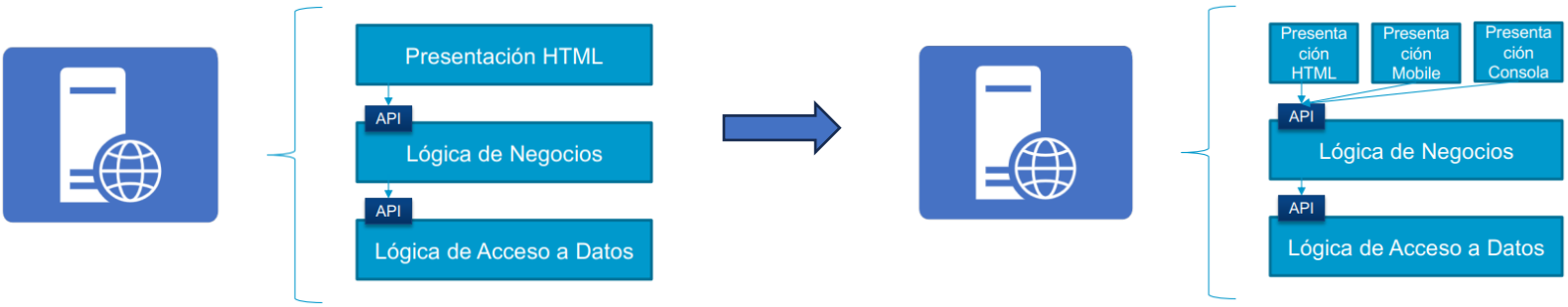
- Capa de presentación: se ocupa de presentar la información al usuario y gestionar todas las interacciones de usuario.
- Capa de gestión de datos: que gestiona los datos que pasan hacia y desde el cliente. Esta capa puede implementar comprobaciones en los datos, generar páginas Web, etc.
- Capa de procesamiento de aplicación: se ocupa de implementar la lógica de la aplicación y, de este modo, proporciona la funcionalidad requerida a los usuarios finales.
- Capa de base de datos que almacena los datos y ofrece servicios de gestión de transacción, etc.

_ Las capas de abajo proveen servicios a las capas de arriba, los pedidos fluyen de arriba hacia abajo, la capa superior solo se comunica con la capa inmediata inferior y cada capa puede ser reemplazada o modificada sin afectar toda la arquitectura.

_ Proporciona los siguientes beneficios:

- Mantenibilidad
- Reusabilidad
- Organización
- Agrupación de desarrolladores según sus conocimientos.

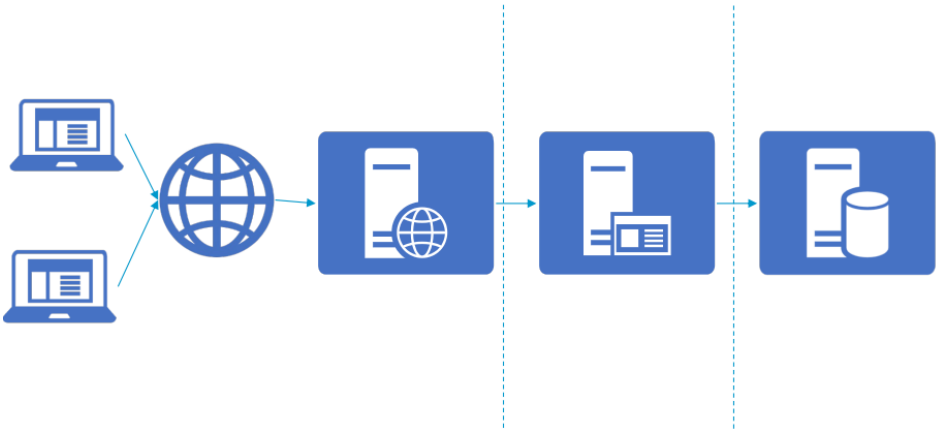
_ Como desventaja, por ejemplo, si tenemos que agregar una nueva función de negocios, debemos que modificar las otras capas.



_ Podemos agregar otros tipos de presentación, no solo el HTML original, también una app web o una consola, pero que hablen con la misma capa inferior. La lógica de negocios no cambia, por lo que si tiene que mostrar una lista con los clientes o dar de alta un proveedor, no importa donde sea, si en una página web o en una app web.

Arquitectura N-Capas (N-Tier): divide una aplicación en capas lógicas y niveles físicos. Las capas son una forma de separar responsabilidades y gestionar dependencias. Cada capa tiene una responsabilidad específica. Una capa superior puede utilizar los servicios de una capa inferior, pero no al revés.

_ En este caso hablamos de capas físicas. La idea es que puedo correr toda la capa de presentación como un web server, que solo se dedica a aceptar request del cliente, mandarlas al application server, que tiene la lógica de negocios, y finalmente esta manda request a la aplicación de base de datos para realizar alguna operación de ser necesario.



_ Esta arquitectura me permite escalar de manera independiente, como por ejemplo tener varios application server. Una aplicación de N niveles puede tener una arquitectura de capa cerrada o una arquitectura de capa abierta:

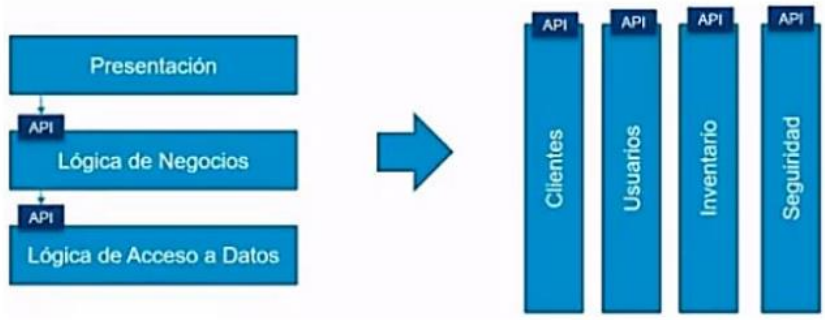
- Arquitectura de capa cerrada: una capa solo puede llamar a la siguiente capa inmediatamente hacia abajo.

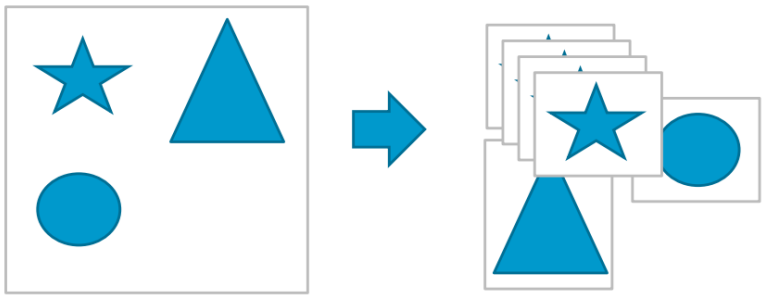
- Arquitectura de capa abierta: una capa puede llamar a cualquiera de las capas debajo de ella.

_ Beneficios:

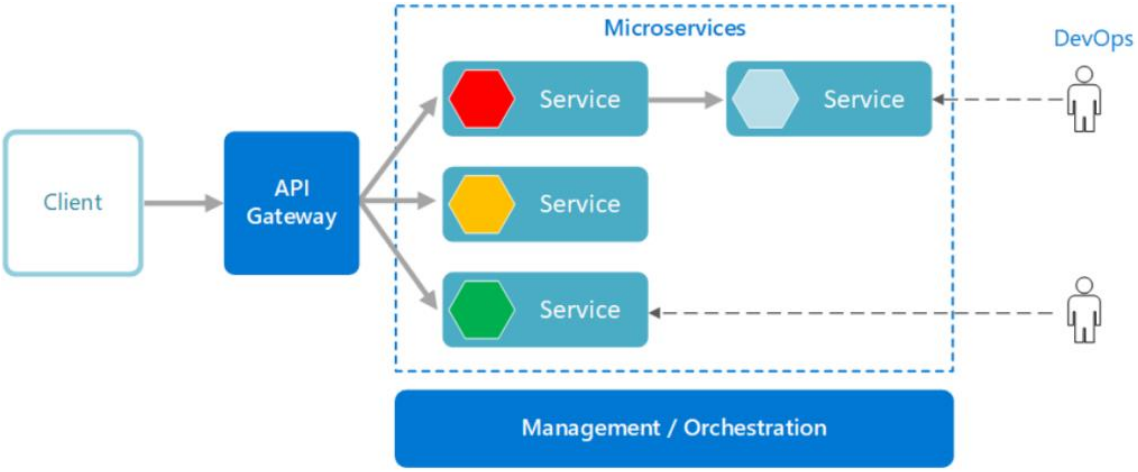
- Portabilidad entre la nube y local, y entre plataformas en la nube.
- Menos curva de aprendizaje para la mayoría de los desarrolladores.
- Evolución natural del modelo de aplicación tradicional.
- Abierto a un entorno heterogéneo (Windows / Linux).

Arquitectura microservicios: en ingeniería de software, una aplicación monolítica hace referencia a una aplicación software en la que la capa de interfaz de usuario, lógica de negocios y la capa de acceso a datos están combinadas en un mismo programa y sobre una misma plataforma. Si rompemos el monolito que habíamos visto como desventaja en la arquitectura anterior, surgen lo que llamamos microservicios.





_ Una arquitectura de microservicios consta de una colección de pequeños servicios autónomos. Cada servicio es autónomo y debe implementar una única capacidad empresarial.



_ Los microservicios son pequeños, independientes y poco acoplados. Un solo equipo pequeño de desarrolladores puede escribir y mantener un servicio. Los microservicios son de tamaño pequeño, habilitados para mensajería, limitados por contextos, desarrollados de forma autónoma, implementables de forma independiente, descentralizados y construidos y lanzados con procesos automatizados.

_ En la arquitectura microservicios hablamos de trade-offs, lo cual hace referencia a que los servicios son más simples pero la arquitectura se vuelve más compleja, y dicha complejidad es manejada herramientas y automatización, e introducimos el concepto de monitoreo.

_ Nombramos algunas de las características de los servicios:

- Cada servicio es una base de código separada, que puede ser administrada por un pequeño equipo de desarrollo.
- Los servicios se pueden implementar de forma independiente. Un equipo puede actualizar un servicio existente sin reconstruir y volver a implementar toda la aplicación.
- Los servicios se comunican entre sí mediante API bien definidas. Los detalles de implementación interna de cada servicio están ocultos para otros servicios.
- Los servicios no necesitan compartir la misma pila de tecnología, bibliotecas o marcos.
- Coarse gain: son de grano grueso, agrupan muchas funcionalidades.
- Autonomus: no necesitan de otros servicios para realizar sus tareas.
- Self contain: no están en distintas partes, son una única unidad.

_ Los beneficios de implementar microservicios son:

- Agilidad: debido a que los microservicios se implementan de forma independiente, es más fácil administrar las correcciones de errores y las versiones de funciones. Puede actualizar un servicio sin volver a implementar la aplicación completa y revertir una actualización si algo sale mal.
- Equipos pequeños y enfocados. Un microservicio debe ser lo suficientemente pequeño como para que un solo equipo de funciones pueda construirlo, probarlo e implementarlo. Los equipos pequeños promueven una mayor agilidad.
- Base de código pequeña: en una aplicación monolítica, con el tiempo existe una tendencia a que las dependencias del código se enreden. Agregar una nueva función requiere tocar el código en muchos lugares. Al no compartir códigos o almacenes de datos, una arquitectura de microservicios minimiza las dependencias y eso facilita la adición de nuevas funciones.
- Mezcla de tecnologías: los equipos pueden elegir la tecnología que mejor se adapte a su servicio, utilizando una combinación de pilas de tecnología según corresponda.
- Escalabilidad: los servicios se pueden escalar de forma independiente, lo que le permite escalar los subsistemas que requieren más recursos, sin escalar toda la

aplicación. Con un orquestador como Kubernetes o Service Fabric, puede empaquetar una mayor densidad de servicios en un solo host, lo que permite una utilización más eficiente de los recursos.
- Aislamiento de datos: es mucho más fácil realizar actualizaciones de esquemas, porque solo se ve afectado un microservicio.

_ Los beneficios de los microservicios no son gratuitos. Estos son algunos de los desafíos a considerar antes de embarcarse en una arquitectura de microservicios:

- Complejidad: una aplicación de microservicios tiene más partes móviles que la aplicación monolítica equivalente. Cada servicio es más simple, pero todo el sistema en su conjunto es más complejo.
- Falta de gobernanza: puede terminar con tantos lenguajes y marcos diferentes que la aplicación se vuelva difícil de mantener.
- Congestión y latencia de la red: el uso de muchos servicios pequeños y granulares puede resultar en una mayor comunicación entre servicios. Además, si la cadena de dependencias de servicio se vuelve demasiado larga (el servicio A llama a B, que llama a C ...), la latencia adicional puede convertirse en un problema.
- Gestión: para tener éxito con los microservicios se requiere una cultura DevOps madura.
- Control de versiones: se pueden actualizar varios servicios en cualquier momento, por lo que, sin un diseño cuidadoso, es posible que tenga problemas con la compatibilidad hacia atrás o hacia adelante.

Cubo de escalabilidad

_ Es un modelo de escalabilidad de tres dimensiones usado para representar de manera simple las formas de escalar una aplicación más allá del tradicional escalamiento vertical, que consiste en aumentar los recursos de procesamiento. Básicamente, podemos escalar una aplicación según tres ejes, X, Y, y Z; donde cada eje representa una manera diferente de escalar.

Escalamiento en eje X: es también conocido como escalamiento horizontal (scale out) y consiste en ejecutar varias copias de una aplicación detrás de un balanceador de carga. Si hay N copias, entonces cada copia se encarga de 1/N de la carga. Este es un método sencillo para escalar una aplicación, de uso general y común en servidores de aplicación o servidores web. Sin embargo, este tipo de escalamiento presenta algunos inconvenientes:

- Debido a que cada copia accede potencialmente a todos los datos, se requieren caches con mayor capacidad de memoria para ser eficaz.
- Otro inconveniente se basa en lo que se menciona más arriba sobre el manejo del desarrollo creciente y la complejidad de la aplicación.

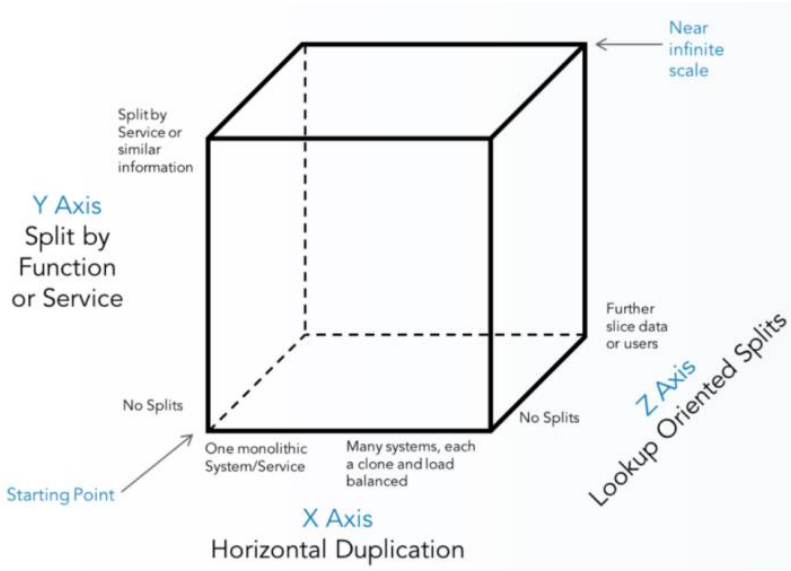
Escalamiento en eje Z: de manera similar al eje X, cada servidor ejecuta una copia idéntica del código. La gran diferencia es que cada servidor es responsable sólo por un subconjunto de los datos. Algún componente del sistema es responsable de encaminar cada pedido al servidor apropiado. Uno de los criterios de enrutamiento de uso común es un atributo de la solicitud, tal como la clave primaria de la entidad que se quiere acceder. Otro ejemplo típico es el de aplicaciones que tasan consumos de servicios, donde la carga se reparte en distintos servidores según la zona o región del cliente, o según alguna otra clave o característica. El escalamiento de eje Z tiene varios beneficios:

- Cada servidor sólo se ocupa de un subconjunto de datos.
- Mejora la utilización de cache y reduce el uso de memoria y E/S de tráfico.
- Mejora la escalabilidad de transacción dado que las solicitudes se distribuyen normalmente a través de múltiples servidores.
- También mejora el aislamiento de fallas, dado que una falla hace que sea inaccesible solo una parte de los datos.

_ Por otro lado, el escalamiento Z tiene algunas contras:

- Un inconveniente es el incremento de complejidad de la aplicación.
- Se necesita implementar un esquema de partición que puede ser difícil, especialmente si necesitamos volver a particionar los datos.
- No resuelve los problemas de desarrollo creciente y la complejidad de la aplicación. Para resolver estos problemas debemos aplicar el escalamiento de eje Y.

Escalamiento en eje Y: a diferencia del eje X y del eje Z, que consisten en la ejecución de múltiples copias idénticas de la aplicación, el escalamiento de eje Y divide la aplicación en múltiples y diferentes servicios. Cada servicio es responsable de una o más funciones estrechamente relacionadas. Hay un par de maneras diferentes de descomponer una aplicación en servicios. Un enfoque consiste en utilizar la descomposición basada en verbos, y definir los servicios que implementan un único caso de uso, como por ej. "login" o "buscar". La otra opción es descomponer la aplicación por sustantivos, y crear servicios responsables por todas las operaciones relacionadas con una entidad particular, como por ej. "clientes" u "'ordenes". Una aplicación puede usar una combinación de ambas descomposiciones, basada en verbos y basada en sustantivos.



Integración continua y despliegue seguro

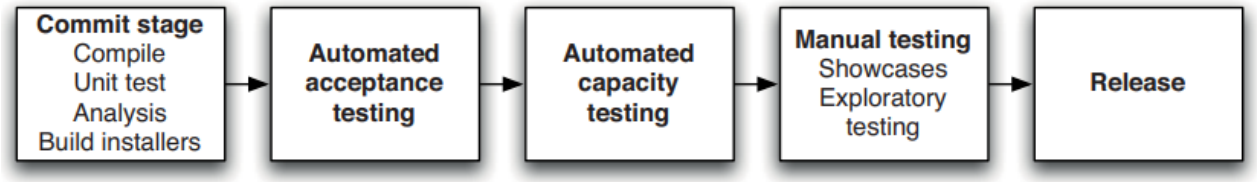
El problema de entrega de software

Introducción

_ El problema más importante al que nos enfrentamos como profesionales del software es este: si alguien piensa en una buena idea, ¿cómo entregársela a los usuarios tan rápido como sea posible?, existen muchas metodologías de desarrollo de software, pero la mayoría se enfocan principalmente en la gestión de requisitos y su impacto en el esfuerzo de desarrollo. Muchas cubren los distintos enfoques para el diseño, desarrollo y pruebas del software, pero también, cubren sólo un fragmento del flujo de valor que entrega valor a las personas y organizaciones que patrocinan nuestros esfuerzos.

_ Entonces, nos preguntamos ¿qué sucede una vez que se identifican los requisitos y se diseñan, desarrollan y prueban las soluciones?, ¿cómo se unen y coordinan estas actividades para que el proceso sea lo más eficiente y confiable posible?, ¿cómo permitimos que los desarrolladores, los evaluadores y el personal de operaciones y construcción trabajen juntos de manera efectiva?. A continuación, describimos un patrón eficaz para llevar el software desde el desarrollo hasta el lanzamiento. Describimos técnicas y mejores prácticas que ayudan a implementar este patrón y mostramos cómo este enfoque interactúa con otros aspectos de la entrega de software.

Deployment pipeline: es el patrón central al cual nos referimos, y este es, en esencia, una implementación automatizada del proceso de creación, implementación, prueba y lanzamiento de su aplicación.

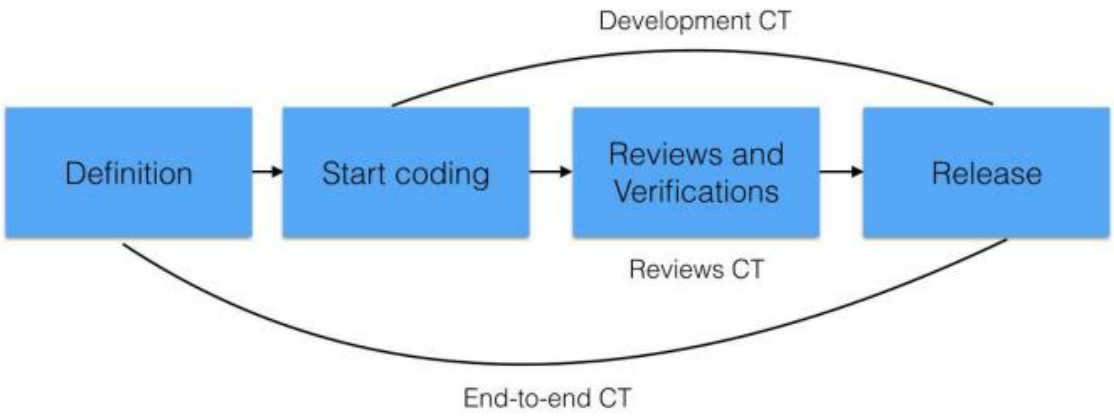


_ La forma en que funciona el deployment pipeline, es la siguiente, cada cambio que se realiza en la configuración, el código fuente, el entorno o los datos de una aplicación desencadena la creación de una nueva instancia del pipeline. Uno de los primeros pasos del pipeline es crear archivos binarios e instaladores. El resto de este, ejecuta una serie de pruebas en los binarios para demostrar que se pueden publicar (released). Cada prueba que pasa la versión candidata (release) nos da más confianza en que esta combinación particular de código binario, información de configuración, entorno y datos funcionará. Si el candidato a la liberación pasa todas las pruebas, puede ser liberado o lanzado.

_ El deployment pipeline tiene sus bases en el proceso de integración continua (continuous integration) llevado a su conclusión lógica. El deployment pipeline tiene como objetivo:

- Hace que cada parte del proceso de creación, implementación, prueba y lanzamiento de software sea visible para todos los involucrados, lo que ayuda a la colaboración.
- Mejora la retroalimentación para que los problemas se identifiquen y resuelvan lo más temprano posible en el proceso.
- Permite a los equipos implementar y lanzar cualquier versión de su software en cualquier entorno a voluntad a través de un proceso totalmente automatizado.

Cycle time (CT): es el tiempo entre que se decide hacer un cambio y se tiene disponible en producción. A veces el cycle time puede durar meses, semanas, horas o minutos. Algo clave para mejorar el CT es la automatización, buenas prácticas y patrones.



Antipatrones

_ El día del lanzamiento de un software tiende a ser tenso. Para la mayoría de los proyectos, es el grado de riesgo asociado con el proceso lo que hace que el lanzamiento sea un momento aterrador, ya que hay muchas cosas que pueden salir mal:

- Preparación manual de los entornos por un team de operations.
- Instalación de dependencias.
- App artifacts copiados a los servidores de producción.
- Configuración copiada o creada manualmente.
- La aplicación es iniciada pieza por pieza.

_ Algunos antipatrones son:

Despliegue manual: la mayoría de las aplicaciones modernas de cualquier tamaño son complejas de implementar e implica muchas partes móviles. Muchas organizaciones lanzan software manualmente. Con esto queremos decir que los pasos necesarios para implementar dicha aplicación se tratan como separados y atómicos, cada uno

realizado por un individuo o equipo. Se deben tomar decisiones dentro de estos pasos, dejándolos propensos a errores humanos. Incluso si este no es el caso, las diferencias en el orden y el tiempo de estos pasos pueden conducir a diferentes resultados.

_ Los signos de este antipatron son:

- La producción de documentación extensa y detallada que describe los pasos a seguir y las formas en que los pasos pueden salir mal.
- Verificación manual para confirmar que la aplicación se está ejecutando correctamente.
- Interacción frecuente con equipo de desarrollo para explicar por qué una implementación va mal en un día de lanzamiento.
- Correcciones frecuentes al proceso de lanzamiento (release) durante el transcurso de una publicación (release).
- Entornos en un clúster que difieren en su configuración, por ejemplo, servidores de aplicaciones con diferentes configuraciones de grupo de conexiones, sistemas de archivos con diferentes diseños, etc.
- Releases o lanzamientos que tardan más de unos minutos en realizarse.
- Releases que son impredecibles en su resultado, que a menudo tienen que ser revertidas o encontrarse con problemas imprevistos.
- Sentarse delante de un monitor a 2 AM. el día del lanzamiento, tratando de averiguar cómo hacerlo funcionar.

_ Ahora, mencionamos las razones por las cuales deberíamos automatizar los releases:

- Cuando las implementaciones no están completamente automatizadas, se producirán errores cada vez que se realizan. Incluso con excelentes pruebas de implementación, los errores pueden ser difíciles de rastrear.
- Cuando el proceso de implementación no está automatizado, no es repetible o confiable, lo que lleva a perder tiempo depurando errores de implementación.
- Un proceso de implementación manual debe documentarse:
    - Mantener la documentación es una tarea compleja y requiere mucho tiempo entre varias personas, por lo que la documentación es generalmente incompleta o desactualizada en un momento dado.
    - La documentación debe hacer suposiciones sobre el nivel de conocimiento del lector y en realidad se suele escribir como ayuda memoria para la persona que realiza el despliegue, haciéndolo opaco para otros.
- Las implementaciones manuales dependen de la implementación de un experto. Si él o ella está de vacaciones o deja de trabajar, estaremos en problemas.
- Hemos oído decir que un proceso manual es más auditable que un uno automatizado. Y es incorrecto, ya que con un proceso manual, no hay garantía de que se haya seguido la documentación.

_ El proceso de implementación automatizado debe ser utilizado por todos, y debe será la única forma en que se implemente el software. Esta disciplina asegura que el

script de implementación funcionará cuando sea necesario y si se produce algún problema tras el lanzamiento, puede asegúrese de que sean problemas con la configuración específica del entorno, no del script.

No testear en un entorno como producción: en este patrón, la primera vez que el software se implementa en un entorno similar de producción (por ejemplo, la puesta en escena - release staging) es una vez que la mayor parte del trabajo de desarrollo ha finalizado o está hecho, o al menos, "hecho" según lo que define el equipo de desarrollo.

_ Los signos de este antipatron son:

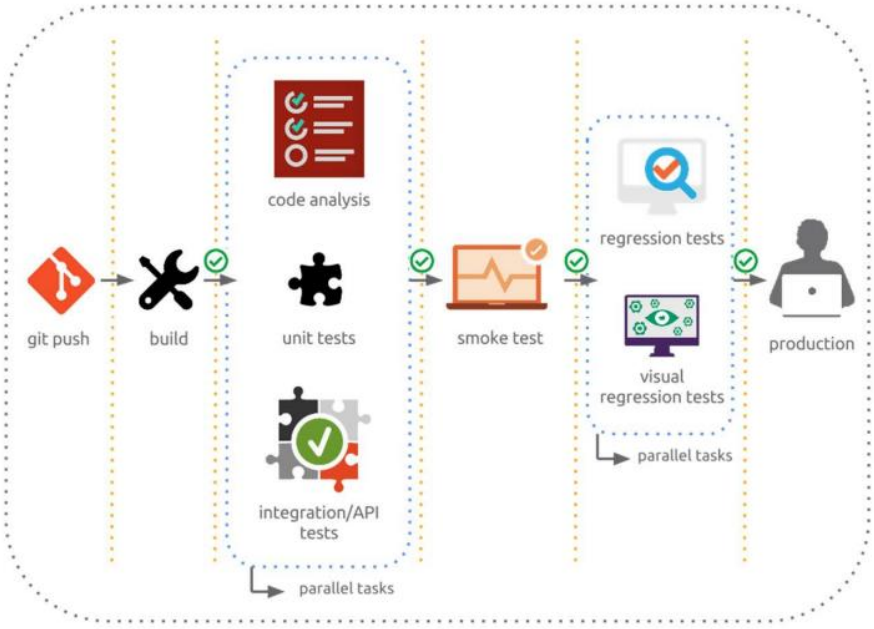
- Si los testers han estado involucrados en el proceso hasta este punto, han probado el sistema en entornos de desarrollo.
- El entorno de producción es lo suficientemente caro como para que el acceso a él esté estrictamente controlado, o no está en su lugar a tiempo, o nadie se molestó en crea uno.
- El equipo de desarrollo reúne los instaladores, archivos de configuración, migraciones de bases de datos y documentación de implementación para pasar a la gente que realizan la implementación real, todo ello sin probar en un entorno que parece producción o puesta en escena.
- Existe poca colaboración, si es que existe, entre el equipo de desarrollo y las personas que realmente realizan implementaciones para crear esta garantía.

_ Una vez que la aplicación se implementa en staging, es común que aparezcan nuevos errores. Desafortunadamente, a menudo no hay tiempo para arreglarlos todos porque el plazo se acerca rápidamente y, en esta etapa del proyecto, diferir la fecha de lanzamiento es inaceptable. De modo que los errores más críticos se corrigen apresuradamente y una lista de los defectos conocidos son almacenados por el director del proyecto para su custodia, para ser despriorizados hasta que comience el trabajo en la próxima versión. A veces puede ser incluso peor que esto. Aquí hay algunas cosas que pueden exacerbar los problemas asociados con una liberación:

- Cuando se trabaja en una nueva aplicación, el primer release probablemente sea el más problemático.
- Cuanto más largo sea el ciclo de release, más tiempo tendrá que hacer el equipo de desarrollo suposiciones incorrectas antes de que ocurra la implementación, y más tiempo tomará para arreglarlos.
- En organizaciones grandes donde el proceso de release se divide en diferentes grupos tales como desarrollo, DBA, operaciones, pruebas, etc., el costo de la coordinación puede ser enorme, a veces paralizando el proceso de release en el infierno de los tickets.
- Cuanto mayor sea la diferencia entre los entornos de desarrollo y de producción, menos realistas son los supuestos que deben hacerse durante desarrollo.

_ La solución a todo esto es:

- Integrar las actividades de prueba, implementación y lanzamiento (testing, deployment y release) en el proceso de desarrollo. Conviértalos en una parte normal y continua del desarrollo para que cuando esté listo para lanzar su sistema a producción haya poco o ningún riesgo, porque lo has ensayado en muchas ocasiones diferentes parecida a la producción de entornos de prueba.
- Asegurarse que todos los involucrados en el proceso de entrega de software, desde la compilación y el lanzamiento, del equipo de testers a desarrolladores, trabajen juntos desde el inicio del proyecto.



<u>Administración manual de la configuración en producción:</u> muchas organizaciones gestionan la configuración de sus entornos de producción a través de un equipo de operaciones. Si se necesita un cambio, como un cambio en la configuración de conexión de base de datos o un aumento en el número de subprocesos en un grupo de subprocesos en un servidor de aplicaciones, luego se lleva a cabo manualmente en los servidores de producción.

_ Los signos de este antipatron son:

- Habiendo desplegado con éxito muchas veces a la puesta en escena, el despliegue en producción falla.
- El equipo de operaciones tarda mucho en preparar un entorno para un lanzamiento.
- La configuración del sistema se realiza modificando la configuración directamente en los sistemas de producción.

_ La solución a todo esto es:

- Todos los aspectos de cada uno de sus entornos de prueba, preparación y producción, específicamente la configuración de cualquier elemento de terceros de su sistema, debe ser aplicado desde el control de versiones a través de un proceso automatizado.
- Una de las prácticas clave es la gestión de la configuración: significa ser capaz de recrear repetidamente cada pieza de infraestructura utilizada por su aplicación. Debería poder recuperar (rollback) el entorno de producción exactamente, preferiblemente de forma automatizada.
- Cada cambio hecho a producción debe ser registrado y auditable.

_ Ahora bien, nos preguntamos si esto se puede mejorar, y respondemos que el lanzamiento de software puede, y debe, ser de bajo riesgo, proceso frecuente, barato, rápido y predecible. Para lograrlo, nuestro objetivo como profesionales de software es brindar software a los usuarios lo más rápido posible. La velocidad es esencial porque hay un costo de oportunidad asociado a la no entrega o entrega demorada del software. Solo puede comenzar a obtener un retorno de su inversión una vez que se lanza su software. Por lo tanto, uno de nuestros dos objetivos principales es encontrar formas de reducir el tiempo del ciclo, el tiempo que lleva decidir hacer un cambio, ya sea una corrección de errores o una función, para que esté disponible para los usuarios y obtener un feedback. Una parte importante de la utilidad es la calidad. Nuestro software debe ser apto para su propósito.

_ Por lo tanto, para refinar ligeramente nuestro objetivo, queremos encontrar formas de ofrecer alta calidad, software valioso de una manera eficiente, rápida y confiable. Y para lograr estos objetivos (tiempos de ciclo cortos y de alta calidad), debemos realizar cambios frecuentes y automatizados lanzamientos de nuestro software.

- Automatizado:
  - Si el proceso de compilación, implementación, prueba y lanzamiento no está automatizado, no es repetible. Cada vez que se haga será diferente, debido a cambios en el software, la configuración del sistema, los entornos, y el proceso de liberación.
  - Dado que los pasos son manuales, son propensos a errores, y no hay forma de revisar exactamente lo que se hizo.
  - Lanzar software es con demasiada frecuencia un arte; debería ser una disciplina de ingeniería.
- Frecuente:
  - Si las liberaciones son frecuentes, el delta entre liberaciones será pequeña. Esto reduce significativamente el riesgo asociado con la liberación y hace que sea mucho más fácil retroceder.
  - Los lanzamientos frecuentes también conducen a retroalimentación, de hecho, la necesitan.

_ La retroalimentación es esencial para releases frecuentes y automatizados. Hay tres criterios para que los comentarios sean útiles:

1) Cualquier cambio, del tipo que sea, debe desencadenar el proceso de retroalimentación: una aplicación de software que funcione se puede descomponer de manera útil en cuatro componentes: código ejecutable, configuración, entorno de host y datos. Si alguno de ellos cambia, puede provocar un cambio en el comportamiento de la aplicación.
- El código ejecutable cambia cuando se realiza un cambio en el código fuente. Cada vez que se realiza un cambio en el código fuente, el binario resultante debe construirse y probarse. Para ganar control sobre este proceso, construyendo y probando el binario debe ser automatizado. La práctica de construir y probar su aplicación en cada registro se conoce como integración continua.
- Todo lo que cambie entre entornos debe capturarse como configuración información. Cualquier cambio en la configuración de una aplicación, en cualquier entorno, debe ser probado. Si el software debe ser instalado por los usuarios, las posibles opciones de configuración deben probarse en un rango representativo de sistemas de ejemplo.

_ El proceso de retroalimentación implica probar cada cambio en una moda acoplada, en la medida de lo posible. Las pruebas variarán según el sistema, pero normalmente incluirán al menos las siguientes comprobaciones:

- El proceso de creación del código ejecutable debe funcionar. Esto verifica que la sintaxis de su código fuente es válida.
- Deben aprobarse las pruebas unitarias del software. Esto verifica que su aplicación el código se comporta como se esperaba. El software debe cumplir ciertos criterios de calidad, como la cobertura de prueba y otras métricas específicas de tecnología.
- Deben pasar las pruebas de aceptación funcional del software. Esto comprueba que su la aplicación se ajusta a sus criterios de aceptación empresarial, que ofrece el valor comercial que se pretendía.
- Las pruebas no funcionales del software deben pasar. Esto verifica que la aplicación funciona suficientemente bien en términos de capacidad, disponibilidad, seguridad, y así sucesivamente para satisfacer las necesidades de sus usuarios.
- El software debe pasar por pruebas exploratorias y una demostración para el cliente y una selección de usuarios.

2) La retroalimentación debe entregarse lo antes posible: la clave para una retroalimentación rápida es la automatización. Con procesos totalmente automatizados, la única restricción es la cantidad de hardware que puede lanzar al problema. Sin embargo, el deployment pipeline requiere muchos recursos, especialmente una vez que tenga un conjunto completo de pruebas

automatizadas. Podemos caracterizar de la siguiente manera las pruebas en esta etapa:

- Commit-stage:
  - Tests de ejecución rápida.
  - Que sean independientes del entorno (environment).
  - Tener un test coverage alto.
  - Si estos tests fallan no se puede releasear el software.
- Tests funcionales:
  - Son más lentos.
  - Si fallan aún se puede elegir releasear el software con known bugs.
  - Deben correr en un entorno lo más parecido a producción posible.
- Tests no funcionales:
  - Requieren más tiempo.
  - Requieren más recursos.

3) El equipo de desarrollo debe recibir comentarios y luego actuar en consecuencia: es esencial que todos los involucrados en el proceso de entrega de software estén involucrado en el proceso de retroalimentación. Ser capaz de reaccionar a la retroalimentación también significa transmitir información. Utilizando paneles grandes y visibles (que no necesitan ser electrónicos) y otras notificaciones es fundamental para garantizar que la retroalimentación sea, de hecho, retroalimentada y el paso final en la cabeza de alguien. Finalmente, la retroalimentación no es buena a menos que se actúe en consecuencia. Cuando hay que hacer algo, es responsabilidad de todos. equipo para detener lo que están haciendo y decidir un curso de acción. Sólo una vez esto se hace si el equipo continúa con su trabajo. Los beneficios de este enfoque son:

- Proceso de release, repetible, confiable y predecible.
- Reducción del ciclo de release y aumento de la calidad.
- Le da más poder al team:
  - Se puede elegir una versión deseada y deployarla en un entorno deseado.
  - No hay que esperar un “buen release”.
- Reduce errores y el estrés, ya que las tareas se hacen triviales, repetibles y predecibles.

Candidato a release

_ Es el proceso de construcción, implementación y prueba que aplicamos al cambio que valida que puede ser lanzado o no. Si bien, cualquier cambio puede dar lugar a un artefacto que se puede liberar a los usuarios, no es recomendable comenzar de esa manera. Cada cambio debe evaluarse para determinar su idoneidad. Si el producto se encuentra libre de defectos, y cumple con los criterios de aceptación establecido por el cliente, entonces puede ser liberado. La mayoría de los enfoques para lanzar software

identifican candidatos de lanzamiento al final del proceso. Entonces, decimos que cada check in debería ser un release candidate.

_ A continuación, tenemos los release candidates detectados al final del proceso de desarrollo:

Principios de la entrega de software

Cree un proceso confiable y repetible para lanzar software: liberar el software debería ser fácil porque ha probado todas las partes del proceso de liberación cientos de veces antes. La repetitividad y la fiabilidad se derivan de dos principios: automatice casi todo y conserve todo lo que necesita para construir, implementar, probar, y libere su aplicación en control de versiones. La implementación de software implica, en última instancia, tres cosas:

- Aprovisionamiento y gestión del entorno en el que su aplicación ejecutar (configuración de hardware, software, infraestructura y servicios externos).
- Instalar la versión correcta de la aplicación.
- Configurar su aplicación, incluyendo cualquier dato o estado que requiera.

Automatizar casi todo: Como el despliegue de la aplicación y las validaciones. Pero hay algunas cosas que es imposible automatizar:

- Las pruebas exploratorias se basan en testers experimentados.
- Demostraciones de software funcional a representantes de su comunidad de usuarios no puede ser realizada por computadoras.
- Aprobaciones de cumplimiento, por definición, requieren la intervención humana.

_ La mayoría de los equipos de desarrollo no automatizan su proceso de lanzamiento porque parece una tarea tan abrumadora. Es más fácil hacer las cosas manualmente.

Mantener todo bajo control de la configuración: todo lo que necesita para construir, implementar, probar y lanzar su aplicación debe mantenerse en alguna forma de almacenamiento versionado. Esto incluye documentos de requisitos, scripts de prueba, casos de prueba automatizados, scripts de configuración de red, implementación scripts, scripts de creación, actualización, degradación e inicialización de bases de datos, aplicaciones scripts de configuración de pila de cationes, bibliotecas, cadenas de herramientas, documentación técnica, y así. Debería ser posible que un nuevo miembro del equipo se siente en una nueva estación de trabajo, consulte el repositorio de control de revisiones del proyecto y ejecute un solo comando para construir e implementar la aplicación en cualquier entorno accesible, incluido el puesto de trabajo de desarrollo local.

Si “duele”, hazlo más frecuentemente y trae el dolor antes: tenemos lo siguiente:

- Si la integración es a menudo un proceso muy doloroso, debemos hacerlo para cada cambio.
- Si el testing es un proceso doloroso, que ocurre justo antes del lanzamiento, no lo debemos hacer al final. En cambio, debemos hacerlo continuamente desde el inicio del proyecto de forma continua.
- Si lanzar software es doloroso, debemos intentar lanzarlo cada vez que alguien verifique en un cambio, que pasa todas las pruebas automatizadas.
- Si crear la documentación de la aplicación es complicado, debemos hacerlo a medida que desarrollamos nuevas funciones, en lugar de dejarlo para el final.

Build quality in: cuanto antes detecte defectos, más barato son para arreglar. Los defectos se solucionan de forma más económica si nunca se registran en control de versiones en primer lugar. Las pruebas automatizadas integrales y la implementación automatizada están diseñadas para detectar defectos lo antes posible en el proceso de entrega. El siguiente paso es arreglarlos. Hay otros dos corolarios de "calidad de construcción":

- Las pruebas no son una fase, y ciertamente no una que comience después de la fase de desarrollo. Si la prueba es dejada hasta el final, será demasiado tarde. No habrá tiempo para arreglar los defectos.
- Las pruebas tampoco son el dominio, pura o incluso principalmente, de los probadores. Todos en el equipo de entrega es responsable de la calidad de la aplicación todo el tiempo.

Done significa released: para algunos equipos de entrega ágiles, "hecho" significa lanzado a producción. Esta es la situación ideal para un proyecto de desarrollo de software. Sin embargo, no siempre es práctico utilizar esto como una medida de hecho. La versión inicial de un sistema de software puede tomar un tiempo antes de que esté en un estado en el que los usuarios externos reales se beneficien de eso. Así que volveremos a marcar la siguiente mejor opción y diremos que una funcionalidad se "hace" una vez que se ha exhibido con éxito, es decir, demostrado y probado por representantes de la comunidad de usuarios, a partir de un entorno de producción.

Todos son responsables del proceso de release: idealmente, todos dentro de una organización están alineados con sus objetivos, y las personas trabajen juntos para ayudar a cada uno a encontrarlos. Al final, el equipo tiene éxito o fracasa como equipo, no como individuos. Sin embargo, en demasiados proyectos la realidad es que los desarrolladores arrojan su trabajo por encima de la pared a los testers. Entonces los testers lanzan trabajo sobre el muro al equipo de operaciones en el momento del lanzamiento. Cuando algo sale mal la gente pasa tanto tiempo culpándose unos a otros como arreglando los defectos que inevitablemente surgen de un enfoque tan aislado. Empiece por involucrar a todos en el proceso de entrega juntos desde el iniciar un nuevo proyecto y asegurarse de que tengan la oportunidad de comunicarse de forma regular y frecuente. Iniciar un sistema donde todos puedan ver, de un vistazo, el

estado de la aplicación, sus diversas compilaciones, qué pruebas han pasado y el estado de los entornos en los que se pueden implementar.

- Este es uno de los principios centrales del movimiento DevOps, el cual se centra en alentar la colaboración entre todos los involucrados en la entrega de software con el fin de lanzar software valioso de forma más rápida y fiable.

Mejora continua: vale la pena enfatizar que el primer lanzamiento de una aplicación es solo la primera etapa en su vida. Todas las aplicaciones evolucionan y seguirán en las próximas versiones. Es importante que su proceso de entrega también evoluciona con él. Todo el equipo debe reunirse periódicamente y realizar una retrospectiva sobre el proceso de entrega. Esto significa que el equipo debe reflexionar sobre lo que ha sucedido, es decir, lo que ha salido bien y lo que ha ido mal, y discutir ideas sobre cómo mejorar las cosas. Es fundamental que todos los miembros de la organización participen en este proceso.

Integración continua

Proyectos sin integración continua

_ Hay un estado “extraño” pero no poco común en los proyectos de software. La aplicación está en un estado en el que no funciona aún. Esto suele pasar mayormente en proyectos desarrollados por teams grandes, donde cada uno desarrolla una porción del proyecto, y a nadie le interesa tratar de correr la aplicación como un todo hasta que esté terminada. También, en proyectos con long-lived branches, en proyectos que los tests de aceptación se dejan para el final, y se suele planear fases de integración largas, muchas veces porque no se sabe cuánto puede tardar.

Proyectos donde se usa la integración continua

_ Descrito en 1999 por Kent Beck en Xtreme Programing, significando un cambio de paradigma:

- Sin CI: software “roto” hasta que alguien prueba lo contrario durante la fase de testing o integración.
- Con CI: se verifica que el software funciona con cada cambio, y si se rompe se arregla inmediatamente

_ El objetivo de CI consiste en tener la aplicación en un estado funcional constantemente. Antes de comenzar debemos tener:

Version control: todo tiene que estar en un sistema de control de versiones.

Builds automatizados: los scripts de build tiene que ser tratado como código base del producto.

Compromiso del equipo de desarrollo:

- Entender que es una práctica, no una tool (herramienta).
- Requiere disciplina por parte del equipo.
- Hacer varios check in frecuentes y pequeños.
- Compromiso de arreglar el build apenas se rompe.

Pasos, una vez que el cambio esté listo para check-in:

1. Fijarse si el build está corriendo: si falla, arreglarlo con el resto del team.
2. Cuando el build y los tests pasan: actualizar el codigo en el ambiente local (local environment).
3. Ejecutar el build y los test cases localmente.
4. Si el build y los test pasan localmente, hacer check in.
5. Esperar que el build server buildee con nuestros cambios.
6. Si falla, arreglar el problema inmediatamente: volver al paso 3.
7. Si el build pasa seguir con la próxima tarea.

Pre-requisitos:

- Hacer check in regularmente:
  - A trunk o main o master.
  - Al menos 2 veces al día.
  - Origina cambios más pequeños y es menos probable que rompan el build.
  - Se tiene una versión reciente que funciona a la cual revertir los cambios
- Crear un test suite exhaustivo:
  - Unit tests.
  - Component tests.
  - Acceptance tests.
- Mantener el build y el proceso de testing corto:
  - El límite es 10 minutos pero 5 minutos es mejor.
  - Parece contradecir el anterior, diferentes tipos de tests en diferentes etapas del pipeline.
- Usar workspaces privados
  - Los desarrolladores deben poder compilar y correr los tests en sus entornos privados.
  - Entornos de desarrollo compartidos deberían ser la excepción.

Operación básica: esencialmente hay 2 componentes:

- Un servicio que ejecuta un “workflow” a intervalos regulares.
- Un servicio que provee una vista de los resultados del build y los tests, notifica el estado de los builds, provee acceso a los instaladores, y reportes.

Prácticas esenciales:

- No hacer commit en un build roto.
- Correr los tests localmente antes de hacer commit o usar al building server para que lo haga:
  - Update workspace, run local build, run local tests, and commit if success.
- Esperar que termine el build en main después del commit antes de continuar: hay que asegurarse que pase y sino arreglarlo o revertir el cambio.
- No irse a casa con un build roto: para evitar irse tarde hacer varios check in frecuentes y más temprano, no a último momento.
- Siempre estar preparado para revertir a la versión anterior: si no podemos arreglar el problema por cualquier razón, hay que revertir el cambio.
- Time-box fixing antes de revertir el cambio.
- No comentar tests cases que fallan.
- Aceptar la responsabilidad de todas las roturas por nuestros cambios.
- Test-Driven development.

Deployment pipeline

_ La integración continua es un gran paso adelante en productividad y calidad para la mayoría de los proyectos que lo adoptan.

- Asegura que los equipos que trabajan juntos para crear sistemas grandes y complejos pueden hacerlo con un mayor nivel de confianza y control de lo que se puede lograr sin él.
- Asegura que el código que creamos, como equipo, funciona brindándonos retroalimentación rápida sobre cualquier problema que podamos introducir con los cambios que cometemos.
- Se centra principalmente en afirmar que el código se compila con éxito y pasa un conjunto de pruebas unitarias y de aceptación.

_ Sin embargo, CI no es suficiente. CI se centra principalmente en equipos de desarrollo. La salida del sistema CI normalmente forma la entrada al proceso de prueba manual y de allí al resto de la versión proceso. Gran parte del desperdicio en la liberación de software proviene del progreso de software a través de pruebas y operaciones:

- Los equipos de operaciones y construcción esperan documentación o arreglos.
- Los testers esperan versiones "buenas" del software, la creación de “buenos” entornos y la instalación de software.
- Los equipos de desarrollo reciben informes de errores semanas después de que el equipo haya comenzado la nueva funcionalidad. Descubrir, hacia el final del proceso de desarrollo, que la arquitectura de la instalación no admitirá los requisitos no funcionales del sistema.

_ Esto conduce a un software que no se puede implementar porque ha tardado tanto en un entorno de producción y con errores porque el ciclo de retroalimentación entre el equipo de desarrollo y el equipo de pruebas y operaciones es tan largo. Hay varias mejoras incrementales en la forma en que se entrega el software que producirá beneficios inmediatos, como enseñar a los desarrolladores a escribir software listo para producción, ejecutando CI en sistemas similares a producción e instituyendo equipos multifuncionales. Sin embargo, aunque prácticas como estas ciertamente mejorarán importa, todavía no le dan una idea de dónde están los cuellos de botella en el proceso de entrega o cómo optimizarlos.

_ La solución es adoptar un enfoque más holístico e integral para brindar software. Hemos abordado los problemas más amplios de la gestión de la configuración y automatizar grandes extensiones de nuestros procesos de construcción, implementación, prueba y lanzamiento. Esto crea un poderoso circuito de retroalimentación: dado que es tan simple su aplicación en entornos de prueba, su equipo recibe comentarios rápidos tanto en el código como en el proceso de implementación. Con lo que terminamos es (en lenguaje Lean) un sistema de atracción (Pull system):

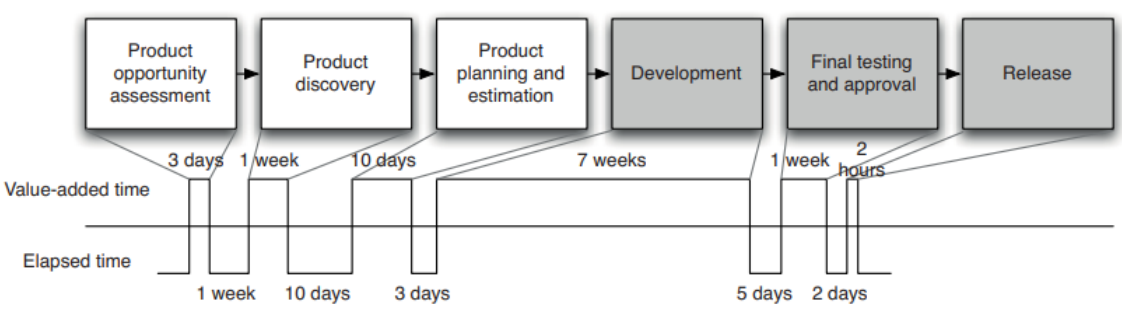
- Implementación de equipos de prueba se integra en los propios entornos de prueba, con solo pulsar un botón.
- Operaciones puede implementar compilaciones en entornos de ensayo y producción con solo botón.
- Los desarrolladores pueden ver qué builds han pasado por qué stage y qué problemas se encontraron.
- Los gerentes pueden vigilarlo mediante métricas como tiempo de ciclo, rendimiento y calidad del código.

_ Como resultado, todos en el proceso de entrega obtienen dos cosas, por un lado acceso a las cosas que necesitan cuando las necesitan, y por otro, visibilidad en el proceso de lanzamiento para mejorar la retroalimentación de modo que los cuellos se pueden identificar, optimizar y eliminar. Esto conduce a un proceso de entrega que no solo es más rápido sino también más seguro.

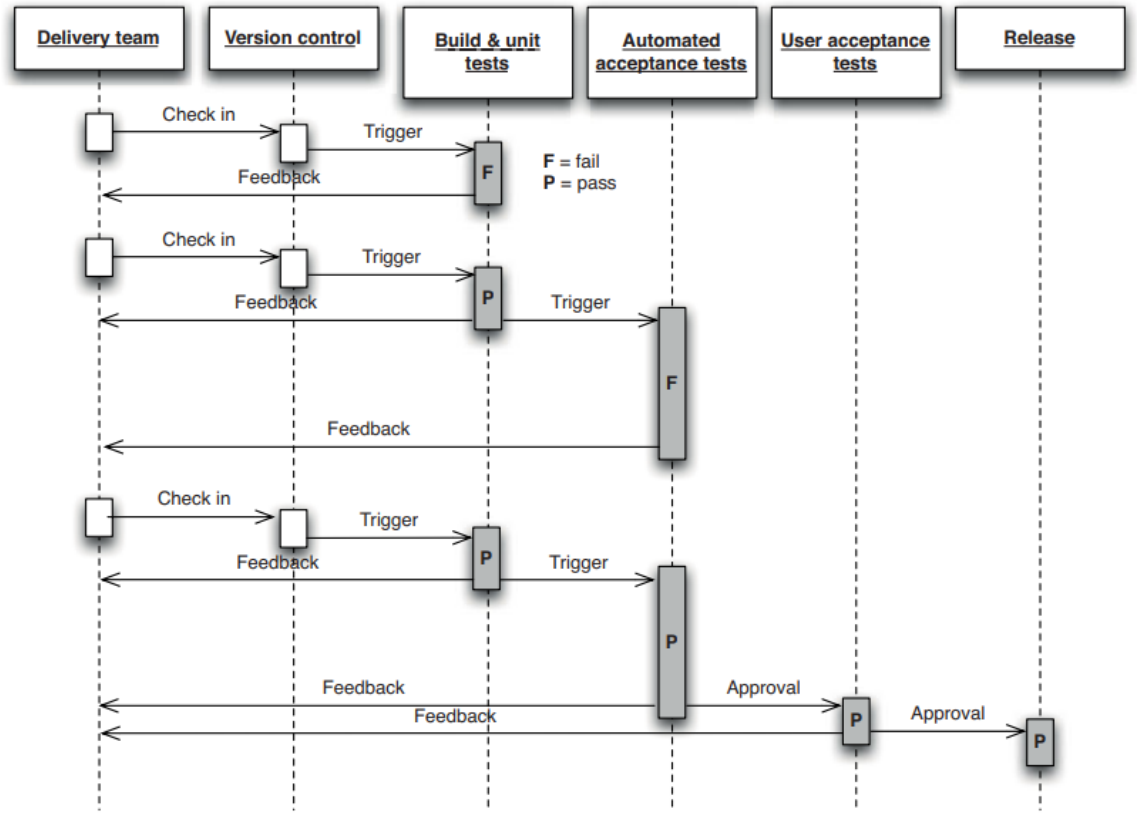
Definición

_ Deployment pipeline es una manifestación automatizada del proceso para llevar el software del control de versiones a manos de sus usuarios.

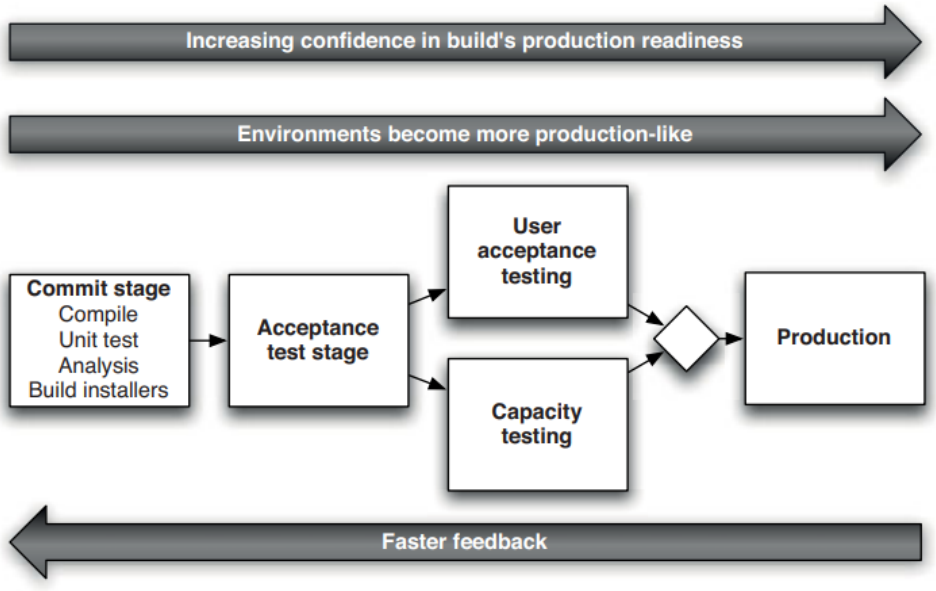
_ Cada cambio en nuestro software pasa por un proceso complejo en su camino hacia ser liberado. Ese proceso implica la construcción del software, seguido por el progreso de estas compilaciones a través de múltiples etapas de prueba e implementación. Esta, a su vez, requiere la colaboración entre muchos individuos, y quizás varios equipos.



_ Una forma de entender el deployment pipeline y cómo se mueven los cambios a través de ella es visualizarla como un diagrama de secuencia, como se muestra en el siguiente grafico:



- Entrada: una versión de VCS.
- El proceso lleva al build a través de una serie de etapas.
- En cada etapa se evalúa al build desde diferentes perspectivas.
- Cuanto más lejos llega un build, mayor confianza en la calidad del build.
  - Mayor la cantidad de recursos invertidos en ese build.
  - Environment más parecido a production.
- Objetivo: eliminar builds defectuosos lo más rápido posible para obtener feedback rápido y asi los builds defectuosos no continúan en el pipeline.



Consecuencias:

- Primero:
  - No se deployan en production releases que no fueron “ensayados”.
  - Se evitan bugs de regresión, incluso en fixes de emergencia.
  - Se mitigan riesgos relacionados al entorno de ejecución.
- Segundo:
  - Releases, rápidos, confiables y repetibles.
  - Se pueden hacer releases más frecuentes.
  - Se puede volver atrás o avanzar si se lo desea.
  - Los releases no tienen riesgo.

_ Para lograr este estado “envidiable”:

- Automatizar un conjunto de tests que prueben que nuestro reléase candidate cumple con su propósito.
- Automatizar deployment a todos los entornos, test, staging, producción.
- Remover toda tarea manual.
- Las etapas que se deben automatizar dependen de cada proceso, pero se pueden identificar algunos comunes.

Estructura

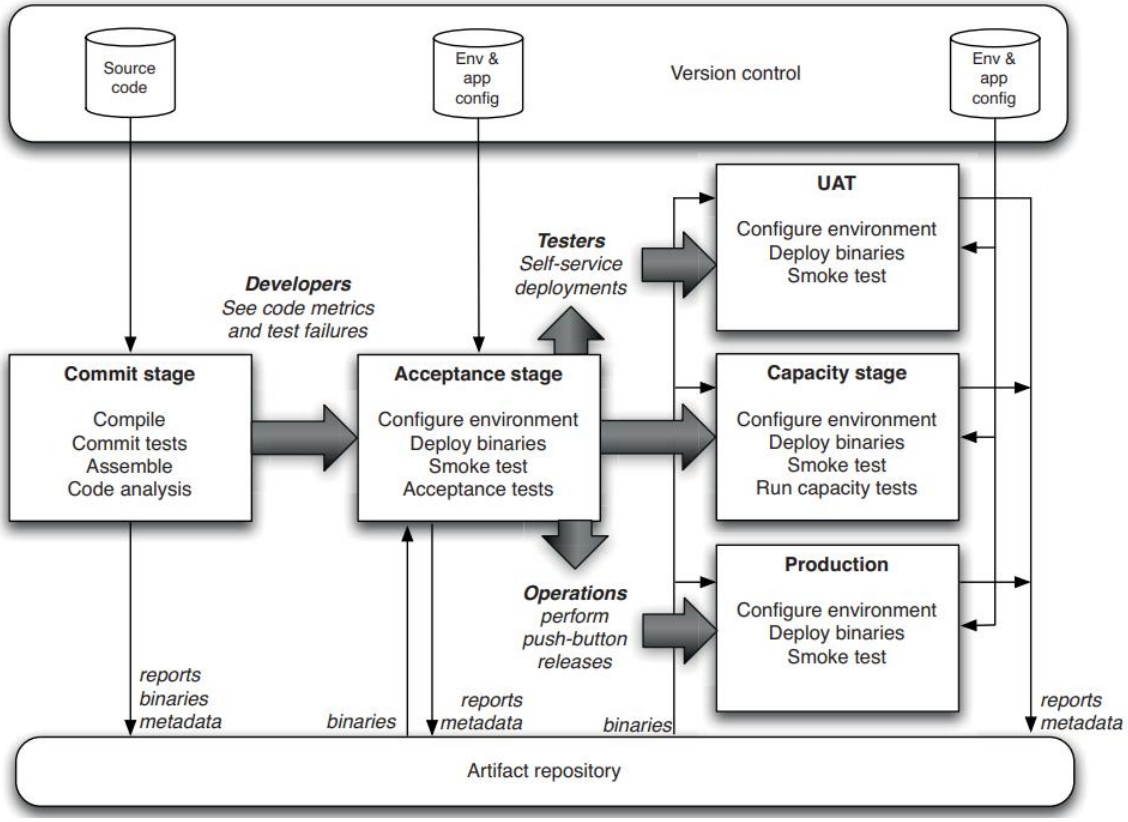
Commit stage: afirma que el sistema funciona a nivel técnico. Eso compila, pasa un conjunto de pruebas automatizadas (principalmente a nivel de unidad) y ejecuta análisis estático de código.

Automated Acceptance test stage: afirman que el sistema funciona en nivel funcional y no funcional, que conductualmente satisface las necesidades de sus usuarios y las especificaciones del cliente.

Manual test stage: estas afirman que el sistema es utilizable y cumple con sus requerimientos, detectar cualquier defecto no detectado por pruebas automatizadas y verificar que proporciona valor a sus usuarios. Estas etapas normalmente pueden incluir exploratorias, entornos de prueba, entornos de integración y UAT (aceptación del usuario pruebas).

Release stage: entrega el sistema a los usuarios, ya sea como software empaquetado o desplegándolo en un entorno de producción o de ensayo (un entorno de ensayo entorno es un entorno de prueba idéntico al entorno de producción).

_ A continuación, vemos un deployment pipeline típico:

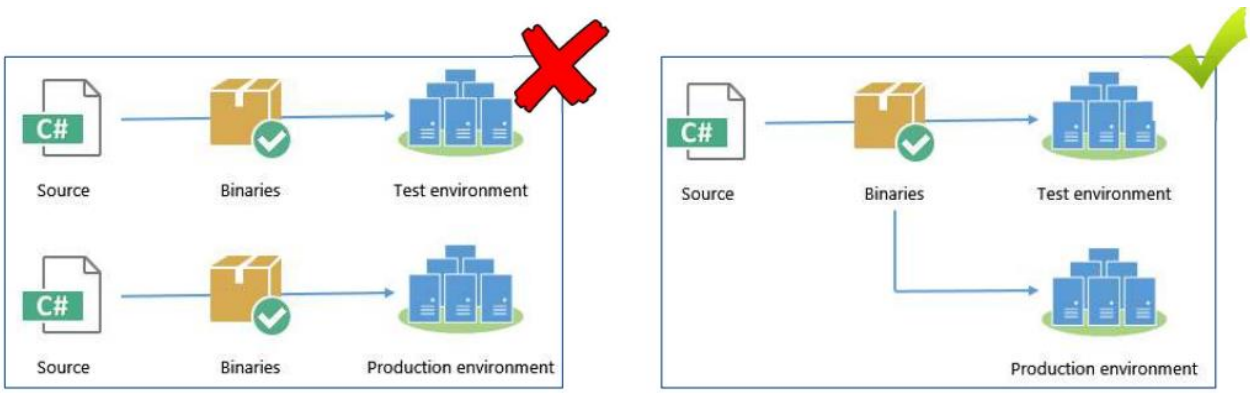


Practicas

Solo cree sus archivos binarios una vez: muchos sistemas de compilación utilizan el código fuente contenido en el sistema de control de versiones como la fuente canónica de muchos pasos. Cada vez que compila el código, corre el riesgo de introducir alguna diferencia. Este antipatrón viola dos principios importantes:

- El primero es mantener el diseño deployment pipeline eficiente, por lo que el equipo recibe comentarios lo antes posible. Recomendar el apilamiento viola este principio porque lleva tiempo, especialmente en sistemas grandes.
- El segundo principio es construir siempre sobre lo conocido. Los binarios que se implementan en producción deben ser exactamente los mismos que los que pasaron por el proceso de prueba de aceptación.

_ Si volvemos a crear binarios, corremos el riesgo de que se introduzca algún cambio entre la creación de los binarios y su lanzamiento. Una vez que hayamos creado nuestros binarios, los reutilizaremos sin volver a crearlos en el punto de uso.

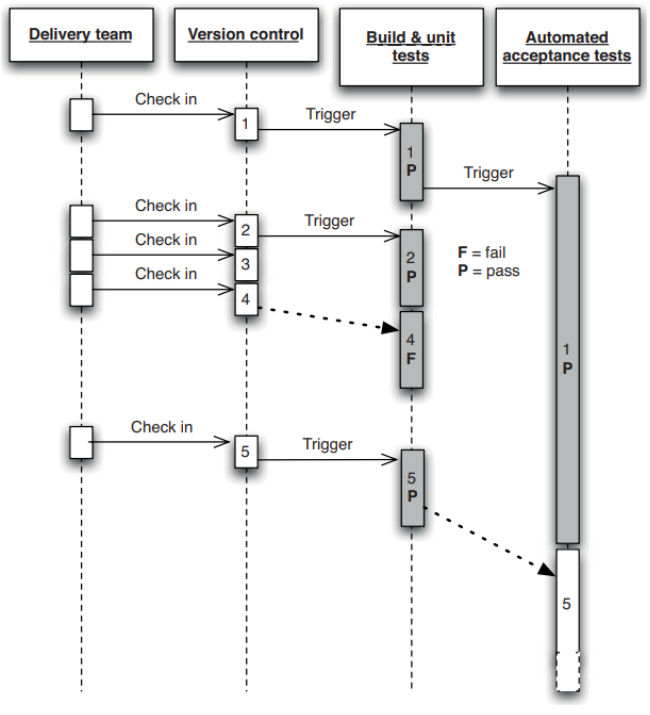


Deploya de la misma manera en todos los entornos: es esencial utilizar el mismo proceso para implementar en todos los entornos, ya sea la estación de trabajo de un desarrollador o analista, un entorno de prueba o producción, en para garantizar que el proceso de construcción e implementación se pruebe de manera efectiva. Cada entorno es diferente de alguna manera, tendrá una dirección IP única, pero a menudo hay otras diferencias. Esto no significa que deba utilizar un script de implementación diferente para cada entorno. En su lugar, mantenga separadas las configuraciones que son únicas para cada entorno.

Smoke-test todos los deployments: cuando implemente su aplicación, debe tener un script automatizado que realiza una prueba de humo para asegurarse de que esté en funcionamiento. Su prueba de humo también debe verificar que cualquier los servicios de los que depende su aplicación están en funcionamiento, como una base de datos, bus de mensajería o servicio externo. La prueba de humo, o prueba de despliegue, es probablemente la prueba más importante para escribir una vez que tenga un conjunto de pruebas unitarias en funcionamiento; de hecho, podría decirse que es incluso más importante. Le da la confianza de que su aplicación realmente se ejecuta.

Deploya en una copia de producción: el otro problema principal que experimentan muchos equipos al comenzar a funcionar es que su producción el entorno es significativamente diferente de su entorno de prueba y desarrollo. Para tener un buen nivel de confianza en que la publicación en vivo realmente funcionará, necesita hacer sus pruebas e integración continua en entornos que son tan similar a su entorno de producción.

Cada cambio debería propagarse por el pipeline instantáneamente: antes de que se introdujera la integración continua, muchos proyectos ejecutaban varias partes de su proceso fuera de una programación. El proceso de implementación toma un enfoque diferente: la primera etapa debe activarse en cada registro, y cada etapa debe activar la siguiente inmediatamente después de completar con éxito. Esto no siempre es posible cuando los desarrolladores (especialmente en grandes equipos) se registran con mucha frecuencia, dado que las etapas de su proceso pueden tomar una cantidad de tiempo no insignificante. El problema se muestra en la siguiente imagen:



_ En este ejemplo, alguien verifica un cambio en el control de versiones, creando versión 1. Esto, a su vez, desencadena la primera etapa en la tubería (pruebas unitarias y de compilación). Esto pasa y activa la segunda etapa: las pruebas de aceptación automatizadas. Luego, alguien verifica otro cambio, creando la versión 2. Esto activa el construir y realizar pruebas unitarias de nuevo. Sin embargo, a pesar de que han pasado, no pueden activar una nueva instancia de las pruebas de aceptación automatizadas, ya que ya están corriendo.

_ Mientras tanto, se han producido dos registros más en rápida sucesión. Sin embargo, el sistema de CI no debería intentar construir ambos (si siguió esa regla), y los desarrolladores continuaron registrándose al mismo ritmo, las compilaciones ir más y más por detrás de lo que los desarrolladores están haciendo actualmente.

_ En cambio, una vez que finaliza una instancia de las pruebas unitarias y de compilación, el sistema CI comprueba si hay nuevos cambios disponibles y, de ser así, se basa en los más recientes conjunto disponible; en este caso, la versión 4. Supongamos que esto rompe las pruebas de construcción y unidad etapa.

Si cualquier parte del pipeline falla detén el pipeline: si falla una implementación en un entorno, todo el equipo es dueño de ese fracaso. Deben detenerse y arreglarlo antes de hacer cualquier otra cosa.

Diseño en las metodologías ágiles

Introducción

Conceptos

_ Las empresas operan en un entorno global que cambia rápidamente. En ese sentido, deben responder frente a nuevas oportunidades y mercados, al cambio en las condiciones económicas, así como al surgimiento de productos y servicios competitivos.

Software: es parte de casi todas las operaciones industriales, de modo que el nuevo software se desarrolla rápidamente para aprovechar las actuales oportunidades, con la finalidad de responder ante la amenaza competitiva. En consecuencia, la entrega y el desarrollo rápidos son por lo general el requerimiento fundamental de los sistemas de software. De hecho, muchas empresas están dispuestas a negociar la calidad del software y el compromiso con los requerimientos, para lograr con mayor celeridad la implementación que necesitan del software.

_ Tenemos otras definiciones:

- Programas de cómputo y documentación asociada.
- Según el estándar 729 de la IEEE, es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación.

- Término genérico que se refiere a una colección de datos e instrucciones de computadora qué le dicen a la computadora cómo hacer su trabajo. En contraste, el hardware es el que realiza el trabajo.

Ingeniería de software: la IEEE la define como la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software.

- Sistemático -> normas.
- Procedimientos disciplinado -> orden y subordinación entre los miembros.
- Cuantificable -> medible.

Proceso de software: es una serie de actividades relacionadas que conduce a la elaboración de un producto de software. Requiere la definición de los roles y la especificación de las actividades o pasos a seguir. Existen muchos diferentes procesos de software, pero todos deben incluir cuatro actividades que son fundamentales para la ingeniería de software:

1. Especificación del software: tienen que definirse tanto la funcionalidad del software como las restricciones de su operación.
2. Diseño e implementación del software: debe desarrollarse el software para cumplir con las especificaciones.
3. Validación del software: hay que validar el software para asegurarse de que cumple lo que el cliente quiere.
4. Evolución del software: el software tiene que evolucionar para satisfacer las necesidades cambiantes del cliente.

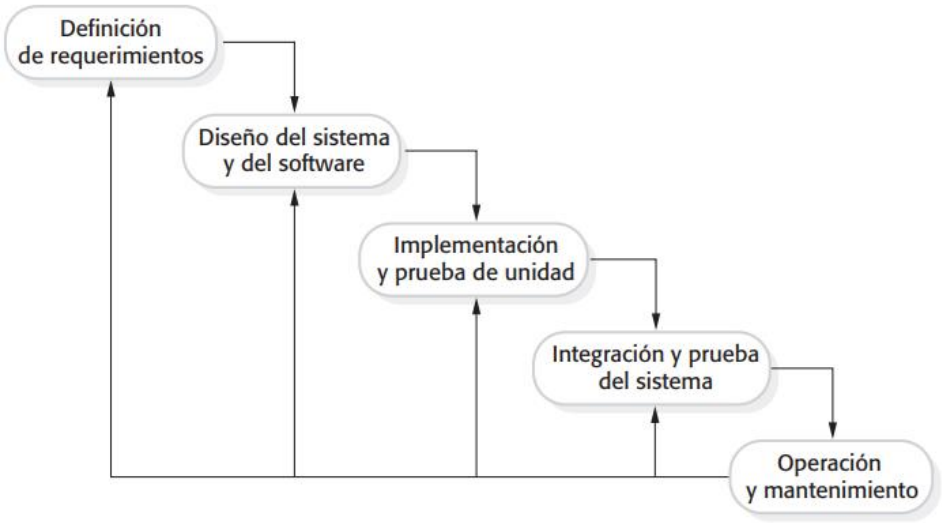
_ Respecto al como haríamos el planning, debemos tener en cuenta:

1. Definir objetivos, actividades para alcanzarlo y criterios de verificación y control en base a los requerimientos.
2. Estimar esfuerzo, tiempo y, por supuesto, costo.
3. Planificación y organización de tareas y recursos.
4. Analizar posibles riesgos.

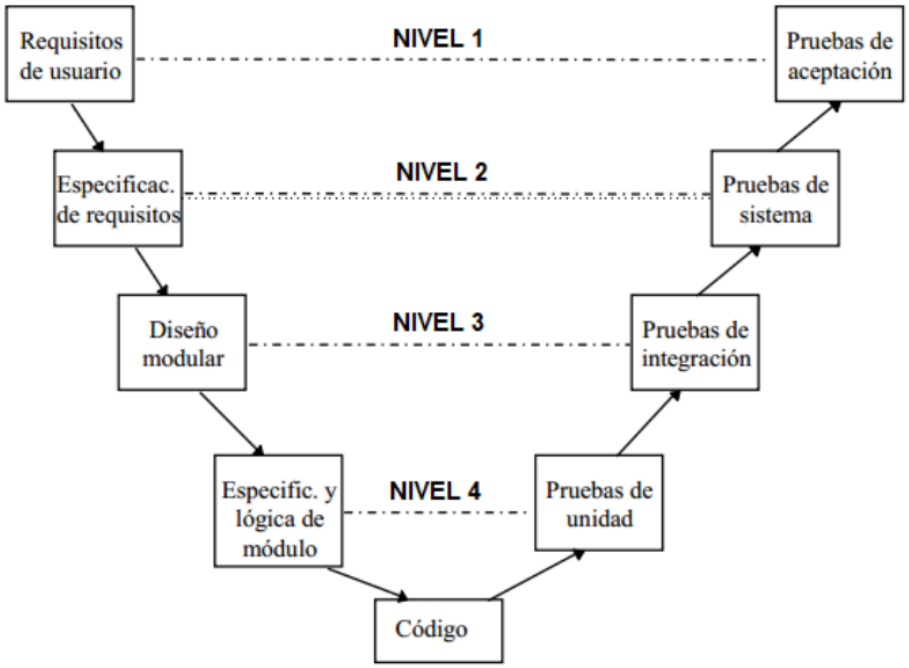
Modelo de proceso de software: representación simplificada del proceso de software que pueden utilizarse como base para crear procesos de software específicos. Cada modelo del proceso representa a otro desde una particular perspectiva y, por lo tanto, ofrece sólo información parcial acerca de dicho proceso. Tenemos los siguientes modelos:

- Modelo en cascada: este modelo presenta las siguientes desventajas:
  - No refleja realmente el proceso de desarrollo del software.
  - Se tarda mucho tiempo en pasar por todo el ciclo.
  - Perpetua el fracaso de la industria del software en su comunicación con el usuario final.
  - El mantenimiento se realiza en el código fuente.
  - Las revisiones de proyectos de gran complejidad son muy difíciles.

- Impone una estructura de gestión de proyectos.



- Modelo en V: esta estructura obedece al principio de que para cada fase del desarrollo debe existir un resultado verificable. Existe una fase correspondiente o paralela de verificación o validación.
  - Nivel 1: está orientado al cliente, y se compone del análisis de requisitos y especificaciones, se traduce en un documento de requisitos y especificaciones.
  - Nivel 2: se dedica a las características funcionales del sistema propuesto. Puede considerarse el sistema como una caja negra, y caracterizarla únicamente con aquellas funciones que son directa o indirectamente visibles por el usuario final, se traduce en un documento de análisis funcional.
  - Nivel 3: define los componentes hardware y software del sistema final, a cuyo conjunto se denomina arquitectura del sistema.
  - Nivel 4: es la fase de implementación, en la que se desarrollan los elementos unitarios o módulos del programa.



Las ventajas de este modelo son:

- Optimización de la comunicación entre las partes involucradas a través de términos y responsabilidades claramente definidos.
- Minimización de riesgos y mejor planificación a través de roles, estructuras y resultados fijos y predeterminados.
- Ahorro de costes gracias al procesamiento transparente a lo largo de todo el ciclo de vida del producto.
- En general, el modelo puede ayudar a evitar malentendidos y trabajo innecesario. También garantiza que todas las tareas se completen en el plazo y orden adecuado y mantiene los periodos de inactividad al mínimo.

Ahora bien, como desventajas tenemos:

- El modelo en cuatro niveles puede ser demasiado simple para mapear todo el proceso de desarrollo desde el punto de vista de los desarrolladores.
- Su estructura relativamente rígida permite una respuesta poco flexible a los cambios durante el desarrollo, y, por lo tanto, promueve un curso lineal del proyecto. Sin embargo, si el modelo se entiende y se utiliza correctamente, es posible utilizar el modelo V para el desarrollo ágil.

- Modelo secuencial: se recomienda usar estos modelos cuando los requerimientos se entienden por completo o cuando el software no cambiará (sistema embebido). Pero puede ser inflexible, se hace difícil responder al cambio de los requerimientos y es poco probable que se construya lo que el cliente realmente necesita.

Agile

Introducción

_ En la década de 1990 el descontento con estos enfoques engorrosos de la ingeniería de software, condujeron a algunos desarrolladores de software a proponer nuevos "métodos ágiles", los cuales permitieron que el equipo de desarrollo se enfocara en el software en lugar del diseño y la documentación.

_ El proceso ágil es el remedio universal para el fracaso en los proyectos de desarrollo de software. Las aplicaciones de software desarrolladas a través del proceso ágil tienen tres veces la tasa de éxito del método en cascada tradicional y un porcentaje mucho menor de demoras y sobrecostos. El software debería ser construido en pequeños pasos iterativos, con equipos pequeños y enfocados.

_ Estos procesos 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. Se dirigen a simplificar el proceso burocrático al evitar trabajo con valor dudoso a largo plazo, y a eliminar documentación que quizá nunca se emplee.

_ La filosofía detrás de los métodos ágiles se refleja en el manifiesto ágil, que acordaron muchos de los desarrolladores líderes de estos métodos. Este manifiesto afirma:

"Estamos descubriendo mejores formas para desarrollar software, al hacerlo y al ayudar a otros a hacerlo. Gracias a este trabajo llegamos a valorar:

> *A los individuos y las interacciones sobre los procesos y las herramientas*
>
> *Al software operativo sobre la documentación exhaustiva*
>
> *La colaboración con el cliente sobre la negociación del contrato*
>
> *La respuesta al cambio sobre el seguimiento de un plan*

Esto es, aunque exista valor en los objetos a la derecha, valoraremos más los de la izquierda."

_ Aunque todos esos métodos ágiles se basan en la noción del desarrollo y la entrega incrementales, proponen diferentes procesos para lograrlo. Sin embargo, comparten una serie de principios, según el manifiesto ágil y, por ende, tienen mucho en común.

Principios de los métodos ágiles:

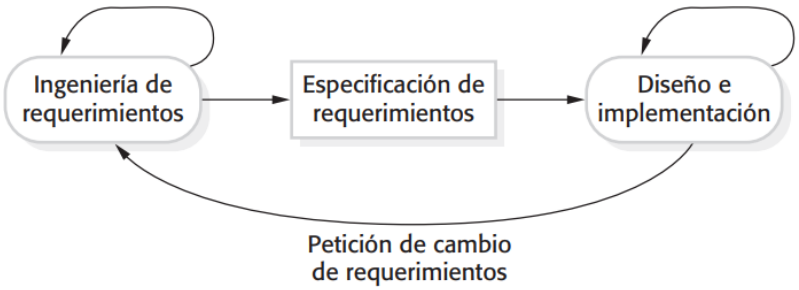
- Participación del cliente: los clientes deben intervenir estrechamente durante el proceso de desarrollo. Su función consiste en ofrecer y priorizar nuevos requerimientos del sistema y evaluar las iteraciones del mismo.
- Entrega incremental: el software se desarrolla en incrementos y el cliente especifica los requerimientos que se van a incluir en cada incremento.
- Personas, no procesos: tienen que reconocerse y aprovecharse las habilidades del equipo de desarrollo. Debe permitirse a los miembros del equipo desarrollar sus propias formas de trabajar sin procesos establecidos.
- Adoptar el cambio: esperar a que cambien los requerimientos del sistema y, de este modo, diseñar el sistema para adaptar dichos cambios.
- Mantener simplicidad: enfocarse en la simplicidad tanto en el software a desarrollar como en el proceso de desarrollo. Siempre que sea posible, trabajar de manera activa para eliminar la complejidad del sistema.

Plan guiado y el desarrollo ágil

Desarrollo guiado por plan:

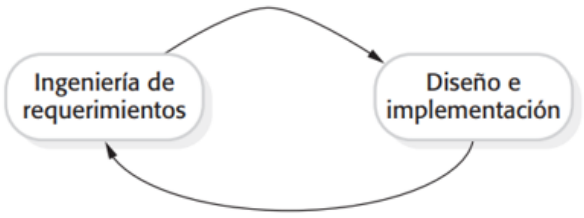
- Etapas bien definidas: el proyecto se divide en etapas claramente definidas, como análisis, diseño, implementación y pruebas. Cada etapa tiene sus salidas asociadas (objetivos y actividades específicas).
- Entregables por etapas bien definidos: cada etapa del proyecto tiene entregables claramente definidos que deben completarse antes de pasar a la siguiente etapa. Esto a menudo implica una revisión y aprobación formal antes de avanzar.

- Entradas de una etapa es la salida de la etapa anterior: las salidas de una etapa se utilizan como entradas o base para la siguiente etapa. Esto asegura una secuencia lineal y predecible en el proceso de desarrollo.
- Las iteraciones se producen dentro de las actividades: si bien puede haber iteraciones en cada etapa, estas se limitan a actividades específicas dentro de esa etapa.
- Inflexible: tiende a ser menos flexible ante los cambios en los requisitos o las prioridades, ya que está basado en un plan predefinido. Los cambios significativos pueden requerir una reevaluación del plan y un proceso de cambio formal.



Desarrollo ágil:

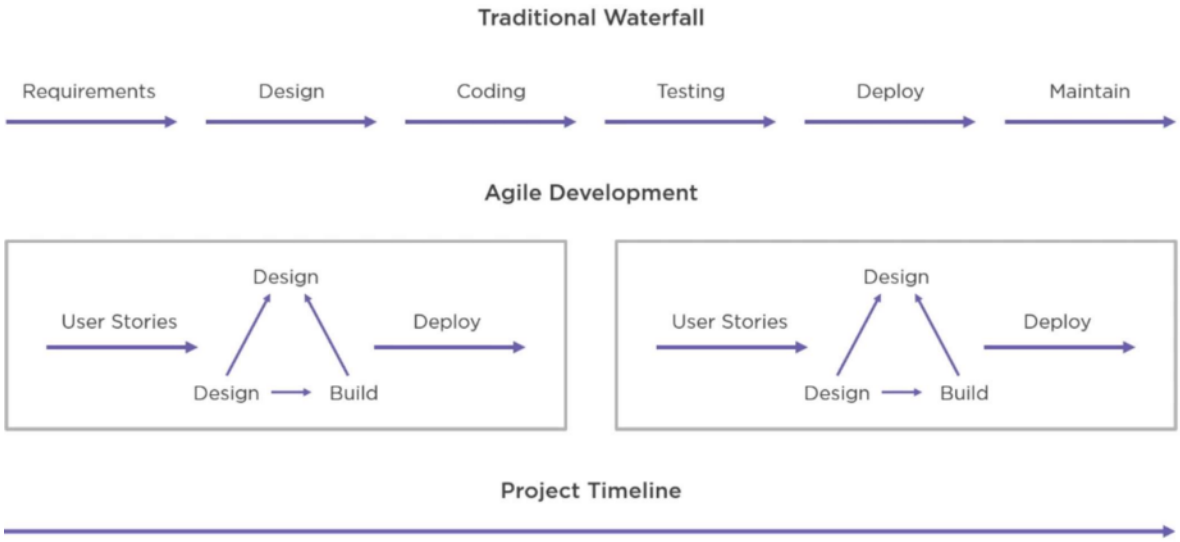
- Especificación, diseño, implementación y pruebas son intercalados: no se espera que todas las especificaciones se completen antes de la implementación. Estas actividades se realizan de manera intercalada y se adaptan según las necesidades.
- Salidas del proceso decididas mediante negociación durante el proceso de desarrollo de software (feedback): los detalles y las prioridades pueden cambiar durante el proceso, y las decisiones se toman mediante la comunicación constante con los interesados.
- Iterativo e incremental: se basa en ciclos cortos de desarrollo llamados iteraciones, donde se entregan partes funcionales del software en cada ciclo.
- Responde al cambio: la flexibilidad es fundamental. Los cambios en los requisitos se pueden abordar de manera más fluida, y el software se adapta a medida que se obtiene más información y feedback.



Waterfall vs agile

_ Lo primero que hay que tener en cuenta es que ambas metodologías suponen dos enfoques distintos para administrar y desarrollar un proyecto. Agile es más colaborativo y enfocado a cambios, siendo una filosofía que implica una forma distinta de trabajar y de organizarse. Waterfall es mucho más secuencial, controlado y estricto, donde tiene mucho más peso la parte inicial del proyecto y la planificación del mismo.

_ Los dos se pueden implementar en una gran variedad de proyectos, aunque la flexibilidad que ofrece Agile resulta esencial para las empresas que se ven obligadas a adaptarse a un entorno que siempre está sujeto a cambios. Esto genera, además, una ventaja competitiva frente a la ejecución de proyectos bajo metodologías tradicionales, o Waterfall.



Diferencia clave de cascada y ágil:

- Waterfall es un modelo de ciclo de vida secuencial, mientras que Agile es una iteración continua de desarrollo y prueba en el proceso de desarrollo de software.
- La metodología Agile es conocida por su flexibilidad, mientras que Waterfall es una metodología estructurada de desarrollo de software.
- Waterfall sigue un enfoque incremental, mientras que Waterfall es un proceso de diseño secuencial.
- Agile realiza pruebas al mismo tiempo que el desarrollo de software, mientras que en la metodología Waterfall, las pruebas se realizan después de la fase de "Construcción".
- Agile permite cambios en los requisitos de desarrollo del proyecto, mientras que Waterfall no tiene la posibilidad de cambiar los requisitos una vez que comienza el desarrollo del proyecto.

Principios manifiesto Ágil

_ El manifiesto ágil está compuesto por 12 principios separados en 3 grupos:

1)_ Entrega del software en intervalos regulares y frecuentes: la mayor prioridad es satisfacer al cliente a través de la entrega temprana y continua de software valioso. Entregar software de trabajo con frecuencia, de una pareja de semanas a un par de meses, con una preferencia a la escala de tiempo más corta. Los patrocinadores, desarrolladores y usuarios deberían poder para mantener un ritmo constante indefinidamente.

2)_ Comunicación en el equipo:

- Los profesionales de negocios y los desarrolladores deben trabajar juntos a diario a lo largo del proyecto.
- El método más eficiente y efectivo para transmitir información a un equipo de desarrollo y dentro de él es la conversación cara a cara.
- Las mejores arquitecturas, requisitos y diseños surgen de equipos autoorganizados.
- Construye proyectos alrededor de individuos motivados. Hay que brindarles el entorno y el apoyo que necesitan y confía en ellos para que hagan el trabajo.
- En intervalos regulares, el equipo reflexiona sobre cómo volverse más efectivo y luego ajusta y adapta su comportamiento en consecuencia.

3)_ Excelencia en el diseño:

- La atención continua a la excelencia técnica y al buen diseño aumenta la agilidad.
- La simplicidad, el arte de maximizar la cantidad de trabajo no hecho, es esencial.
- Los procesos ágiles aprovechan el cambio para la ventaja competitiva del cliente.

Roles en equipos Agiles:

- Un equipo debe tener un experto en el producto o en el dominio.
- Un equipo tiene miembros con habilidades funcionales cruzadas.
- Un equipo debería tener algún papel de liderazgo.
- Un equipo puede beneficiarse de un entrenador o mentor ágil.

Ideas equivocadas sobre Agile:

- Ágil significa “sin compromiso”.
- El desarrollo ágil no es previsible.
- El ágil es una bala de plata.
- Sólo hay una forma de hacer que Agile.
- El Agile no necesita un diseño inicial.
- Ser ágil no es doloroso.

- Estamos haciendo scrum, entonces no necesitamos programar en parejas, refactorizar o el desarrollo guiado por pruebas (TDD).

Errores que se comenten en los teams Agile nuevos:

- Ignorar la reacción de los clientes.
- Pruebas deficientes.
- La falta de potenciación del equipo.
- Falta de reuniones retrospectivas y de demostración.

Ventajas de Agile:

- Un pronto retorno de la inversión.
- Construir los productos adecuados.
- La reacción de los clientes reales.
- Continuamente entregar mejor calidad.

Desventajas de Agile:

- Es difícil evaluar el esfuerzo requerido en el comienzo del desarrollo del software ciclo de vida.
- Puede ser muy exigente con el tiempo de los usuarios.
- Potencial de arrastre del alcance.
- Es más difícil para los nuevos principiantes integrarse en el equipo al ser intenso.

Problemas con métodos ágiles:

- Puede ser difícil mantener el interés de los clientes que están involucrados en el proceso.
- Los miembros del equipo pueden ser inadecuados.
- Priorizar cambios puede ser difícil donde hay múltiples partes interesadas.
- Mantener simplicidad requiere un trabajo extra.
- Los contratos pueden ser un problema, como con otros enfoques para desarrollo iterativo.

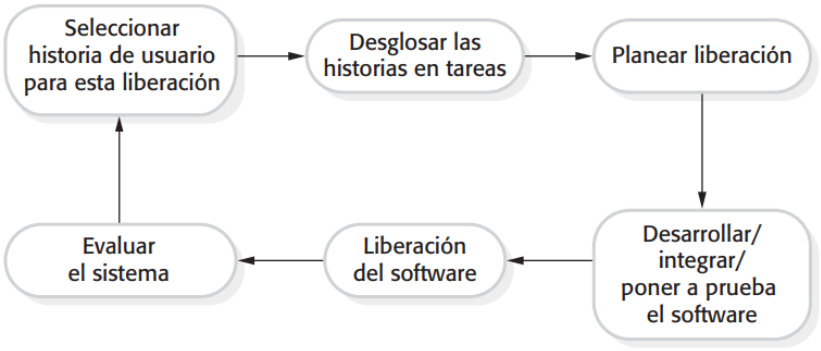
Extreme Programming

Concepto

_ La programación extrema (Extreme programming - XP) es quizás el método ágil mejor conocido y más ampliamente usado. Es una metodología de desarrollo de software la cual intenta mejorar la calidad del software y la velocidad de respuesta ante los requerimientos cambiantes del cliente.

_ En la programación extrema, los requerimientos se expresan como escenarios (llamados historias de usuario), que se implementan directamente como una serie de tareas. Los programadores trabajan en pares y antes de escribir el código desarrollan pruebas para cada tarea. Todas las pruebas deben ejecutarse con éxito una vez que el

nuevo código se integre en el sistema. Entre las liberaciones del sistema existe un breve lapso. La siguiente imagen ilustra el proceso XP para producir un incremento del sistema por desarrollar:



Actividades

Escribir código: en XP, escribir código es una actividad fundamental. Los programadores se enfocan en crear código de alta calidad y funcionamiento.

Testear el sistema: XP promueve la escritura de pruebas (tests) para verificar el correcto funcionamiento del software. Esto incluye pruebas unitarias, de integración y de aceptación.

Escuchar a los clientes y los usuarios: la retroalimentación constante de los clientes y usuarios es esencial en XP. Se busca comprender sus necesidades y requerimientos para adaptar el software en consecuencia.

Diseñar para reducir el acoplamiento: la arquitectura del software se diseña de manera que los componentes estén lo menos acoplados posible. Esto facilita la flexibilidad y el cambio en el software.

Valores

La comunicación es esencial: se promueve una comunicación abierta y constante entre los miembros del equipo de desarrollo, así como con los clientes y usuarios.

Simplicidad: XP favorece soluciones simples y elegantes en lugar de complejas. Se busca evitar la sobrecarga innecesaria.

Aprender del feedback: XP valora la retroalimentación como una oportunidad para aprender y mejorar continuamente el proceso de desarrollo.

Tener coraje: se anima a los miembros del equipo a tomar decisiones valientes y enfrentar los problemas en lugar de ignorarlos.

Respetar al equipo y al proyecto: se fomenta el respeto entre los miembros del equipo y hacia el proyecto en sí. Todos contribuyen al éxito del proyecto.

Principios

Feedback rápido: se busca obtener retroalimentación rápida para identificar problemas y oportunidades de mejora en una etapa temprana del desarrollo.

Construir con simplicidad (DTSTTCPW, KISS, YAGNI): XP aboga por mantener el código simple y evitarla inclusión de características innecesarias. Las siglas se refieren a "Don't Repeat Yourself" (No te repitas), "Keep It Simple, Stupid" (Mantenlo simple, tonto) y "You Aren't Gonna Need It" (No lo necesitarás).

Cambio incremental: XP promueve la implementación de cambios pequeños y frecuentes en lugar de cambios masivos y menos frecuentes. Esto facilita la adaptación a las necesidades cambiantes del proyecto.

Practicas

_ Tenemos mas de 14 practicas aproximadamente, pero mencionamos algunas:

1)_ Feedback detallado: implica obtener comentarios específicos y concretos sobre el software, lo que ayuda a identificar áreas de mejora. Esto puede incluir revisiones de código, pruebas de usuario y evaluaciones de calidad para garantizar que el producto cumple con los requisitos y expectativas de los clientes.

2)_ Proceso continuo: XP promueve un proceso de desarrollo constante y sin interrupciones prolongadas. Los equipos trabajan en ciclos cortos y regulares para entregar incrementos de software funcional. Esto permite una respuesta ágil a los cambios en los requisitos y una entrega más rápida de valor al cliente.

3)_ Entendimiento común: la comunicación efectiva y la comprensión compartida de los objetivos y requisitos del proyecto son esenciales. Todos los miembros del equipo deben tener una visión clara y común del proyecto para colaborar de manera efectiva y alinear sus esfuerzos hacia los mismos objetivos.

4)_Bienestar del programador: se enfoca en cuidar la salud y el bienestar de los programadores. Esto implica evitar el agotamiento, promover un ambiente de trabajo equilibrado y apoyar a los miembros del equipo en su desarrollo profesional y personal, lo que a su vez mejora la productividad y la calidad del trabajo.

5)_ Planeación incremental: los requerimientos se registran en tarjetas de historia (story cards) y las historias que se van a incluir en una liberación se determinan por el tiempo disponible y la prioridad relativa. Los desarrolladores desglosan dichas historias en “tareas” de desarrollo.

6)_ Liberaciones pequeñas: al principio se desarrolla el conjunto mínimo de funcionalidad útil, que ofrece valor para el negocio. Las liberaciones del sistema son frecuentes y agregan incrementalmente funcionalidad a la primera liberación.

7)_ Diseño simple: se realiza un diseño suficiente para cubrir sólo aquellos requerimientos actuales.

8)_ Desarrollo de la primera prueba: se usa un marco de referencia de prueba de unidad automatizada al escribir las pruebas para una nueva pieza de funcionalidad, antes de que esta última se implemente.

9)_ Refactorización: se espera que todos los desarrolladores refactoricen de manera continua el código y, tan pronto como sea posible, se encuentren mejoras de éste. Lo anterior conserva el código simple y mantenible.

10)_ Programación en pares: los desarrolladores trabajan en pares, y cada uno comprueba el trabajo del otro; además, ofrecen apoyo para que se realice siempre un buen trabajo.

Historias de usuario

_ Las historias de usuarios son descripciones cortas y simples de una característica contada desde la perspectiva de la persona que desea la nueva capacidad, generalmente un usuario o cliente del sistema. Estas siguen una simple plantilla:

> Como un <tipo de usuario>,
>
> Quiero <algún objeto>,
>
> Para que <alguna razón>.

_ En XP, el cliente o el usuario es parte del equipo de XP y es responsable de tomar decisiones sobre los requisitos:

1. Las solicitudes de los usuarios se expresan como escenarios o historias del usuario.
2. Estas se ordenan en un backlog por prioridades.
3. El equipo de desarrollo les asigna un costo estimado.
4. El cliente elige las historias para su inclusión en el próximo release en base a sus prioridades y los costos estimados.

_ Las historias de usuarios son parte de un enfoque ágil. Ayudan a cambiar el enfoque de escribir sobre los requisitos a hablar de ellos. Todas las historias de usuarios ágiles incluyen una o dos frases escritas y, lo que es más importante, una serie de conversaciones sobre la funcionalidad deseada. Las historias de usuarios se escriben a lo largo de todo el proyecto ágil, y cualquiera puede escribirlas. Algunos ejemplos son:

- Como usuario, puedo hacer una copia de seguridad de todo mi disco duro.
- Como usuario avanzado, puedo especificar los archivos o carpetas a respaldar en base al tamaño del archivo, la fecha de creación y la fecha modificada.
- Como usuario, puedo indicar a las carpetas que no hagan copias de seguridad para que mi unidad de copia de seguridad no se llene con cosas que no necesito que se salven.

Diseño en XP

_ El diseño en la Programación Extrema (XP) se enfoca en varios principios clave. Estos principios se oponen al enfoque de "Implementar para hoy, diseñar para mañana" porque XP enfatiza la incertidumbre del futuro. En lugar de tratar de predecir y diseñar para todas las eventualidades futuras, XP se enfoca en diseñar de manera ágil y simple para abordarlas necesidades actuales de manera efectiva.

Simplicidad: el diseño del sistema debe ser lo más simple posible. Esto significa evitar la complejidad innecesaria y mantener las soluciones elegantes y directas. La simplicidad facilita la comprensión del código y su mantenimiento.

Diseño Correcto: el diseño debe ser correcto, lo que significa que debe cumplir con los requisitos del usuario y funcionar como se espera. En XP, la corrección es fundamental.

Superar pruebas: el diseño debe pasar todas las pruebas. Esto implica que el software debe funcionar correctamente y ser robusto.

Eliminación de duplicación: el diseño no debe tener lógica duplicada. La duplicación de código es propensa a errores y dificulta el mantenimiento. En XP, se busca eliminar esta duplicación.

Comunicar intenciones: el diseño debe comunicar claramente las intenciones del desarrollador. Esto significa que el código debe ser fácil de leer y comprender para cualquier miembro del equipo.

Mínimas unidades: el diseño debe tenerla menor cantidad de unidades, como clases, métodos y funciones. Esto fomenta la simplicidad y evita la sobrecarga innecesaria.

_ Las ventajas de tener un diseño simple son:

- No se pierde tiempo en funcionalidad superflua.
- Es más fácil de entender.
- Es más fácil de “refactorizar” y de crear “propiedad colectiva” del código.
- Ayuda a mantener a los programadores en schedule o cronograma.

Refactoring en XP

_ Un precepto fundamental de la ingeniería de software tradicional es que se tiene que diseñar para cambiar. Esto es, deben anticiparse cambios futuros al software y diseñarlo de manera que dichos cambios se implementen con facilidad. Vale la pena el gasto de tiempo y esfuerzo anticipando los cambios ya que esto reduce los costos más tarde en la vida del ciclo XP, sin embargo, descarta este principio ya que sostiene que los cambios no se pueden prever de forma fiable. En su lugar, propone la mejora constante de código (Refactoring/Reconstrucción), la cual está motivado por los smells.

_ Los términos "Design Smells" (malos olores de diseño) y "Code Smells"(malos olores de código) se refieren a características o patrones en el diseño de software y el código, que indican posibles problemas más profundos que pueden afectarla calidad del software. Estos "smells" no son errores directos, es decir, el código puede funcionar correctamente, pero indican debilidades en el diseño que pueden conducir a problemas en el desarrollo. Sin embargo, son señales de advertencia que sugieren que el código o el diseño del software pueden necesitar ser refactorizados o mejorados para evitar problemas futuros y mantener un código limpio y mantenible.

Design Smells: se centran en problemas más amplios relacionados con la arquitectura y estructura del software, y su impacto en la capacidad del software para adaptarse a cambios futuros.

- Rigidez: ocurre cuando el diseño del software es difícil de cambiar o extender. Esto puede llevar a problemas si es necesario adaptar el software a nuevas necesidades o requisitos.
- Fragilidad: se refiere a la facilidad con la que pequeños cambios en el software pueden causarla rotura de otras partes del sistema. Esto hace que el mantenimiento y la evolución del software sean complicados.
- Inmovilidad: sucede cuando el código o las partes del sistema son difíciles de reutilizar en otros contextos o proyectos, lo que limita la flexibilidad y la eficiencia del desarrollo.
- Viscosidad: se refiere a la tendencia de un sistema a resistir cambios que mejoran la calidad. Puede deberse a obstáculos en el proceso de desarrollo o decisiones arquitectónicas deficientes.

Code Smells: son características específicas en el código que sugieren problemas potenciales o más profundos. Se enfocan en problemas a nivel de código que afectan la legibilidad y mantenibilidad inmediata del software. Algunos ejemplos son:

- Métodos muy largos: funciones o métodos con un exceso de líneas de código, lo que dificulta su comprensión y mantenimiento.
- Métodos muy similares: varios métodos que realizan tareas similares pero no están refactored en un método común.
- Clases grandes: clases que han crecido demasiado y se convierten en lo que se conoce como "God objects".
- Dependencia inapropiada: clases que tienen dependencias indebidas de los detalles de implementación de otras clases.
- Duplicación de código: porciones idénticas o muy similares de código que existen en múltiples ubicaciones del sistema.
- Ciclomatic complexity: funciones con muchas ramificaciones y bucles que pueden requerir división en funciones más pequeñas.

Technical debt: los Code smells son indicadores de factores que contribuyen al technical debt. Technical debt es un concepto en el desarrollo de software que refleja el costo implícito de rework causado por elegir una solución fácil en lugar de usar una mejor solución que hubiera tomado más tiempo en desarrollarse. Se puede comparar con las deudas financieras, si el technical debt no se paga acumula intereses, haciendo cambios futuros más difíciles.

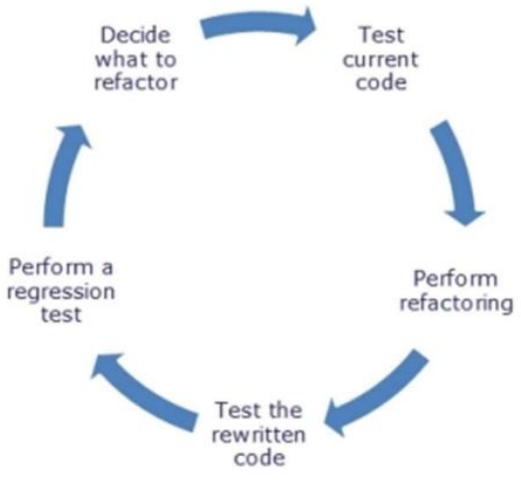
Refactorización: los "Code Smells" son señales de advertencia que indican que una parte del código puede necesitar ser refactorizada. La refactorización es un proceso de mejora del código existente para que sea más limpio, más legible y más mantenible, sin cambiar su funcionalidad externa.

_ Los beneficios de la refactorización incluyen:

- La refactorización hace que el código sea más fácil de entender y de leer. Esto facilita la corrección de errores, ya que se vuelve más evidente dónde y cómo se deben realizarlos cambios.
- Un código refacturado generalmente utiliza patrones de diseño reconocibles y es más flexible. Esto hace que sea más sencillo agregar nuevas funcionalidades o modificar el código existente sin introducir errores.
- La refactorización es una forma de abordar el "technical debt" (deuda técnica). El "technical debt" se refiere a la acumulación de código de baja calidad o no óptimo a lo largo del tiempo. La refactorización ayuda a pagar esa deuda técnica, mejorando la calidad del código y reduciendo los riesgos asociados con un código deficiente.

_ Un aspecto importante de la refactorización es que debe estar respaldada por un conjunto sólido de pruebas automatizadas con una suficiente cobertura. Estas pruebas garantizan que, a medida que se realiza la refactorización, no se introduzcan nuevos errores o problemas en el código existente. Por lo tanto, un "arnés de prueba automatizado" (conjunto de pruebas automatizadas) con una buena cobertura es esencial antes de emprenderla refactorización.

_ La refactorización se lleva a cabo mediante ciclos iterativos de pequeñas transformaciones del programa y pruebas intercaladas. Esto significa que se realizan pequeñas mejoras en el código, seguidas de pruebas para garantizar que no se haya introducido ningún problema nuevo. Este enfoque iterativo garantiza que el código se mantenga en un estado funcional y de alta calidad a lo largo del tiempo, lo que facilita su mantenimiento y evolución.



Técnicas de Refactorización: son acciones específicas que se pueden aplicar para mejorar la calidad del código y su estructura. Estas técnicas se utilizan para mejorar la calidad y la estructura del código, lo que a su vez facilita su mantenimiento y evolución a medida que se desarrolla el software.

_ Técnicas que permiten una mayor abstracción:

- Encapsular campo: esta técnica implica forzar al código a acceder a un campo (variable) a través de métodos getter y setter en lugar de acceder directamente al campo. Esto permite un mayor control y encapsulación de los datos.
- Generalizar tipo: consiste en crear tipos más generales o abstractos que permiten compartir más código. Esto fomenta la reutilización y la flexibilidad en el diseño.
- Sustituir código de comprobación de tipo con estado/estrategia: esta técnica implica reemplazar bloques de código que realizan comprobaciones de tipo con un enfoque basado en estado o estrategia, lo que puede hacer que el código sea más limpio y extensible.
- Sustituir condicionales con polimorfismo: se refiere a reemplazar bloques de código condicionales (if-else) con un diseño basado en polimorfismo, donde se utilizan clases y métodos especializados para realizar diferentes acciones en lugar de múltiples condiciones.

_ Técnicas para dividir el código en piezas más lógicas:

- Componentización: esta técnica descompone el código en unidades semánticas reutilizables con interfaces claras y bien definidas. Esto facilita la reutilización y la comprensión del código.
- Extraer clase: consiste en tomar parte del código de una clase existente y moverlo a una nueva clase. Esto se hace para mantenerla cohesión y mejorarla organización del código.
- Extraer método: se utiliza para dividir una parte de un método grande en un nuevo método separado. Esto hace que el código sea más fácil de entender y mantener, y también se aplica a funciones en lenguajes de programación que no son orientados a objetos.

_ Técnicas para mejorar nombres y ubicación del código:

- Mover método o mover campo: estas técnicas implican mover un método o un campo a una clase o archivo fuente más apropiado. Esto se hace para mejorarla organización del código.
- Renombrar método o renombrar campo: se utilizan para cambiar el nombre de un método o campo a uno que refleje mejor su propósito o funcionalidad.
- Subir método: en programación orientada a objetos, esto implica mover un método de una subclase a una superclase para compartirlo entre varias subclases.

- Bajar método: también en programación orientada a objetos, se trata de mover un método de una superclase a una subclase cuando es más relevante en ese contexto.

Pruebas en XP

_ Los desarrolladores escriben UT (unit test) que deben pasar para poder progresar:

- Los clientes escriben pruebas de aceptación con los desarrolladores que deben pasar para saber cuándo se terminó la tarea.
- Los tests son automatizados y se usan como regresión.
- Crean un sistema que acepta el cambio.

Ventajas de las pruebas en XP:

- Promueven un set de pruebas completo.
- Le dan al desarrollador un objetivo.
- La automatización crea una suite de regresión que como seguro durante el “refactoring".

_ Las pruebas son fundamentales para XP y XP ha desarrollado un enfoque en el que el programa se comprueba después de que cada cambio se ha realizado.

Funciones de prueba de XP:

- Desarrollo de las pruebas en primer lugar.
- Desarrollo de pruebas incrementales a partir de escenarios.
- Participación de los usuarios en el desarrollo de la prueba y validación.
- Arneses de pruebas automatizadas se utilizan para ejecutar todas las pruebas de componentes cada vez que una nueva versión está construida.

Desarrollo de las pruebas primero:

- Se ejecutan automáticamente.
- Cada vez que se agrega una nueva funcionalidad.
- Compruebo que la nueva funcionalidad:
  - Funciona.
  - No agregó errores (regression).

Participación de los clientes: el papel del cliente en el proceso de pruebas es ayudar a desarrollar pruebas de aceptación de las historias que han de ser implementadas en la próxima versión del sistema. El cliente que es parte del equipo de pruebas, escribe pruebas simultáneamente al desarrollo.

Automatización de pruebas: significa que las pruebas se escriben como componentes ejecutables antes de que la tarea se implemente. Estos componentes de prueba deben ser independientes, deberían simular la presentación de la entrada para ser probado y debe comprobar que el resultado cumple con las especificaciones de salida.

Dificultades en pruebas XP: los programadores prefieren programación a las pruebas y a veces se toman atajos al escribir pruebas. Por ejemplo, pueden escribir ensayos incompletos que no comprueban todas las posibles excepciones que puedan ocurrir.

Programación en parejas en XP: los programadores trabajan en parejas, en donde para desarrollar código juntos, y los pares se crean dinámicamente.

Scrum

Gestión de proyectos ágil

_ El enfoque ágil de gestión de proyectos, como se ejemplifica en Scrum, es una respuesta a los desafíos tradicionales de la gestión de proyectos de software. La gestión de proyectos ágil se caracteriza por un enfoque más flexible y adaptable que se alinea con la naturaleza incremental y cambiante del desarrollo de software. En lugar de seguir un plan detallado, se enfoca en valores como la colaboración, la comunicación y la entrega de software funcionando para satisfacerlas necesidades del cliente. Scrum es un ejemplo de un marco de trabajo ágil ampliamente utilizado en la gestión de proyectos.

Enfoque tradicional: este enfoque para la gestión de proyectos es el direccionado por plan, los gerentes de proyectos elaboran un plan detallado que describe qué debe ser entregado, cuándo debe ser entregado y quién trabajará en el desarrollo del proyecto. Este plan a menudo se sigue rigurosamente a lo largo del proyecto.

Manifiesto Ágil: define los valores y principios que guían la gestión de proyectos ágil. Estos valores incluyen la prioridad de individuos e interacciones sobre procesos y herramientas, la entrega de software funcionando sobre documentación extensa, la colaboración con el cliente y la respuesta al cambio. La gestión de proyectos ágil destaca la colaboración entre el equipo y el cliente, la comunicación efectiva y la entrega continua de software.

Scrum

_ Scrum es un ejemplo de marco de trabajo ágil, define una estrategia flexible en la que un equipo de desarrollo trabaja de manera colaborativa y autónoma para alcanzar una meta común. Se basa en iteraciones y entrega incremental de funcionalidad.

_ Se utiliza para abordar problemas que son complejos y cambiantes, lo que es común en el desarrollo de software. Proporciona un enfoque ágil para manejar la incertidumbre y los cambios constantes. El objetivo de Scrum es entregar productos de la máxima calidad y valor de manera productiva y creativa. Esto se logra mediante la colaboración y la adaptación continua.

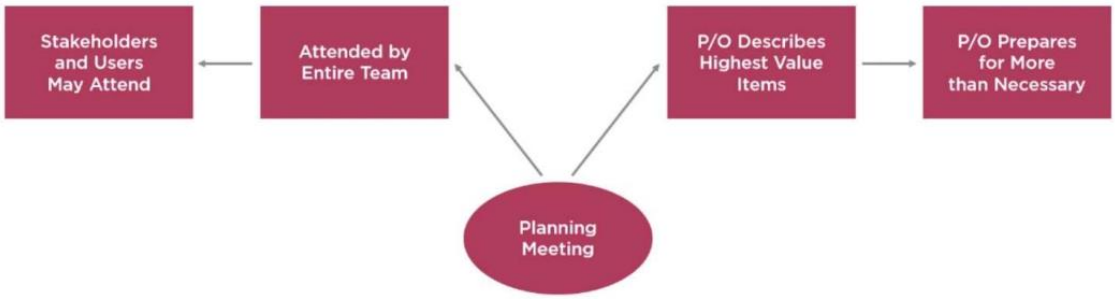
_ Es un marco de trabajo ligero, lo que significa que es fácil de entender y adoptar. No es un proceso o técnica específica, sino un marco en el que se pueden emplear diversas prácticas y procesos. A pesar de su simplicidad aparente, Scrum puede ser

extremadamente difícil de dominar. Requiere una comprensión profunda y la práctica continua para implementarse de manera efectiva.

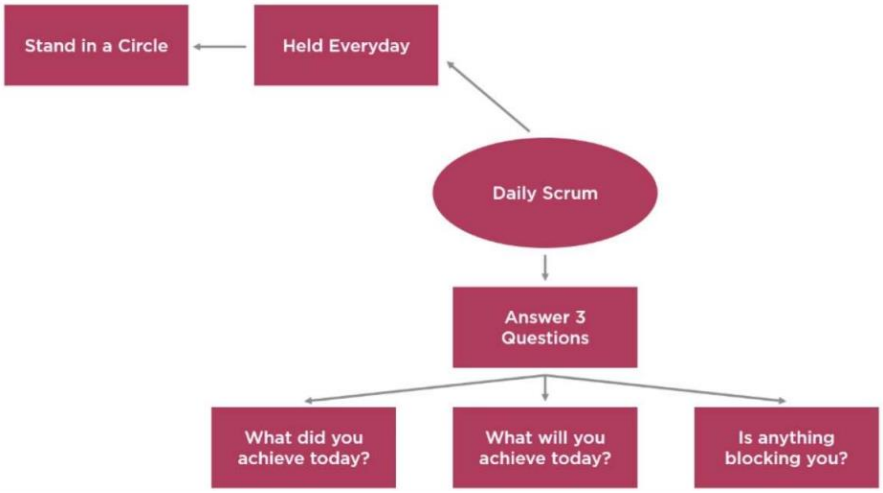
Teoría del control de proceso empírica o empirismo: Scrum se basa en esta teoría, lo cual implica tomar decisiones basadas en hechos reales y experiencia en lugar de suposiciones. La gestión de proyectos Scrum es iterativa e incremental. Los pilares del control empírico son la transparencia, la inspección y la adaptabilidad. Estos principios guían la toma de decisiones en el marco de trabajo.

Eventos formales: además, Scrum utiliza ciclos de feedback para la mejora continua. El tiempo se divide en sprints, que son ciclos cortos. El producto debe estar en un estado "potencialmente entregable" en todo momento, y al final de cada sprint se realizan eventos formales para revisar los resultados y planificar el siguiente sprint. Los eventos formales en Scrum incluyen:

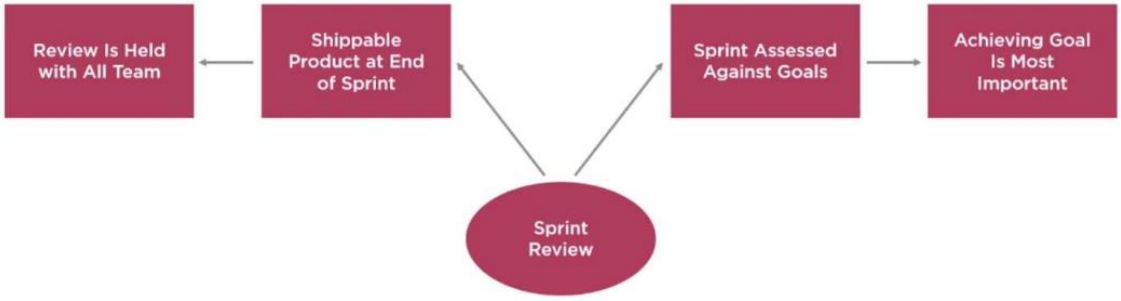
- Reunión de Planificación del Sprint (Sprint Planning Meeting):



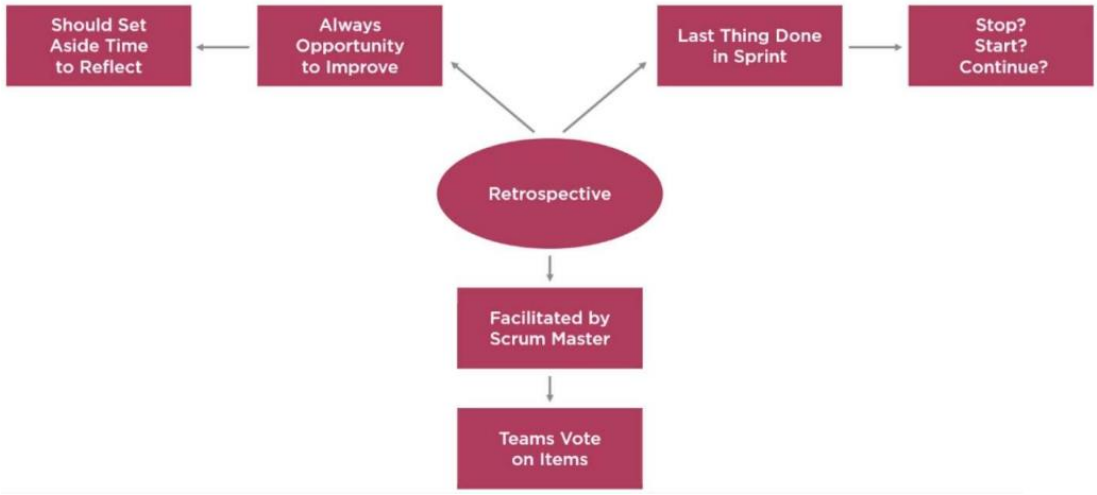
- Scrum Diario (Daily Scrum):



- Revisión del Sprint (Sprint Review):



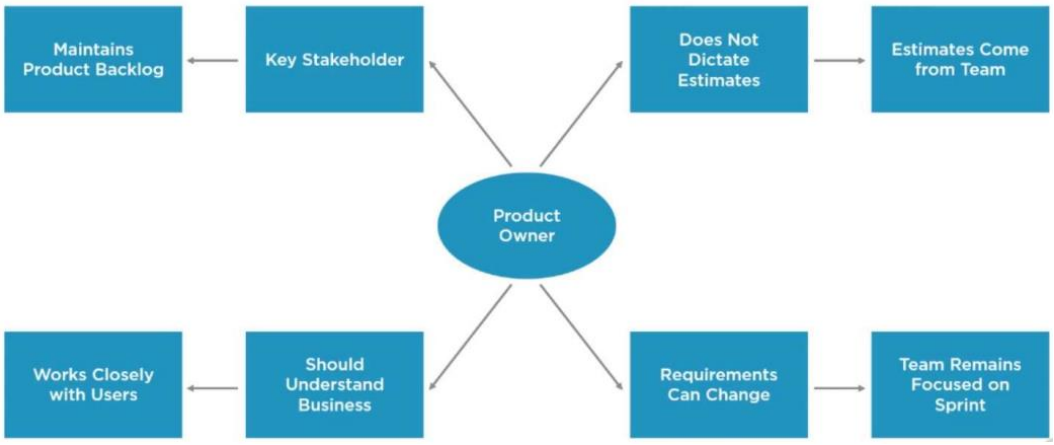
- Retrospectiva del Sprint (Sprint Retrospective):



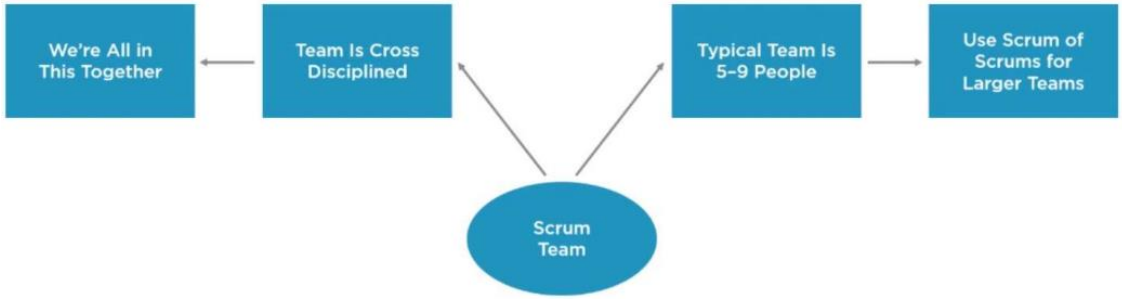
_ Estos eventos ayudan a mantener la transparencia, permiten la inspección y facilitan la adaptabilidad en el proceso de desarrollo.

<u>Roles:</u>

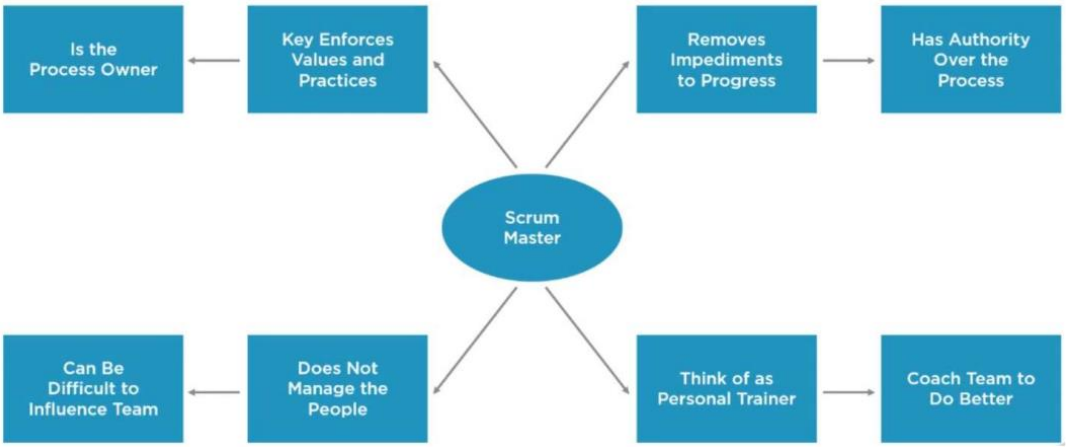
Product Owner: individuo (o posiblemente un pequeño grupo) cuyo trabajo es identificar el producto características o requisitos, priorizarlos para el desarrollo y continuamente revisar el retraso del producto para asegurar que el proyecto siga cumpliendo con los requisitos críticos las necesidades del negocio. El dueño del producto puede ser un cliente pero también puede ser un gerente de producto en una empresa de software u otro representante de los interesados.



- Scrum Team:

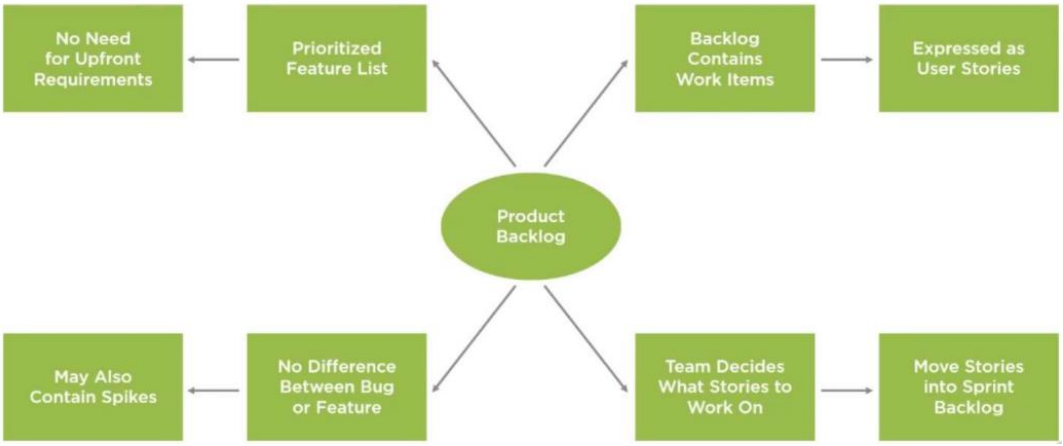


- Scrum Master: responsable de asegurar que el proceso de Scrum sea siguió y guía al equipo en el uso efectivo de Scrum. Responsable de interactuar con el resto de la compañía y de asegurar que el equipo de Scrum no se desvíe por interferencias externas.

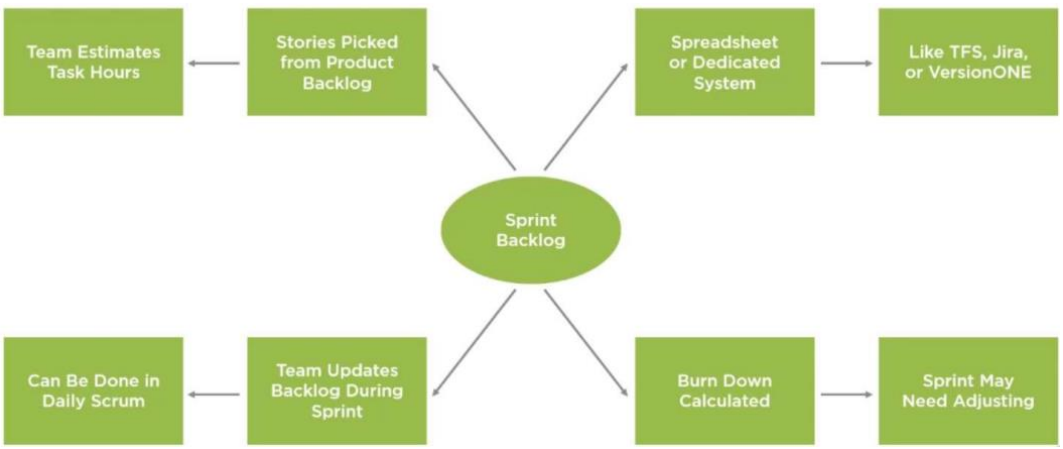


Artefactos:

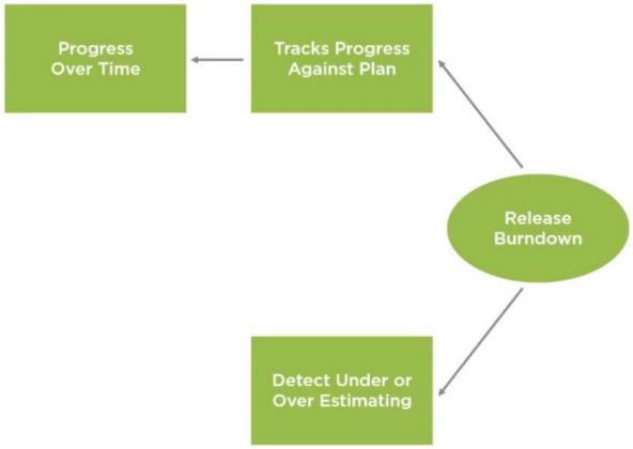
- Product Backlog: lista de cosas "por hacer" que el equipo de Scrum debe abordar. Pueden ser definiciones de características del software, requisitos del software, historias de usuarios o descripciones de tareas suplementarias que se necesitan, como la arquitectura definición o documentación de usuario.



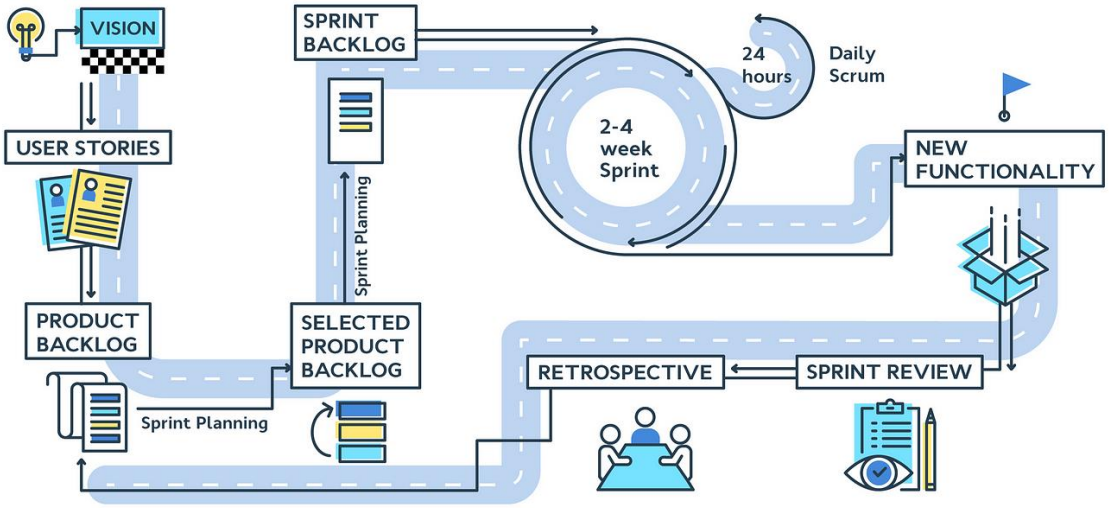
- Sprint Backlog:



- Burndown Chart:



Proceso Scrum:



Ciclo del sprint: los sprints son de longitud fija, normalmente 2-4 semanas. Se corresponden al desarrollo de una versión del sistema en XP. El punto de partida para la planificación es la acumulación de stories, que es la lista de trabajo a realizar en el proyecto.

_ La fase de selección involucra a todo el equipo del proyecto, que trabajan con el cliente para seleccionar las funciones y funcionalidad que se desarrollará durante el sprint.

_ Están basados en prioridades, esfuerzo estimado y la velocidad del team.

_ El planeamiento es el siguiente:

- Estimaciones: durante la planificación del sprint, el equipo realiza estimaciones para determinar cuánto trabajo se puede realizar durante el sprint. Las

estimaciones se pueden realizar utilizando diferentes técnicas, como los puntos de historia o el poker planning.

- Puntos de historia: los puntos de historia son una unidad de medida comúnmente utilizada en Scrum para estimar el esfuerzo requerido para completar una historia. Estas estimaciones ayudan a determinar cuántas historias se pueden incluir en un sprint.
- Poker planning: el poker planning es una técnica en la que los miembros del equipo asignan puntos de historia a las historias de usuario. Esto se hace de manera colaborativa y ayuda a establecer un consenso sobre las estimaciones.
- Velocidad: la velocidad del equipo se refiere a la cantidad de trabajo que el equipo puede completar de manera consistente durante un sprint. Se basa en la experiencia acumulada en sprints anteriores y se utiliza para predecir cuánto trabajo se puede realizar en futuros sprints.

_ Una vez que éstos están de acuerdo, el equipo se organiza para desarrollar el software. Durante esta etapa, el equipo está "aislado" del cliente y la organización, con toda comunicaciones canalizadas a través del denominado "Scrum Master". El papel del Scrum Master es proteger el equipo de desarrollo de las distracciones externas. Al final del sprint, el trabajo realizado es revisado y se presenta a las partes interesadas. Después, el siguiente ciclo comienza nuevamente.

Trabajo en equipo en Scrum y Daily Stand-ups: el Scrum Master es un facilitador que organiza reuniones diarias, rastrea la acumulación de trabajo por hacer, registra las decisiones, mide el progreso contra el atraso y se comunica con los clientes y la gestión fuera del equipo.

_ Todo el equipo asiste a las reuniones diarias cortas donde todos los miembros del equipo comparten información, describen su progreso desde la última reunión, los problemas que han surgido y que se ha previsto para el día siguiente. Esto significa que todos en el equipo saben lo que está pasando y, si surgen problemas, puede volver a planear el trabajo a corto plazo para hacer frente a ellos.

- Preguntas en los daily stand ups: ¿Qué hice ayer?, ¿Qué planeo hacer hoy?, ¿Tengo algún impedimento?

Beneficios del Scrum:

- El producto se divide en un conjunto de fragmentos manejables y comprensibles.
- Requerimientos inestables no retrasan el progreso.
- Todo el equipo tiene visibilidad de todo y por lo tanto se mejora la comunicación del equipo.
- Los clientes ven la entrega a tiempo de los incrementos y obtienen retroalimentación sobre cómo funciona el producto.
- La confianza entre los clientes y los desarrolladores se establece y una cultura positiva se crea en la que todo el mundo espera que el proyecto tenga éxito.

Automatización de pruebas

Introducción

Testing

_ Muchos proyectos solamente dependen del testing manual para verificar que el software cumpla con los requerimientos funcionales y no funcionales. A veces, existen tests automatizados pero están desactualizados y pobremente mantenidos y requieren que se los suplemente con test manual extensivo

Build quality in (construir la calidad desde el inicio): es un principio fundamental en el desarrollo de software ágil. Significa que en lugar de depender en inspecciones y pruebas masivas al final del proceso de desarrollo para garantizar la calidad, se debe enfocar en incorporar la calidad en el producto desde el principio del proyecto y mantenerla a lo largo de todo el ciclo de desarrollo. Esto implica que se integra en todas las etapas del proceso. Para testing y la calidad del software, "building quality in" significa:

- Escribir tests automatizados a diferentes niveles: esto implica la creación de pruebas automatizadas en varios niveles del software, como pruebas de unidad para verificar funciones individuales, pruebas de componentes para evaluar módulos más grandes y pruebas de aceptación para verificar que el software cumple con los requisitos del usuario.
- Ejecutar pruebas como parte del deployment pipeline: el deployment pipeline es un conjunto automatizado de pasos que se inicia cada vez que se realiza un cambio en el código, la configuración o el entorno del software. Como parte de este proceso, se ejecutan pruebas automatizadas para garantizar que los cambios no introduzcan nuevos errores y que el software siga funcionando como se espera.

_ La idea es que el testing automatizado se integre de manera continua en el proceso de desarrollo, lo que permite detectar y corregir problemas de manera temprana y garantizar que la calidad esté presente en cada versión del software que se construye. Esto ayuda a reducir la necesidad de correcciones masivas o inspecciones finales, ya que los problemas se abordan de manera proactiva a lo largo del ciclo de desarrollo.

Proyecto ideal: implica varios aspectos clave para garantizar la calidad y la eficiencia en el proceso de desarrollo. Fomenta una colaboración cercana y continua entre diferentes partes interesadas, incluidos los testers, los desarrolladores y los usuarios. Esto permite una comprensión compartida de los requisitos y la calidad esperada del software.

Tests automatizados: se escriben desde el comienzo del proyecto, incluso antes de que los desarrolladores empiecen a trabajar en las características que los tests van a evaluar ya que describen en detalle cómo se supone que el sistema debe funcionar y

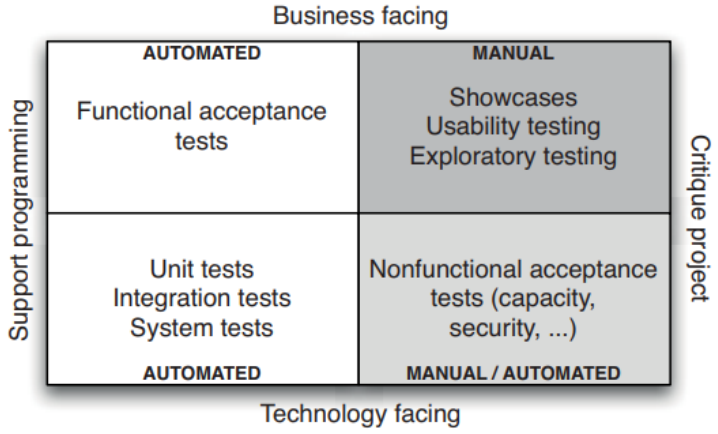
qué resultados se esperan, esto asegura que la calidad sea una prioridad desde el principio.

_ Cuando los tests automatizados pasan, demuestran que la funcionalidad requerida por el cliente ha sido implementada completamente y correctamente. Esto garantiza que el software cumple con los requisitos del usuario. Los tests automatizados se ejecutan en un sistema de CI cada vez que se realiza un cambio en el código, la configuración o el entorno del sistema. Esto asegura que los tests se ejecutan de manera regular y ayuda a detectar problemas temprano.

_ Los tests automatizados no solo se utilizan para verificar nuevas características, sino que también sirven como tests de regresión. Esto significa que se ejecutan para garantizar que las actualizaciones no hayan introducido errores en las áreas existentes del software. Además de las pruebas de funcionalidad, un proyecto ideal también incluye tests automatizados para evaluar aspectos "no funcionales" del sistema, como rendimiento, seguridad y escalabilidad.

Tipos de tests

_ En este diagrama, se clasifica las pruebas según si están orientadas al negocio o a la tecnología, y si apoyan el proceso de desarrollo o se utilizan para criticar el proyecto:



Soporte del proceso de desarrollo, de cara al negocio

Test funcionales de aceptación: desempeñan un papel fundamental en el soporte del proceso de desarrollo en un entorno ágil. Estos se centran en asegurar que los criterios de aceptación de una historia (o feature) se cumplan y que el software cumpla con las expectativas del usuario. Idealmente, los tests funcionales de aceptación se escriben y automatizan antes de que el desarrollo de una historia comience. Esto proporciona una especificación clara de lo que se espera del desarrollo y establece criterios claros para la finalización de la implementación. Estos tests responden a dos preguntas fundamentales en un entorno ágil:

- ¿Cómo sé cuándo terminé la implementación? (respondido por el desarrollador).

- ¿Conseguí lo que quería? (respondido por el usuario).

_ En un proyecto ideal, los usuarios, o aquellos que comprenden mejor las necesidades del negocio, deberían participar en la redacción del criterio de éxito de una historia. Este criterio se traduce en un test de aceptación. Una herramienta usada es "Cucumber", permiten a los usuarios escribir scripts de prueba en un lenguaje más comprensible para ellos.

_ Tanto los testers como los desarrolladores trabajan en la implementación de los tests de aceptación. Los testers aportan su experiencia en pruebas y calidad, mientras que los desarrolladores trabajan en la implementación técnica. La herramienta, como Cucumber, actúa como un mecanismo para mantener sincronizados a todos los involucrados, ya que el mismo script de prueba se convierte en una especificación ejecutable que se ejecuta en el software.

Modelo "Given-When-Then" (GWT): es una forma estructurada de escribir pruebas o especificaciones, como pruebas unitarias, pruebas de aceptación y pruebas funcionales, ayuda a garantizar que se comprendan y validen claramente los comportamientos y resultados del sistema. El modelo es eficaz para describir el "camino feliz" (happy path) de la prueba, que es el escenario ideal en el que todo funciona como se espera:

1. Given (Dado): se establece el contexto inicial necesario para la prueba (configuración de datos, el estado del sistema). Es la preparación previa a la acción principal de la prueba.
2. When (Cuando): se describen las acciones o eventos que se desencadenan, lo que se está probando o la acción principal que se está llevando a cabo. Esto representa la operación que se evalúa en la prueba.
3. Then (Entonces): se establecen las expectativas o resultados esperados después de que se haya realizado la acción descrita en la sección "When", acá se verifica si el sistema se comporta de la manera esperada.

_ Además del "happy path", también es importante considerar escenarios alternativos (alternative paths), estos casos pueden incluir variaciones en el estado inicial, acciones alternativas o resultados inesperados. Y "sad paths" que involucran situaciones de error o condiciones inesperadas.

_ La automatización de los tests de aceptación ofrece numerosos beneficios:

- Costo: elimina la necesidad de realizar pruebas repetitivas manualmente en cada ciclo de desarrollo, lo que reduce los costos laborales y mejora la eficiencia.
- Retroalimentación más rápida: permite identificar y solucionar problemas de manera más eficiente y, acelera el proceso de desarrollo.
- Reducción de la carga de los testers: los testers pueden enfocarse en pruebas más creativas y exploratorias en lugar de realizar pruebas repetitivas y manuales.

- Conjunto de tests de regresión poderoso: cuando se realizan cambios en el código, las pruebas automatizadas pueden ejecutarse para asegurarse de que las nuevas características o correcciones no afecten negativamente las funcionalidades existentes.
- Documentación del sistema: los tests describen claramente cómo se espera que funcione el sistema, lo que facilita la comprensión del comportamiento del software.

Soporte del proceso de desarrollo, de cara a la tecnología

_ Tenemos los siguientes test, escritos y mantenidos exclusivamente por desarrolladores:

Pruebas de unidad (Unit test):

- Prueban una parte en particular del software aislada (funciones o métodos).
- Necesitan simular otras partes del sistema (test doubles).
- No deberían llamar, bases de datos, otros sistemas, etc.
- Deberían ser muy rápidos.
- Deberían por lo menos cubrir el 80%, idealmente el 100%.
- Pierden la posibilidad de detectar bugs que provienen de la interacción entre diferentes partes del software.

Pruebas de componente (Component tests):

- Evalúan componentes más grandes o módulos del sistema para garantizar que funcionen de manera efectiva juntos, pueden abarcar múltiples unidades de código o clases interconectadas.
- Son más lentas.
- A veces se los conoce como tests de integración.

Pruebas de despliegue (Deployment test):

- Se ejecutan cada vez que se instala la aplicación.
- Verifican que el deployment funcionó y que la aplicación fue correctamente instalada, correctamente configurada y es capaz de contactar cualquier servicio que necesite y que este responda.

Critican el proyecto de cara al negocio

_ En esta categoría tenemos pruebas únicamente manuales, y verifican que la aplicación le entrega al usuario el valor que este está esperando, pero no es solo validar que la aplicación cumpla con las especificaciones, es también validar que las especificaciones están correctas. Cuando los usuarios utilizan la aplicación en vida real siempre descubren que se puede mejorar, tal vez se descubren nuevas features inspiradas en la entrega actual. Por eso decimos que el desarrollo de software es iterativo en su esencia.

Demos o reviews:

- Los teams ágiles muestran al final de cada iteración el resultado de lo que se desarrolló.
- Una demo siempre tendrá comentarios, son el usuario y el team los que deciden cuanto quieren cambiar el plan para incorporar el resultado de la review.
- Cuando estos tests pasan es cuando uno realmente puede decir que ha cumplido la labor.
- Cliente contento :-).

Canary testing: se refieren a probar una nueva versión de software o una nueva característica con usuarios reales en un entorno en vivo (de producción). Se realiza enviando algunos cambios de código en vivo a un pequeño grupo de usuarios finales que generalmente no saben que están recibiendo un código nuevo.

Critican el proyecto de cara a la tecnología

_ Los tests de aceptación vienen en dos categorías, tanto funcionales como no funcionales.

Tests no funcionales de aceptación: mientras que los tests funcionales se centran en las características directas del sistema, los tests no funcionales se enfocan en aspectos como la capacidad, disponibilidad y seguridad, entre otros. A menudo, los usuarios no especifican estos requisitos no funcionales desde el principio, pero se dan por sentados en muchos casos. La realización de tests no funcionales suele requerir recursos significativos, herramientas específicas y tiende a ser más lenta. Generalmente, se ejecutan hacia el final del proceso de implementación del sistema para garantizar que se cumplan los estándares requeridos.

Test en el commit stage

_ En el desarrollo de software, la mayoría de las pruebas deben ser pruebas de unidad. Estas pruebas son rápidas de ejecutar y deben cubrir al menos un 80% del código. Sin embargo, es importante recordar que las pruebas de unidad, por sí solas, no garantizan el funcionamiento completo de la aplicación. Para ello, se necesita el respaldo del resto del pipeline de implementación. Para lograr que los tests de unidad sean rápidos, hay varias estrategias efectivas que se pueden seguir:

- Evitar la interfaz de usuario:
  - Los tiempos de los humanos son extremadamente lentos para las máquinas.
  - Gran cantidad de componentes requiere mucho esfuerzo.
- Inyección de dependencias o inversión de control:
  - Dependencias provistas desde fuera.
  - Un buen diseño, permiten introducción de test doubles.

- Evitar la base de datos:
  - O cualquier otro subsistema que no sea el testeado.
  - Debería ser sencillo de evitarla.
- Evitar el asincronismo:
  - En el commit stage.
  - Dividir los tests.
- Usar test-doubles.
- Minimizar el estado en los tests.
  - Testear comportamiento y minimizar el estado.
  - Evitar tests complejos que lleven esfuerzo considerable para preparar los datos de entrada.
- Simular el tiempo:
  - Abstraer la información del tiempo en una clase separada.
  - Inyectar una implementación que simula el tiempo.
- Fuerza bruta:
  - Paralelizar la ejecución.
  - Separar los tests suites para poder paralelizar.

Test doubles

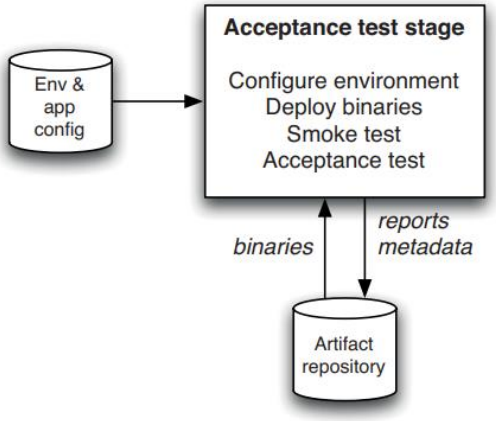
_ En el contexto de las pruebas automatizadas, una estrategia clave es reemplazar partes del sistema con versiones simuladas para tener un mayor control sobre su comportamiento. Estas partes simuladas se conocen como "test doubles" o sustitutos de prueba, y existen varios tipos:

- Dummy objects: se pasan pero nunca se usan.
- Fake objects: tienen implementaciones funcionales, pero tomando atajos que los hacen no usables en producción.
- Stubs: proveen respuestas predeterminadas a las llamadas que reciben durante el tests.
- Spies: registran información en base a como son llamados durante el tests.
- Mocks: son objetos pre-programados con “expectations” que forman una especificación de las llamadas que esperan recibir.

_ El uso de estos "test doubles" permite aislar componentes y simplificar las pruebas al controlar el comportamiento de las partes del sistema que no son relevantes para la prueba actual. Esto facilita la identificación de problemas y garantiza que las pruebas sean efectivas y confiables.

Tests en el acceptance test stage

_ Los tests de aceptación automatizados son una parte fundamental en el pipeline de implementación. Van más allá de la integración continua y se centran en evaluar el cumplimiento de los criterios de aceptación del negocio. Estas pruebas se ejecutan contra cada versión de la aplicación que ha pasado la etapa de compromiso (commit stage), asegurando así que el software cumple con los estándares requeridos antes de su despliegue.



_ Los tests de aceptación automatizados son esenciales por varias razones. En primer lugar, reducen significativamente el costoso proceso de pruebas manuales, ya que no es necesario repetirlo en cada nueva versión del software. Además, al hacer pruebas manuales generalmente al final, los defectos se capturan tarde en el ciclo de desarrollo, lo que aumenta el riesgo.

_ Los tests de unidad y de componente, aunque importantes, no se centran en probar escenarios completos, por lo que no son suficientes para garantizar el cumplimiento de los criterios de aceptación del negocio. Además, los tests de aceptación actúan como una protección contra cambios importantes en el diseño o la arquitectura del software, ya que no es necesario rehacerlos cuando se realizan modificaciones.

_ En última instancia, los tests de aceptación automatizados fomentan la planificación adecuada y ejercen una presión saludable para mantener altos estándares de calidad en el desarrollo de software.

Implementación de los tests de aceptación

_ La implementación de tests de aceptación es esencial en el desarrollo de software, pero presenta desafíos únicos. A diferencia de los tests de unidad, los tests de aceptación no pueden ser completamente stateless, ya que necesitan simular las interacciones del usuario y probar que el sistema cumple con los requerimientos del negocio. Para manejar el estado, se recomienda evitar cargar bases de datos completas de producción y en su lugar mantener un conjunto mínimo y controlado de datos. Además, es fundamental usar la API pública de la aplicación para configurar el estado en lugar de ejecutar scripts directamente contra la base de datos.

_ La encapsulación y el diseño adecuado son esenciales en los tests de aceptación, y se debe resistirla tentación de romper el encapsulamiento para fines de prueba. Es importante evitar la creación de "backdoors" o código específico para pruebas. Además, la asincronía y los tiempos de espera son desafíos en los tests de aceptación, ya que se deben probar interacciones completas. Se debe minimizar el tiempo de espera siempre que sea posible.

_ Los tests de aceptación deben ejecutarse en un entorno similar al de producción y no deben incluir integración con sistemas externos. En su lugar, se pueden utilizar "tests doubles" para controlar el estado y el comportamiento de sistemas externos. Un buen diseño minimiza el acoplamiento entre la aplicación y los sistemas externos, lo que permite implementar patrones como el "circuit breaker".

_ En cuanto a los puntos de integración externos, es fundamental probar la integración en lugar de los sistemas externos en su totalidad. Los tests deben centrarse en las interacciones necesarias y adaptarse a las circunstancias, ya sea que los sistemas externos estén en producción madura o en desarrollo activo. La estrategia es empírica, creando tests que aborden los problemas a medida que se descubren.

_ En resumen, los tests de aceptación requieren planificación cuidadosa, diseño eficiente y estrategias específicas para manejar el estado, la encapsulación, la asincronía y la integración con sistemas externos.

Deployment tests

_ Las pruebas de implementación, o deployment tests, son fundamentales para asegurarse de que el entorno donde se ejecutan los tests de aceptación sea lo más similar a producción posible, idealmente idéntico. Esto proporciona una excelente oportunidad para probar los instaladores en un entorno de producción simulado y garantizar que los componentes se puedan comunicar correctamente. Estas pruebas deben incluirse como una suite adicional al principio del pipeline de implementación para que cualquier falla en ellas cause un fallo inmediato, evitando esperas innecesarias de tiempo de espera.

Métricas

Medición y métricas del software

Medición del software: se ocupa de derivar un valor numérico para un atributo de software como su complejidad o confiabilidad, comparando los valores medidos entre ellos y con los estándares de la organización se pueden extraer conclusiones acerca de: la calidad del software y evaluar la efectividad del proceso, herramientas o métodos.

_ El objetivo final de la medición del software es usarla medición para hacer juicios sobre la calidad del software Idealmente, un sistema podría ser evaluado usando un rango de métricas para medir sus atributos de las medidas obtenidas inferir un valor de su calidad y compararlo contra un threshold para saber si la calidad requerida ha sido alcanzada Sin embargo la industria está aún lejos de la situación ideal.

Métrica del software: es una característica de un sistema de software, de su documentación, o del proceso de desarrollo que puede ser medido de manera efectiva Las métricas pueden ser de control o de predicción. Las métricas de control dan soporte a la gestión del proceso y las métricas de predicción ayudan a predecir características del software.

_ A continuación, vemos ejemplos de métricas de proceso:

- El tiempo que lleva completar un proceso particular: calendar time, etc.
- Los recursos requeridos para un proceso particular: cuántas personas/día, etc.
- El número de ocurrencias de un evento particular:
  - Número de defectos promedio encontrados en una inspección.
  - Número de pedidos de cambio de requerimientos.
  - Número de líneas de código modificadas por pedidos en los requerimientos, etc.

_ A continuación, tenemos ejemplos de métricas de predicción o de producto:

- Están asociadas con el software en sí mismo: complejidad ciclomática, longitud promedio de los identificadores, número de atributos y operaciones que tiene un clase, etc.
- Ambos tipos de métricas pueden influenciar en las decisiones del management Los managers usan mediciones de proceso para decidir si se debería cambiar el proceso métricas de predicción para decidir si el software tiene que ser cambiado o si está listo para ser liberado.

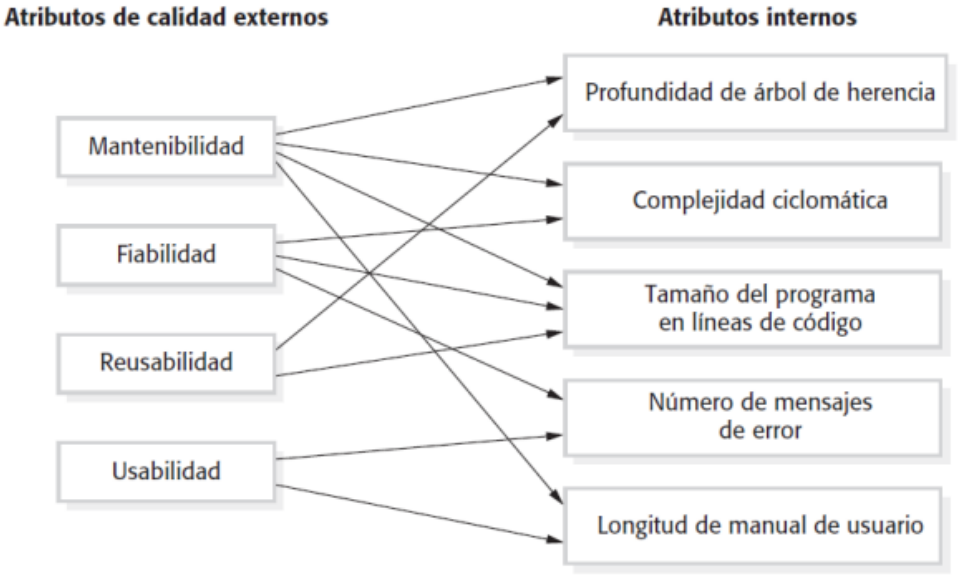
Métricas del software

_ Las métricas del software deberían usarse de dos maneras:

1. Midiendo características del sistema y agregándoles para evaluar atributos de calidad del sistema.
2. Para identificarlos componentes del sistema que están por debajo de los estándares.

_ Es difícil hacer una medición directa de los atributos de calidad del software (mantenibilidad, usabilidad, etc). Estos son atributos externos relacionados a cómo los desarrolladores y los usuarios experimentan el sistema. No pueden medirse objetivamente, pero se pueden medir atributos internos del software como el tamaño o la complejidad y asumir una relación entre ellos.

Relación intuitiva entre atributos internos y atributos externos de calidad:



Condiciones para que una medida de un atributo interno pueda utilizarse como predictor de un atributo de calidad externo: el atributo interno debe poder medirse de manera precisa. Muchas veces se utilizan tools específicas, debe existir una relación entre el atributo interno que se mide y el atributo de calidad externo que se intenta predecir. La relación entre el atributo interno y el atributo de calidad externo debe ser entendida, validada y expresada en términos de una fórmula o modelo Identificar la forma funcional del modelo (lineal, exponencial, etc). Identificar los parámetros que se incluirán en el modelo, calibrar estos parámetros usando datos existentes, o se puede utilizar data-mining para descubrir relaciones y hacer predicciones sobre los atributos externos.

¿Por qué no es tan común el que se adopte un programa de métricas en una organización?: es difícil cuantificar el ROI (retorno de inversión), los beneficios suelen ser a largo plazo, esto hace que las organizaciones sean reacias a invertir en métricas, ya que puede ser difícil demostrar un beneficio financiero inmediato. Además, algunas

organizaciones pueden argumentar que han logrado mejorar la calidad del software en los últimos años sin necesidad de implementar un programa de métricas específico. Esto puede llevar a la percepción de que las métricas no son necesarias para lograr resultados positivos.

_ Introducir métricas puede requerir la implementación de herramientas especializadas, lo que puede aumentar la complejidad y los costos del proceso. Esto puede desalentar a algunas organizaciones, especialmente las más pequeñas, y con la adopción creciente de metodologías ágiles en el desarrollo de software, algunas métricas tradicionales pueden no encajar bien con estos enfoques. Las prácticas ágiles se centran en la entrega de software funcional de manera continua y la colaboración en equipo, lo que puede hacer que las métricas tradicionales sean menos relevantes.

Métricas del producto

_ Desempeñan un papel importante en la evaluación y mejora de la calidad del software, pero su aplicación y relevancia pueden variar según el contexto y los objetivos específicos del desarrollo de software. Estas métricas se dividen en dos categorías principales:

Métricas dinámicas: se recopilan midiendo un programa en ejecución. Suelen estar relacionadas con la eficiencia y la confiabilidad del software, ya que miden el comportamiento del programa en tiempo real. Por ejemplo, el número de errores o bugs reportados, el tiempo de respuesta del software y otros indicadores de rendimiento en tiempo real que pueden relacionarse directamente con la eficiencia y la confiabilidad del software.

Métricas estáticas: se recopilan midiendo representaciones del sistema sin necesidad de ejecutar el programa. El análisis estático del código es una técnica común para recopilar métricas estáticas. Se utilizan para evaluar aspectos como la complejidad, la mantenibilidad y entendimiento del software. Métricas relacionadas con el diseño del software, la calidad del código y la documentación, el tamaño del programa o la complejidad del control pueden ayudar a predecir estos atributos de calidad, aunque la relación es más compleja y no siempre concluyente.

_ Estas son algunas métricas de rendimiento comunes utilizadas en ingeniería de software:

- Tiempo de respuesta (Response Time): se refiere al tiempo que un sistema necesita para procesar una solicitud desde una perspectiva externa. En otras palabras, es el tiempo que un usuario o un sistema debe esperar para obtener una respuesta o resultado de una solicitud.
- Tiempo de reacción (Responsiveness): se refiere a la velocidad con la que un sistema acepta o responde a una solicitud, incluso antes de completar todo el procesamiento de la solicitud. Es el tiempo que transcurre desde que se realiza una solicitud hasta que el sistema proporciona una confirmación o "acknowledge."

- Latencia (Latency): se refiere al tiempo mínimo requerido para obtener cualquier tipo de respuesta de un sistema. Puede variar según si el sistema es local (en la misma ubicación física) o remoto (accedido a través de una red).
- Rendimiento (Throughput): se refiere a la cantidad de trabajo que un sistema puede realizar en una unidad de tiempo determinada. Puede medirse en términos de bytes por segundo (por ejemplo, velocidad de transferencia de datos) o transacciones por segundo (por ejemplo, procesamiento de solicitudes).
- Carga o estrés (Load/Stress): se utiliza para medir cuán estresado o cargado está un sistema en función de su capacidad para manejar múltiples usuarios o solicitudes simultáneas. Por lo general, se utiliza en el contexto de otras métricas de rendimiento, como el tiempo de respuesta. Por ejemplo, un sistema podría tener un tiempo de respuesta de 0.5 segundos con 10 usuarios y 2 segundos con 20 usuarios, lo que indica cómo responde el sistema bajo diferentes cargas.
- Eficiencia (Efficiency): la eficiencia se calcula dividiendo el rendimiento del sistema (por ejemplo, transacciones por segundo) entre los recursos utilizados para lograr ese rendimiento. Por ejemplo, el sistema A podría realizar 20 transacciones por segundo (tps) con 2 unidades centrales de procesamiento (CPUs), mientras que el sistema B realiza 40 tps con 4 CPUs. Ambos sistemas tienen un rendimiento similar, pero el sistema B utiliza más recursos para lograrlo.
- Escalabilidad (Scalability): la escalabilidad mide cómo se ve afectada la performance de un sistema cuando se agregan nuevos recursos. Puede manifestarse de dos maneras:
    - Scale Up: agregar más o mejores recursos a un servidor existente para mejorar su rendimiento. Por ejemplo, aumentarla capacidad de memoria o CPU en un servidor.
    - Scale Out: Agregar más servidores para distribuir la carga y aumentar la capacidad total. Por ejemplo, implementar una arquitectura en la nube o un clúster.
- Capacidad (Capacity): la capacidad se refiere a la carga máxima o el máximo rendimiento que un sistema puede manejar de manera efectiva. Existe un punto en el que la performance se vuelve inaceptable debido a limitaciones de recursos o diseño. Identificar este punto es esencial para garantizar que el sistema pueda satisfacerlas necesidades de la organización y los usuarios.