

UNIDAD 1

HERRAMIENTAS DE CONTROL DE VERSIONES

TERMINOLOGÍA

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

GESTIÓN DE LA 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.

La **administración de la configuración** (CM, por las siglas de *configuration management*) se ocupa de las políticas, los procesos y las herramientas para administrar los sistemas cambiantes de software.

- Es necesario gestionar los sistemas en evolución porque es fácil perder la pista de cuáles cambios y versiones del componente se incorporaron en cada versión del sistema.

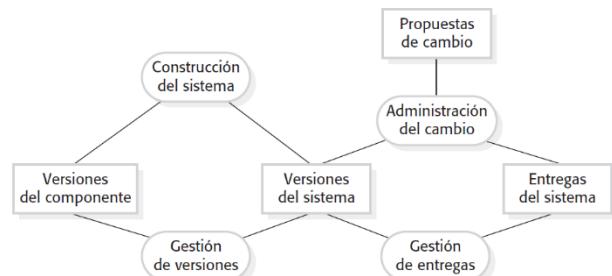
Si no se cuenta con procedimientos efectivos de administración de la configuración, se puede malgastar esfuerzo al modificar la versión equivocada de un sistema, entregar a los clientes la versión incorrecta de un sistema u olvidar dónde se almacena el código fuente del software para una versión particular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

- 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:

- **Gestión de versiones** Esto incluye hacer un seguimiento de las numerosas versiones de los componentes del sistema y garantizar que los cambios hechos por diferentes desarrolladores a los componentes no interfieran entre sí.
- **Construcción del sistema** Éste es el proceso de ensamblar los componentes del programa, datos y bibliotecas, y luego compilarlos y vincularlos para crear un sistema ejecutable.
- **Gestión de entregas (release)** Esto implica preparar el software para la entrega externa y hacer un seguimiento de las versiones del sistema que se entregaron para uso del cliente.
- **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

La **gestión de versiones** (VM, por las siglas de *version management*) es el proceso de hacer un seguimiento de las diferentes versiones de los componentes de software o ítems de configuración, y los sistemas donde se usan dichos componentes.

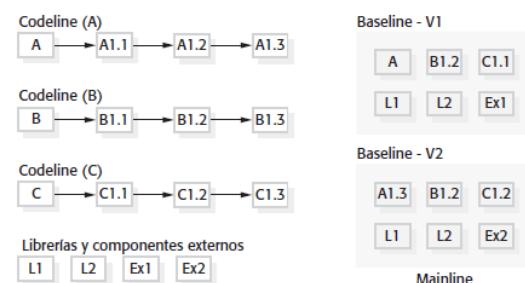
"El proceso por el cual se gestionan los codelines y las baselines"

La figura 25.6 ilustra las diferencias entre línea de código y línea base. En esencia, una **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.

CODELINE: "Conjunto de archivos fuente y otros ítems de configuración que conforman un componente de software a lo largo del tiempo mientras cambian"

"Un codeline contiene todas las revisiones de todos los ítems a lo largo de un camino evolutivo de un componente"

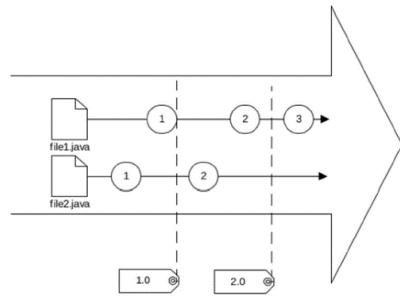
"Una baseline es la definición específica de un sistema"



La **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.

Un **snapshot** contiene una revisión específica de cada ítem en un codeline. Un snapshot de un codeline se dice que es una versión.

Este es un ejemplo de codeline compuesto por dos archivos de configuración (file1.java y file2.java). Cada vez que hago commit, check in o guardo en el sistema de control de versiones creo una nueva versión del archivo (1,2..). Esto es un codeline: un conjunto de todas las versiones de un ítem de configuración mientras cambia a lo largo del tiempo.



HERRAMIENTAS DE VERSIONADO

SISTEMA DE CONTROL DE VERSIONES: "sistemas que identifican, almacenan y controlan el acceso a las diferentes versiones de los ítems de configuración"

Los sistemas de control de versiones (o VCS por sus siglas en inglés) 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.

VCS (VERSION CONTROL SYSTEM) – FUNCIONALIDAD

- **Identificación de Versiones y Releases:** Crea versiones de los ítems cuando estos se envían al sistema.
- **Registro del histórico 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 DE LOS SISTEMAS DE CONTROL DE VERSIONES

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 (VCS) han experimentado grandes mejoras en las últimas décadas y algunos son mejores que otros. Una de las herramientas de 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.

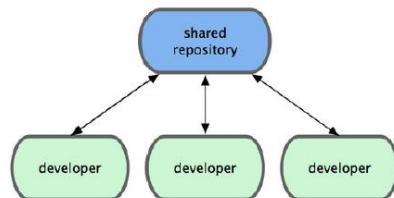
- **Historial completo de cambios a largo plazo de todos los archivos.** Los cambios incluyen la creación y la eliminación de los archivos, así como los cambios de sus contenidos. Este historial también debería incluir el autor, la fecha y notas escritas sobre el propósito de cada cambio. **Tener el 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.

REPOSITORIO Y WORKSPACE

Soporte de desarrollo independiente:

El proyecto será desarrollado por un equipo de desarrolladores sobre un mismo módulo, por ejemplo. Por ello, debemos entender que existe el concepto de **repositorio**: 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**). Una vez concluido su cambio será subido al espacio común (repositorio).



¿QUÉ ES UN ÁREA DE TRABAJO PRIVADA (WORKSPACE)?

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 ítems (OJO, DEPENDE SI ES VSC CENTRALIZADO O DISTRIBUIDO)

Además, debería contener un mecanismo para construir ejecutables a partir de su contenido: Source code, source code for tests, libraries, scripts to build. Puede ser manejado en el contexto de una IDE.

BRANCHING AND MERGING

Al comenzar 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 (**branch**). Esto es normal durante el desarrollo del sistema dónde diferentes usuarios trabajan de manera independiente en diferentes versiones del código y por lo tanto 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.

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, realizando una operación **merge**. 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.

CARACTERISTICAS DE LOS VCS

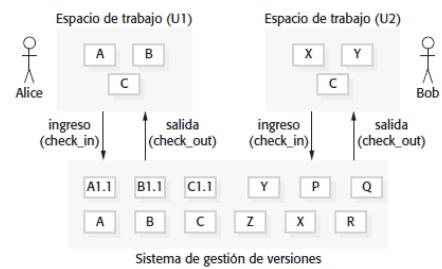
SEGÚN LA ARQUITECTURA

→ **CENTRALIZADO**, por ejemplo: Subversion.

Existe un único repositorio central que contiene todos los ítems de configuración y su historia. Si quiero trabajar sobre este, debo sacar una copia de una versión específica, trabajar en mi espacio de trabajo privado y cuando termino hacer un check in.

Desventajas:

- Es un único 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

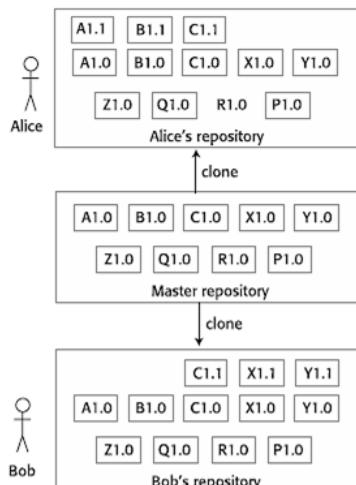


→ **DISTRIBUIDO**, por ejemplo: Git.

Existen copias del mismo repositorio en varias máquinas, trabajo sobre distintas copias de este y después los combino.

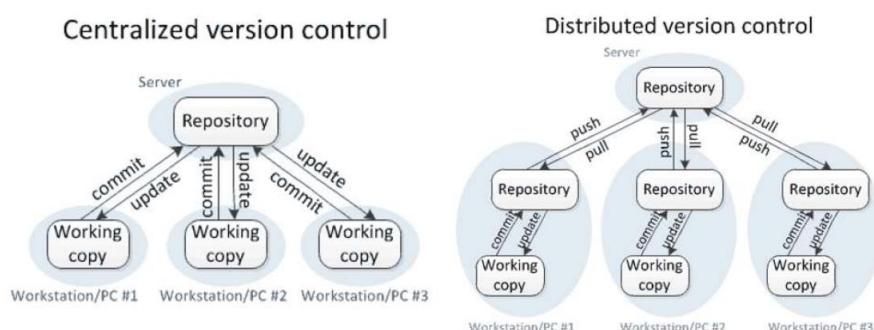
El desarrollador:

- Crea un clon
- instalado en la máquina del desarrollador de manera local
- Trabaja en los archivos y mantienen las nuevas versiones en su repositorio privado en su computadora
- Hacen commit de los cambios y actualizan su repositorio privado
- Hace push a un repositorio remoto de los cambios y por ejemplo sincronizar con el servidor principal del equipo



Ventajas:

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



SEGÚN LA ATOMICIDAD

De acuerdo a si el sistema de control de versiones me 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

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.

Flujo esperado

- 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.
- Bob está bloqueado y no puede continuar. Se va a tomar una taza de café?.
- Developer1 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 checkout y hay que mergear los cambios antes de poder hacer commit

Flujo esperado

- Developer1 hace checkout del archivo foo.c y comienza a modificarlo
- Developer2 hace checkout del archivo foo.c y comienza a modificarlo
- Developer1 termina los cambios y hace checkin
- Developer2 termina los cambios y cuando intenta hacer checkin, el VCS le dice que la version 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 version mergeada

→ COMMIT-BEFORE-MERGE (el que usa git)

El VCS nunca bloquea un commit. Si la versión del repo ha cambiado desde el checkout se crea un nuevo 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 éste. Esto da lugar a la experimentación por la disponibilidad de los branches y la posibilidad de mergear cuando crea conveniente.

GENERACIONES DE VCS

Estas características de centralizado vs distribuido, operaciones por archivo o por conjunto de archivos, manejo de concurrencia con logs, Merge-before-commit y Commit-before-merge dieron lugar a lo que llamamos **generaciones de sistemas de control de versiones**.

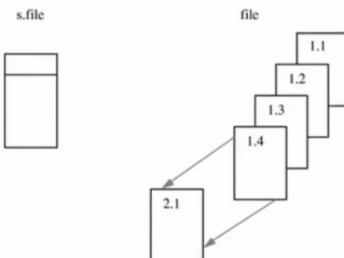
Generation	Networking	Operations	Concurrency	Examples
First	None	One file at a time	Locks	RCS, SCCS
Second	Centralized	Multi-file	Merge before commit	CVS, SourceSafe, Subversion, Team Foundation Server
Third	Distributed	Changesets	Commit before merge	Bazaar, Git, Mercurial

GENERACIONES DE VCS - 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.



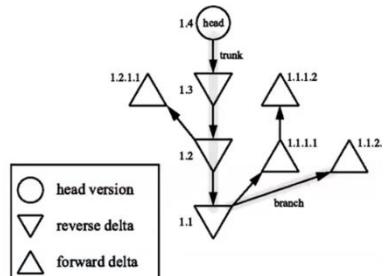
Desventaja:

- Deltas por archivo y no por conjunto de archivos.

GENERACIONES DE VCS - RCS

Buscaba mejorar el problema de SCCS: cuantos más cambios en el archivo, se volvía más lento construir las diferentes versiones. No me interesa que 10 versiones atrás sean rápidas en acceder, pero si las ultimas 5, por ejemplo.

Por ello, 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, teníamos que hacer un forward delta. Entonces, teníamos que 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

GENERACIONES DE VCS - SUBVERSION

Características:

- 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

GENERACIONES DE VCS – GIT

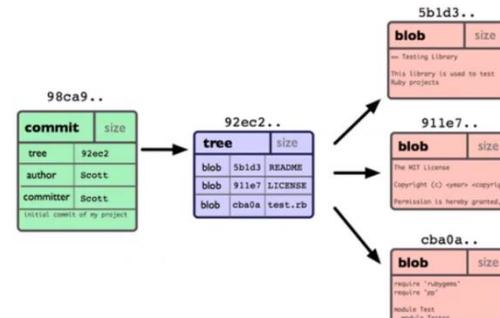
Decimos que es distinto a los anteriores porque no se centra en ahorrar espacio utilizando deltas.

Estructura

Entonces, un commit se compone por tres partes:

- Un **commit** que contiene información del autor, comentario sobre el cambio y un puntero al objeto de tipo **tree**.
- El **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.
- Los 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, son simplemente contenido.

Git toma el contenido, lo mete en un archivo binario, un compress file, calcula un hash para identificarlo únicamente y luego lo agrega en el árbol.



¿Por qué 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, no guarda deltas sino que reutiliza los contenidos.

Características:

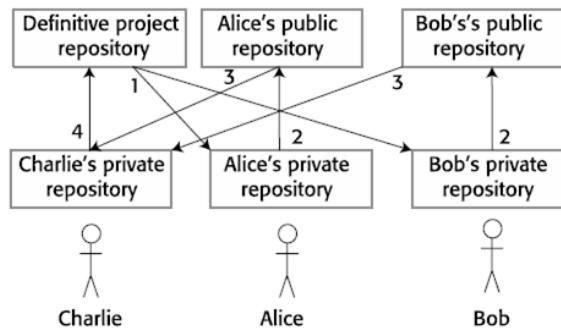
- Commit-before-merge
- Operación por conjunto de archivos
- Distribuido

Ventajas:

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

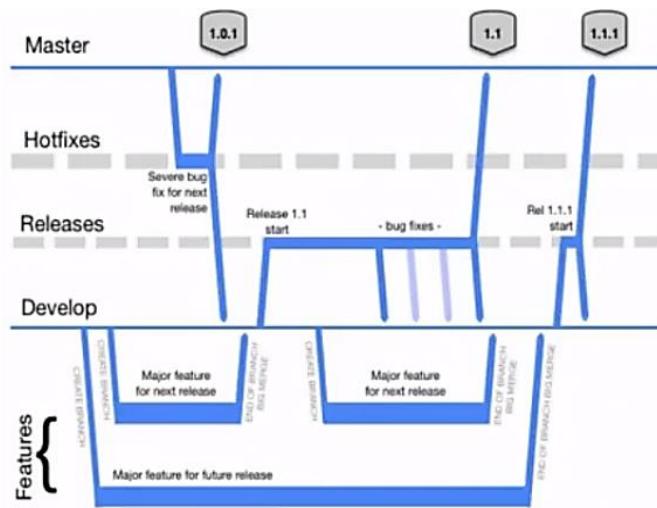
DESARROLLO OPEN SOURCE

- Los VCS distribuidos son esenciales para el desarrollo opensource/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



BRANCHING MODELS – GITHUB FLOW

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 trabajaré y luego con un pull request lo uno al Develop.
- Una vez que tengo todas las features que van a ir juntas a un release, promuevo ese código a una **Release branch**. Probablemente, en esta branch corramos testing, validaciones para descubrir bugs.
- Una vez corregidos los bugs, lo muevo a **Master** y ese es nuestro release.
- También existe el **Hotfixes 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 más pesado, más lento.
- Tengo muchas versiones de mi producto. Los branches están alejados unos de otro y los cambios se unifican en momentos determinados. **NO ESTÁ CONTINUAMENTE INTEGRADO**.

RECOMENDADO CUANDO:

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

NO RECOMENDADO CUANDO:

- En los start ups.
- Se requiere iterar rápidamente.
- Los desarrolladores son en su mayoría seniors.

“Hagamos todos los cambios en master/main”

CARACTERISTICAS

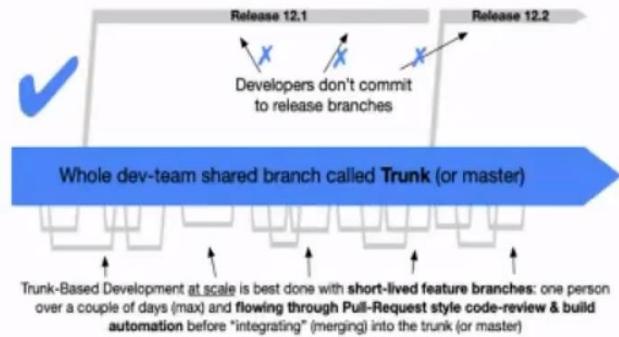
- Más flexible.
- Escala mejor.
- Permite hacer integración continua.

RECOMENDADO CUANDO

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

NO RECOMENDADO CUANDO

- El equipo no tiene experiencia.
- Muchos junior developers.
- El producto es estable.



UNIDAD 2

DISEÑO DE SISTEMAS DISTRIBUIDOS

INTRODUCCIÓN

Prácticamente todos los grandes sistemas basados en computadora son ahora sistemas distribuidos: cada vez que vemos una película on demand, compramos online, pedimos un taxi usando nuestro Smartphone, buscamos algo online. Todas esas 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...) es un sistema distribuido diseñado para las compañías y desarrolladores de software.

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.

Ventajas de usar un enfoque distribuido para el desarrollo de sistemas:

- **Compartición de recursos:** Un sistema distribuido permite compartir los recursos de hardware y software, tales como discos, impresoras, archivos y compiladores, que se asocian con computadoras en una red.
- **Apertura:** Los sistemas distribuidos, 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 un sistema distribuido, 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.

Desventaja: 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.

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.

ESCALABILIDAD:

Una medida que indica como se ve afectada la performance cuando se agregan nuevos recursos

Con relativamente pocos usuarios el tiempo de respuesta es aceptable. Pero, cuando la base de usuarios crece, el tiempo de respuesta se degrada considerablemente. 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. Ej: el RT = 0.5 s con 10 usuarios y 2 s con 20 usuarios

→ **Escalabilidad Vertical**

Compramos el servidor más avanzado, con el máximo de memoria y de poder de procesamiento posibles, pero también el más costoso.

Scale Up: 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. Neuman (1994) identifica 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) y ampliar (scaling out). Lo primero significa sustituir recursos en el sistema con recursos más poderosos. Ampliar 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
 - Vertical
 - Limitada
- Un único punto de fallo
 - Error: si algo se rompe o surge un error, para el resto de las actividades.
 - Mantenimiento: si tengo que actualizar el software, debo parar el resto de las actividades.
- Localización
 - Aumenta la latencia con la distancia, porque los saltos en la red van a ser más.
 - Experiencia de usuario degradada para usuarios remotos: más latencia, más ancho de banda.

CONFLICTOS 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
 - Para manejar el aumento de carga
 - Para manejar el aumento de datos
- Es altamente escalable
 - Idealmente de manera lineal: cuando agrego una cantidad de recursos, aumento la capacidad en misma proporción.
 - Aumentar y disminuir la capacidad según la demanda
- Es altamente disponible
 - No tiene un solo punto de falla
- Experiencia de usuario
 - Latencia independiente de dónde se encuentra el usuario

"Un sistema distribuido 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."

CONCEPTOS IMPORTANTES

Concurrencia

Todas las computadoras operan al mismo tiempo

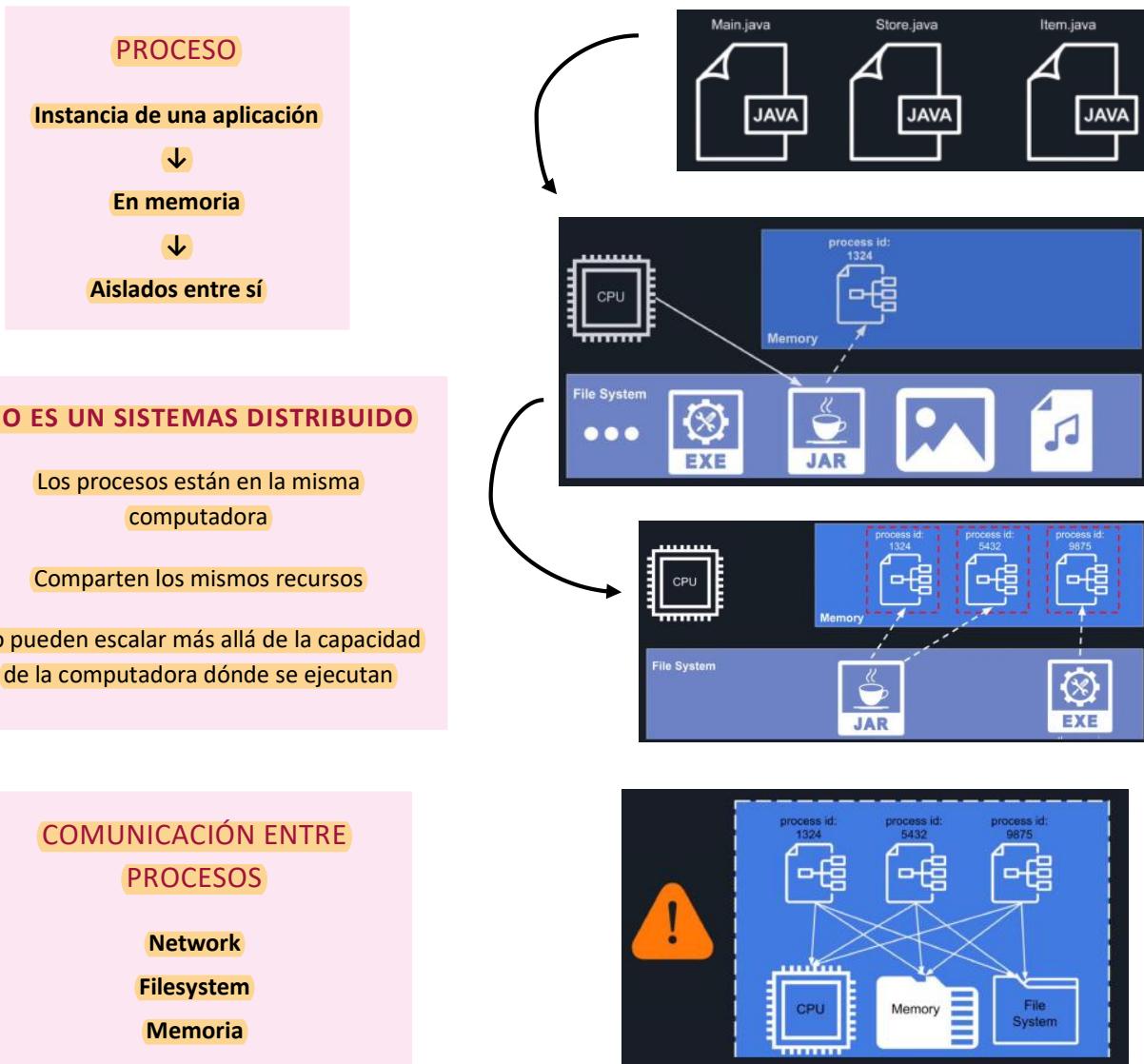
Independencia

Las computadoras funcionan y fallan de manera independiente

No hay reloj global

Las computadoras no comparten un reloj global

PROCESOS



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 Fallo y Confiabilidad

- **Tolerancia a fallos**: Aunque algunas computadoras fallen, otros procesos siguen corriendo en las computadoras que continúan funcionando
- **Mantenimiento**: 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 confinable por definición

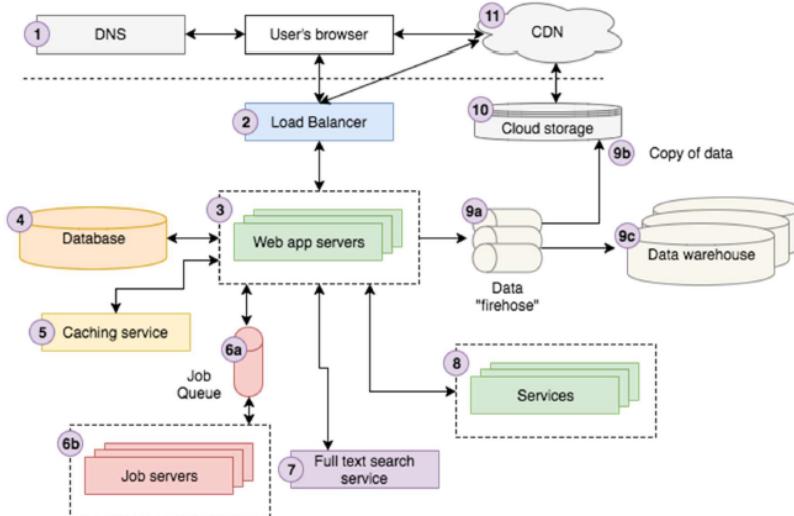
- Puede no estar disponible
- Tener una gran latencia
- Estar congestionada

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

¿Qué es un patrón de diseño?

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.

¿Qué es un patrón arquitectónico?

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 y la organización de los componentes de la arquitectura.

¿Cuál es la diferencia entre un patrón de diseño OO y un patrón de arquitectura? Si este es relevante a la totalidad del sistema entonces hablamos de un **patrón de arquitectura (alto nivel)**; en cambio, si este sólo concierne a un subcomponente, nos referimos a un **patrón de diseño (bajo nivel)**.

PATRONES DE DISEÑO DE SOFTWARE	PATRONES DE ARQUITECTURA DE SOFTWARE
Clases Resuelven un problema puntual	Componentes Entender como encajan las principales partes Como fluye la información Otros aspectos estructurales
AbstractFactory State Strategy Proxy Monads	En capas Service Oriented Microservices Microkernel

COMPUTACIÓN 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 figura 18.4. Ésta 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.

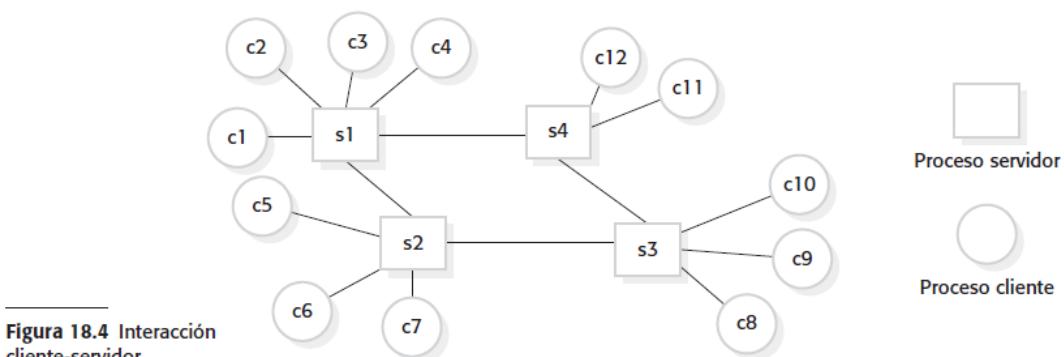


Figura 18.4 Interacción cliente-servidor

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 figura 18.6 ilustra este modelo y muestra una aplicación estructurada en cuatro capas:

- Una capa de presentación que se ocupa de presentar la información al usuario y gestionar todas las interacciones de usuario;
- Una 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, etcétera;
- Una capa de procesamiento de aplicación que se ocupa de implementar la lógica de la aplicación y, de este modo, proporciona la funcionalidad requerida a los usuarios finales;
- Una capa de base de datos que almacena los datos y ofrece servicios de gestión de transacción, etcétera.

Capa de presentación

Capa de gestión de datos

Capa de procesamiento de aplicación

Capa de base de datos

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

Beneficio: cada capa puede ser reemplazada o modificada sin afectar toda la arquitectura

Otros beneficios:

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

Desventaja:

Si tengo que agregar una nueva función de negocios, por ejemplo, tengo que modificar las otras capas. Agrego una funcionalidad como el manejo de ítems, tengo que agregar una presentación y modificar la gestión de los datos.



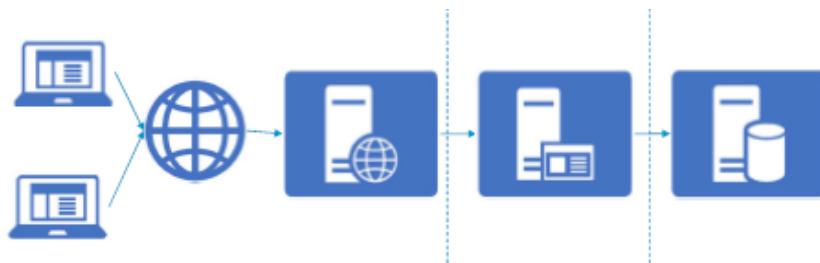
Puedo agregar otros tipos de presentación, no solo el HTML original, también una app web o una consola, pero que hablen con la misma capa inferior. La lógica de negocios no cambia: si tiene que mostrar una lista con los clientes o dar de alta un proveedor no importa donde sea, si una página web o una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

ARQUITECTURA N-CAPAS (N-TIER)

Una arquitectura de N niveles 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: tener varios application server, por ejemplo.

Una aplicación de N niveles puede tener una **arquitectura de capa cerrada** o una **arquitectura de capa abierta**:

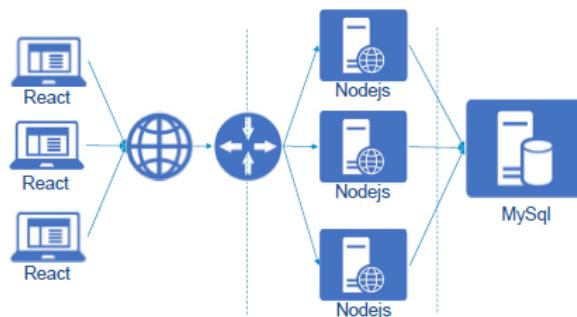
- En una arquitectura de **capa cerrada**, una capa solo puede llamar a la siguiente capa inmediatamente hacia abajo.
- En una arquitectura de **capa abierta**, una capa puede llamar a cualquiera de las capas debajo de ella.

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)

Desafíos

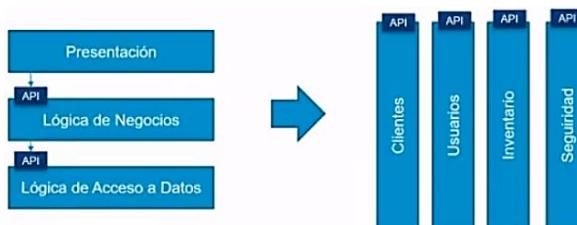
- Es fácil terminar con un nivel intermedio que solo realiza operaciones CRUD en la base de datos, lo que agrega latencia adicional sin realizar ningún trabajo útil.
- El diseño monolítico evita la implementación independiente de funciones. Para escalar horizontalmente tendría que clonar la lógica de negocios.



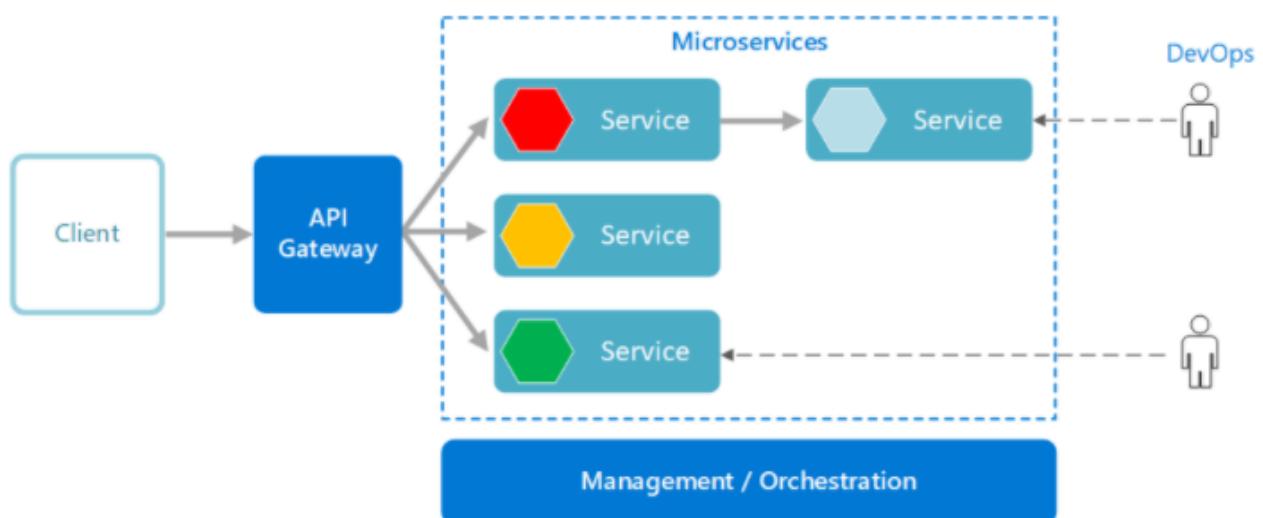
- Puede resultar difícil administrar la seguridad de la red en un sistema grande.

ARQUITECTURA MICROSERVICIOS

Rompiendo 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.



¿Qué son los microservicios?

- Los microservicios son pequeños, independientes y poco acoplados. Un solo equipo pequeño de desarrolladores puede escribir y mantener un servicio.
- 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.

Beneficios

- **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.

Desafíos

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.
- **Desarrollo y prueba.** Escribir un servicio pequeño que depende de otros servicios dependientes requiere un enfoque diferente al de escribir una aplicación tradicional monolítica o en capas. Las herramientas existentes no siempre están diseñadas para trabajar con dependencias de servicios. La refactorización a través de los límites del servicio puede resultar difícil. También es un desafío probar las dependencias de los servicios, especialmente cuando la aplicación evoluciona rápidamente.
- **Falta de gobernanza.** El enfoque descentralizado para crear microservicios tiene ventajas, pero también puede generar problemas. Puede terminar con tantos lenguajes y marcos diferentes que la aplicación se vuelva difícil de mantener. Puede ser útil implementar algunos estándares para todo el proyecto, sin restringir demasiado la flexibilidad de los equipos. Esto se aplica especialmente a funciones transversales como el registro.
- **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. Deberá diseñar las API con cuidado. Evite las API demasiado conversadoras, piense en los formatos de serialización y busque lugares para usar patrones de comunicación asíncrona.
- **Gestión.** Para tener éxito con los microservicios se requiere una cultura DevOps madura. El registro correlacionado entre servicios puede ser un desafío. Normalmente, el registro debe correlacionar varias llamadas de servicio para una sola operación de usuario.

- **Control de versiones.** Las actualizaciones de un servicio no deben interrumpir los servicios que dependen de él. 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.

LATENCIA

La podemos definir como **el tiempo que transcurre entre una orden y la respuesta que se produce a esa orden concreta.**

La latencia en redes, mide la suma de retados que se producen desde que solicitamos una información (o la enviamos) y el nodo remoto nos responde. Dicho de otra forma, mide el tiempo que tarda un paquete de datos en llegar de un lugar a otro. Este tiempo se mide en milisegundos.

¿Qué es un servicio?

A service is a self-contained unit of software that performs a specific task. It has three components: an interface, a contract, and implementation.

CARACTERISTICAS DE LOS SERVICIOS:

- **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.

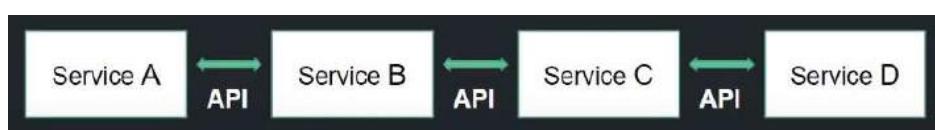
Arquitectura orientada a servicios (SOA)

An approach/paradigm designing/implementing/deploying software systems that are based on distributed resources/assests (i.e. Computation and data)

Un patrón de arquitectura que permite construir sistemas componiendo servicios de manera ligeramente acoplada (i.e. Servicios de negocio, de infraestructura, de seguridad, etc.)

¿Qué es un microservicio?

Microservices are small in size, messaging-enabled, bounded by contexts, autonomously developed, independently deployable, decentralized and built and released with automated processes.



- Trade-offs
 - Los servicios son más simples
 - La arquitectura se vuelve más compleja
- Complejidad manejada con
 - Herramientas
 - Automatización
- Monitoreo

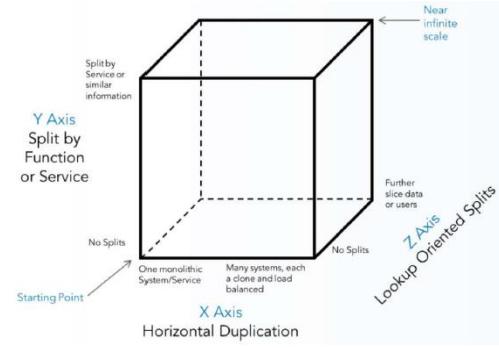
CUBO DE ESCALABILIDAD

El modelo de escalabilidad de tres dimensiones llamado "el cubo de la escalabilidad" es usado para representar de manera simple las formas de escalar una aplicación mas allá del tradicional escalamiento vertical que consiste en aumentar los recursos de procesamiento.

Básicamente, podemos escalar una aplicación según tres ejes: X, 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 es lo que mencionamos 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. Algun 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.

INTEGRACION Y DESPLIEGUE CONTINUO

EL PROBLEMA DE ENTREGA DE SOFTWARE

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 de software: diseño, diseño, desarrollo y pruebas, pero pocos cubren el flujo de valor que ofrece valor a las personas y organizaciones que patrocinan nuestro esfuerzo.

Tomaremos como patrón el proceso de **deployment pipeline**, el cual consiste en la implementación automatizada del proceso de creación (build), implementación (deploy), prueba (test) y lanzamiento (release).

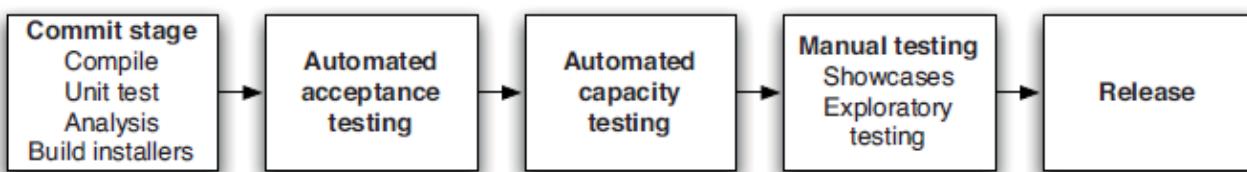


Figure 1.1 *The deployment pipeline*

¿Cómo funciona el deployment pipeline? Cada cambio que se realiza en la configuración de una aplicación, el código fuente, el entorno, o datos, desencadena la creación de una nueva instancia del pipeline. Uno de los primeros pasos en el pipeline es+ crear binarios e instaladores. El resto de este ejecuta una serie de pruebas en los binarios para demostrar que se pueden publicar. Cada prueba que el candidato de release pase nos da más confianza en que esta información de configuración, entorno y datos trabajará. Si el candidato pasa todas las pruebas, puede ser lanzado.

Los **objetivos** del deployment pipeline son tres:

- Hace que cada parte del proceso de construcción, implementación, prueba y lanzamiento de software visible para todos los involucrados, derivando en la colaboración.
- Mejora la retroalimentación para que los problemas que se identifican puedan ser resueltos lo antes posible en el proceso.
- Permite a los equipos implementar y lanzar cualquier versión de su software para cualquier entorno a voluntad a través de un proceso totalmente automatizado.

CYCLE TIME (CT)

“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.

Clave para mejorar el CT → Automatización, buenas prácticas y patrones

ALGUNOS ANTIPATRONES DE RELEASE COMUNES

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:

- 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

Muchas cosas pueden salir mal

ANTIPATRÓN - DESPLIEGUE MANUAL

La mayoría de las aplicaciones modernas de cualquier tamaño son complejas de implementar, lo que 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.

¿Cómo reconocer este antipatrón?:

- 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 release durante el transcurso de una publicación.
- **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 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 A. M. el día del lanzamiento, tratando de averiguar cómo hacerlo funcionar.

¿Por qué 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**.
- Debe **documentarse** un proceso de implementación manual.

Mantener la documentación es una tarea compleja y que 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, usted está en problemas.
- Hemos oido decir que un **proceso manual es más auditabile que un uno automatizado**. 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 ser 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.

ANTIPATRÓN - NO TESTEAR EN UN ENTORNO COMO PRODUCCIÓN

En este patrón, el software se implementa en un entorno de producción (por ejemplo, la puesta en escena – release staging) una vez que el trabajo de desarrollo ha finalizado, no antes.

¿Cómo reconocer este antipatrón?:

- Si los testers han estado involucrados en el proceso hasta este punto, han probado el sistema en entornos de desarrollo.
- En la puesta en escena la gente de operaciones interactúa con la nueva versión.

En algunas organizaciones, se utilizan equipos de operaciones separados para implementar el software en etapas y producción.

- 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.

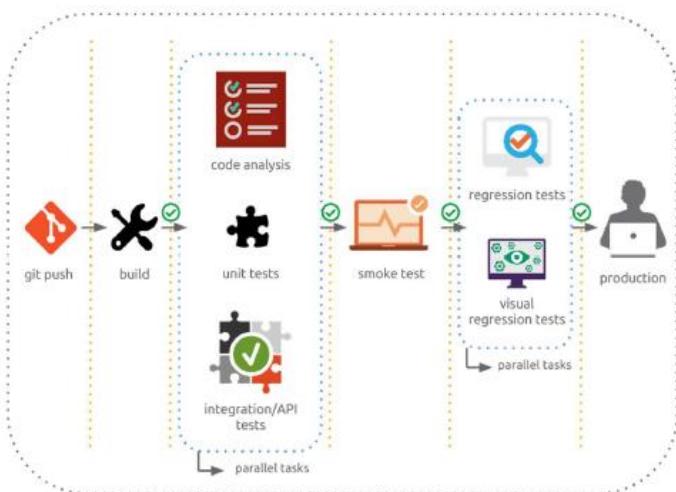
Una vez que la aplicación se implementa en staging, es común que aparezcan nuevos errores. Desafortunadamente, a menudo no hay tiempo para arreglarlos todos porque el plazo se acerca rápidamente y, en esta etapa del proyecto, diferir la fecha de lanzamiento es inaceptable. De modo que los errores más 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:

- Integrar las actividades de prueba, implementación 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.



ANTIPATRÓN - ADMINISTRACIÓN MANUAL DE LA CONFIGURACIÓN EN PRODUCCIÓN

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 subprocessos en un grupo de subprocessos en un servidor de aplicaciones, luego se lleva a cabo manualmente en los servidores de producción.

¿Cómo reconocer este antipatrón?:

- Habiendo desplegado con éxito muchas veces a la puesta en escena, el despliegue en producción falla.
- Los diferentes miembros de un clúster se comportan de manera diferente, por ejemplo, un nodo soporta menos carga o tarda más en procesar solicitudes que otro.
- 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:

- 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. La virtualización puede ayudarlo a comenzar con esto.

- Cada cambio hecho a producción debe ser **registrado y auditable**. A menudo, las implementaciones fallan porque alguien parcheó el entorno de producción la última vez que implementó, pero el cambio no se registró. De hecho, no debería ser posible hacer cambios manuales en los entornos de prueba, ensayo y producción. La única forma para realizar cambios en estos entornos se debe realizar un proceso automatizado.

¿Se puede mejorar? El lanzamiento de software puede, y debe, ser de bajo riesgo, proceso frecuente, barato, rápido y predecible.

¿Cómo lo logramos?

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. La calidad no es igual a la perfección, como dijo Voltaire, "lo perfecto es el enemigo de lo bueno", pero nuestro objetivo siempre debe ser ofrecer software de calidad suficiente para aportar valor a sus usuarios. Entonces, si bien es importante entregar nuestro software lo antes posible, es fundamental mantener un adecuado nivel de calidad.

Por lo tanto, para refinar ligeramente nuestro objetivo, queremos encontrar formas de ofrecer alta calidad, software valioso de una manera eficiente, rápida y confiable.

para lograr estos objetivos (tiempo de ciclo bajo y alta calidad), debemos realizar cambios frecuentes y automatizados lanzamientos de nuestro software. ¿Por qué es esto?

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. Esto significa que hay no hay forma de obtener control sobre el proceso de liberación y, por lo tanto, de garantizar un alto calidad.
- 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. Gran parte de este libro se concentra en recibir comentarios sobre los cambios en su aplicación y su configuración (incluido su entorno, proceso de implementación y datos) lo más rápido posible.

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

- **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.

¿Qué es 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.

→ **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. Si tiene procesos manuales, depende de las personas para obtener el trabajo hecho. Además, realizar procesos manuales de creación, prueba e implementación es aburrido y repetitivo, lejos de ser el mejor uso para las personas.

Sin embargo, el deployment pipeline requiere muchos recursos, especialmente una vez que tenga un conjunto completo de pruebas automatizadas. Podemos caracterizar las pruebas en esta etapa como sigue:

- Commit-stage
 - Tests de ejecución rápida
 - Que sean independientes del environment
 - Tener un test coverage alto
 - Si estos tests fallan no se puede releesar el software
- Tests funcionales
 - Son más lentos
 - Si fallan aún se puede elegir releesar 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

→ **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. Un proceso basado en la mejora continua es esencial para la entrega rápida de software de calidad. Los procesos iterativos trabajan con este tipo de actividad: en al menos una vez por iteración se lleva a cabo una reunión retrospectiva donde todos discuten cómo mejorar el proceso de entrega para la próxima iteración.

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. Esto requiere disciplina y planificación. 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.

Beneficios de este enfoque:

- 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 provenientes de una pobre Administración de la Configuración
- Reduce el stress porque las tareas se hacen triviales, repetibles y predecibles
- Flexibilidad de despliegue se vuelve trivial iniciar la app en un nuevo entorno
- La práctica hace a la perfección: usar el mismo mecanismo de deployment en todos lados

CANDIDATO A RELEASE

Es el proceso de construcción, implementación y prueba que aplicamos al cambio que valida que puede ser lanzado.

Si bien cualquier cambio puede dar lugar a un artefacto que se puede liberar a los usuarios, no empieces 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 establecidos por el cliente, entonces puede ser liberado.

La mayoría de los enfoques para lanzar software identifican candidatos de lanzamiento al final del proceso.



Figure 1.2 Traditional view of release candidates

Los enfoques tradicionales para el desarrollo de software retrasan la nominación de un candidato de liberación hasta que se hayan tomado varios pasos largos y costosos para asegúrese de que el software sea de suficiente calidad y funcionalmente completo. En esta etapa, la calidad de la aplicación suele ser significativamente mayor, por lo que la prueba manual es solo una afirmación de la integridad funcional.

Cada check in debería ser un release candidate.



PRINCIPIOS DE LA ENTREGA DE SOFTWARE

CREE UN PROCESO CONFIABLE Y REPETIBLE PARA LANZAR SOFTWARE

Liberar el software debería ser fácil. 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

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.

Sin embargo, la lista de las cosas que no se pueden automatizar son mucho más pequeñas de lo que mucha gente piensa. 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

- La integración es a menudo un proceso muy doloroso. Si esto es cierto en su proyecto, integre cada vez que alguien se registra y lo hace desde el inicio del proyecto.
- Si el testing es un proceso doloroso que ocurre justo antes del lanzamiento, no lo hagas al final. En cambio, hazlo continuamente desde el inicio del proyecto.
- Si lanzar software es doloroso, intente lanzarlo cada vez que alguien verifique en un cambio que pasa todas las pruebas automatizadas. Si no puede entregarlo a usuarios reales en cada cambio, libérelo en un entorno similar a la producción.
- Si crear la documentación de la aplicación es complicado, hágalo a medida que desarrolla nuevas funciones en lugar de dejarlo para el final. Hacer documentación para una característica parte de la definición de DONE y automatizar el proceso en la medida de lo posible.

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, puro 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.

Este principio tiene un corolario interesante: no está en el poder de una sola persona para hacer algo. Requiere varias personas en un equipo de entrega para trabajar juntos para hacer cualquier cosa.

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 movimiento se centra en el mismo objetivo que establecemos en este libro: 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: 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.

INTEGRACION 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.

¿DÓNDE 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.

- En proyectos con **long-lived branches**
- En proyectos que los **tests de aceptación se dejan para el final**
- Se suele planear **fases de integración largas**, muchas veces porque no se sabe cuánto puede tardar.

PROYECTOS DONDE SE USA LA INTEGRACIÓN CONTINUA

Descripto 1ero → 1999 por Kent Beck en **Xtreme Programming**, 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

Objetivo de CI: 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
- Requiere disciplina por parte del equipo
- Hacer check ins 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 code en el local env
3. Ejecutar el build y los test cases localmente
4. Si el build y los test pasan localmente, hacer checkin
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 checkin 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
 - 5 minutos es mejor
 - Parece contradecir el anterior, diferentes tipos de tests en diferentes etapas -> 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
- Corre los tests localmente antes de hacer commit o usar al building server para que lo haga
 - Update workspace
 - Run local build
 - Run local tests
 - 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 check ins 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.
- CI 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:

- **Equipos de operaciones** y construcción esperan documentación o arreglos.
- **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: acceso a las cosas que necesitan cuando las necesitan y 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.

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

Cada cambio en su 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.

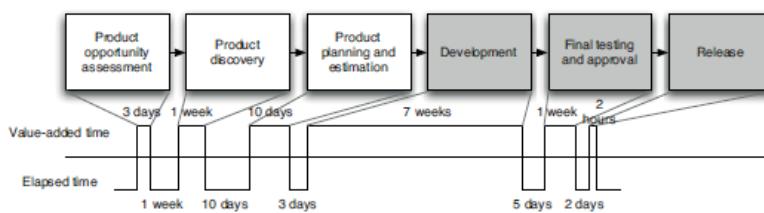


Figure 5.1 A simple value stream map for a product

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 la Figura 5.2

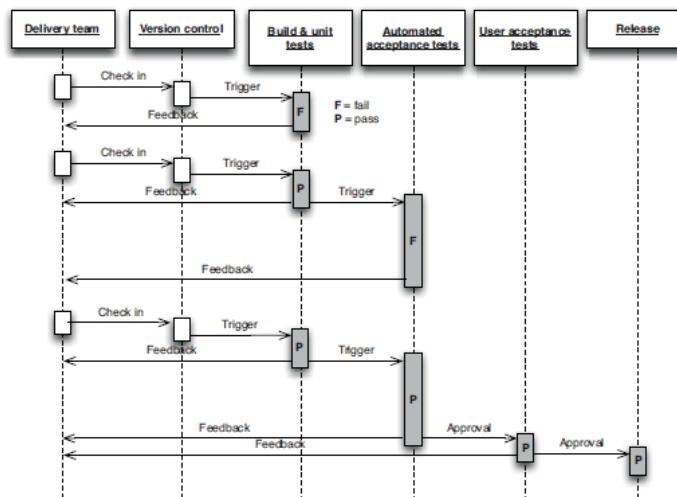


Figure 5.2 Changes moving through the deployment pipeline

- 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
- más lejos llega un build, mayor confianza en la calidad del build
- mayor la cantidad de recursos invertidos en ese
- Env más parecido a prod
- Objetivo: eliminar builds defectuosos
 - Lo más rápido posible
 - Para obtener feedback rápido
 - Builds defectuosos no continúan en el pipeline

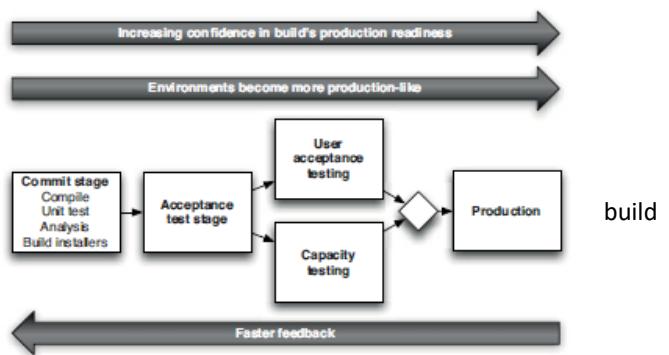


Figure 5.3 Trade-offs in the deployment pipeline

CONSECUENCIAS

Primero

- No se deployan en prod 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

¿Cómo?

Para lograr este estado "enviable"

- Automatizar un conjunto de tests que prueben que nuestro release 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** afirman que el sistema funciona en nivel funcional y no funcional, que conductualmente satisface las necesidades de sus usuarios y las especificaciones del cliente.
- **Las etapas de prueba manual** 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).

Deployment pipeline típico

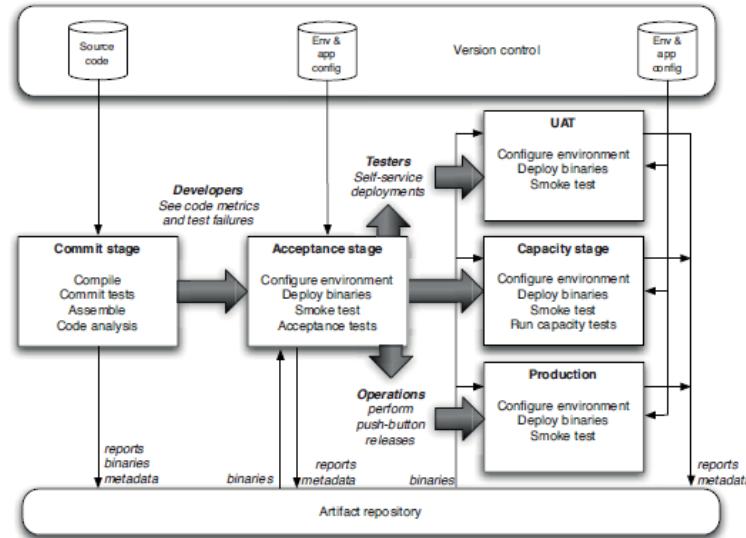


Figure 5.4 Basic deployment pipeline

DEPLOYMENT PIPELINE – PRACTICES

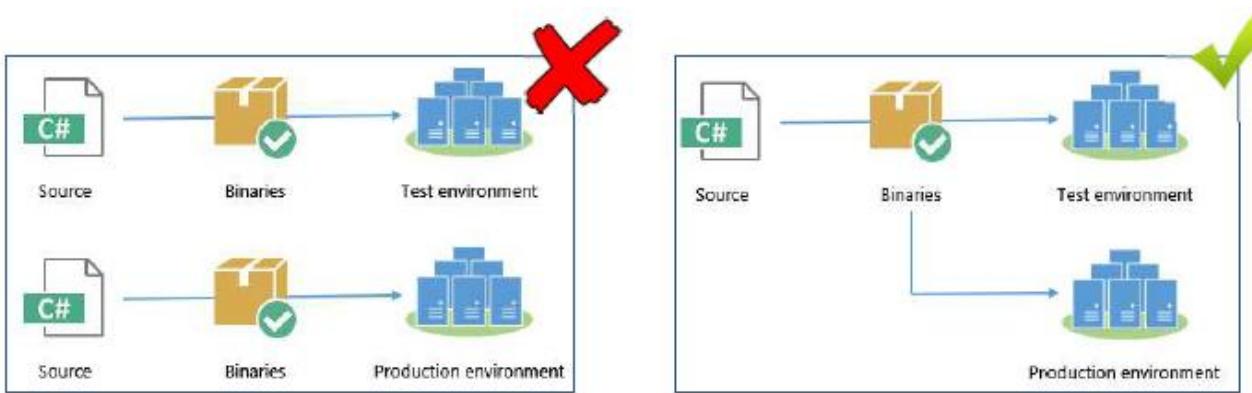
SOLO CREE SUS ARCHIVOS BINARIOS UNA VEZ

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: sistema operativo y ajustes de configuración de middleware, la ubicación de bases de datos y servicios externos, y otra información de configuración que debe establecerse en el momento de la implementación. 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.

Si no funciona, su prueba de humo debería poder brindarle algunos diagnósticos básicos en cuanto a si su aplicación está inactiva porque algo de lo que depende no es trabajando.

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**; por ejemplo, las compilaciones pueden ejecutarse cada hora, la aceptación pruebas nocturnas y pruebas de capacidad durante el fin de semana. 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.

Por supuesto, 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 Figura 5.6.

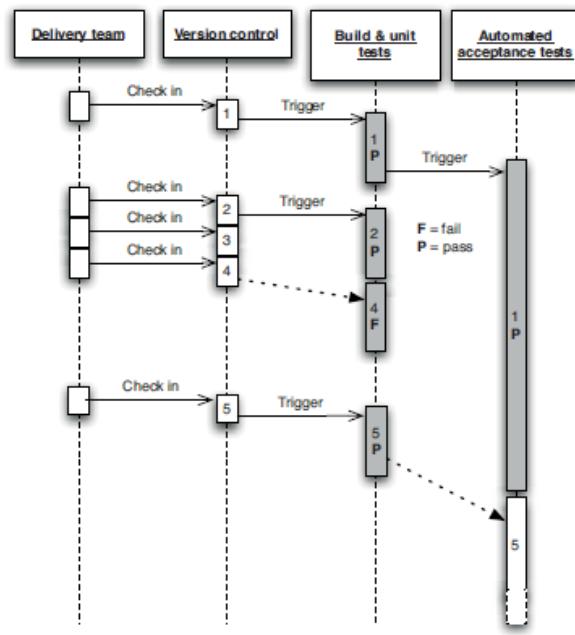


Figure 5.6 Scheduling stages in a pipeline

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 irían 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. Suponga 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.

UNIDAD 4

DISEÑO EN LAS METODOLOGÍAS ÁGILES

INTRODUCCIÓN

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.

- El **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.

"Programas de cómputo y documentación asociada."

"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 (extraído del estándar 729 del IEEE)."

"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**

"Aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software" (IEEE 1993)

Sistemático -> normas, procedimientos

Disciplinado -> Orden y subordinación entre los miembros

Cuantificable -> Medible

En términos de proceso

- ¿Cómo se debe desarrollar software de manera correcta? ¿Cuáles son los pasos a seguir?
- ¿Qué actividades lo componen?
 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.
- ¿Cómo harían el planning?
 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.
- ¿Cuáles son los roles que identifican?

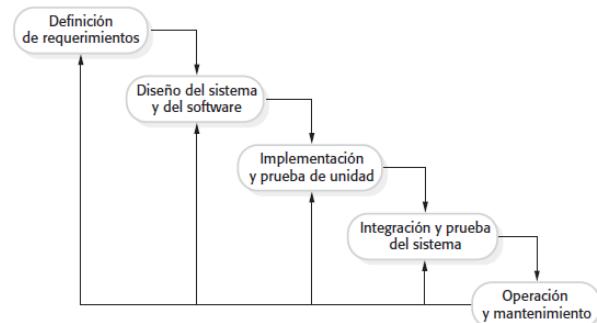
- **Modelo de Proceso de Software**

"Representación simplificada del proceso de software que pueden utilizarse como base para crear procesos de software específicos "

• MODELO EN CASCADA

Desventajas:

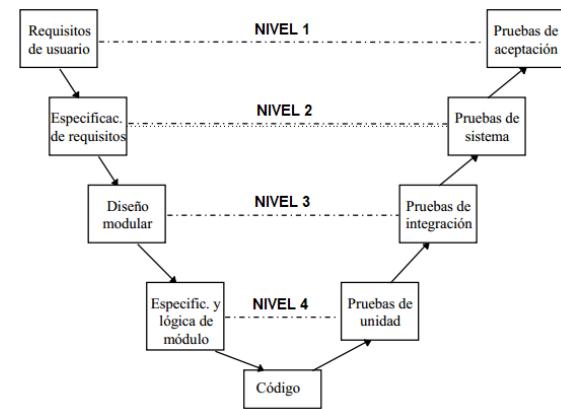
- No refleja realmente el proceso de desarrollo del software
- Se tarda mucho tiempo en pasar por todo el ciclo
- Perpetua el fracaso de la industria del software en su comunicación con el usuario final
- El mantenimiento se realiza en el código fuente
- Las revisiones de proyectos de gran complejidad 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.

- El nivel 1 está orientado al “cliente”. Se compone del análisis de requisitos y especificaciones, se traduce en un documento de requisitos y especificaciones.
- El 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.
- El nivel 3 define los **componentes hardware y software** del sistema final, a cuyo conjunto se denomina arquitectura del sistema.
- El nivel 4 es la **fase de implementación**, en la que se desarrollan los elementos unitarios o módulos del programa.



Ventajas:

- 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 períodos de inactividad al mínimo.

Desventajas:

- 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.

Se recomienda usar los modelos secuenciales cuando los requerimientos se entienden por completo o cuando el software no cambiará (sistema embebido)

AGILE

En la década de 1990 el descontento con estos enfoques engorrosos de la ingeniería de software condujo 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.” [Standish Group] 2012

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 quizás nunca se emplee.

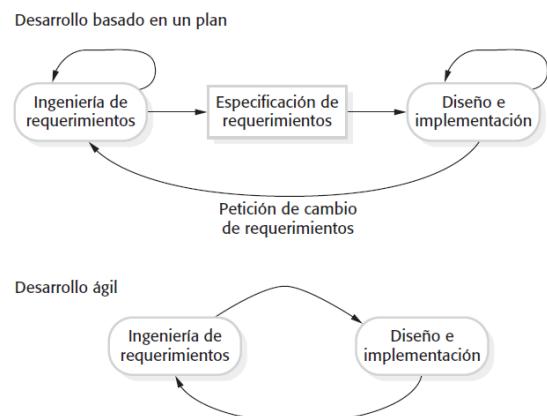
PLAN GUIADO Y EL DESARROLLO ÁGIL

Desarrollo guiado por plan

- Identifica etapas separadas en el proceso de software con salidas asociadas a cada etapa.
- Las salidas de una etapa se usan como base para planear la siguiente actividad del proceso.
- La iteración ocurre dentro de las actividades con documentos formales usados para comunicarse entre etapas del proceso.
- Soporta el desarrollo y la entrega incrementales. Es perfectamente factible asignar requerimientos y planear tanto la fase de diseño y desarrollo como una serie de incrementos.

Desarrollo ágil

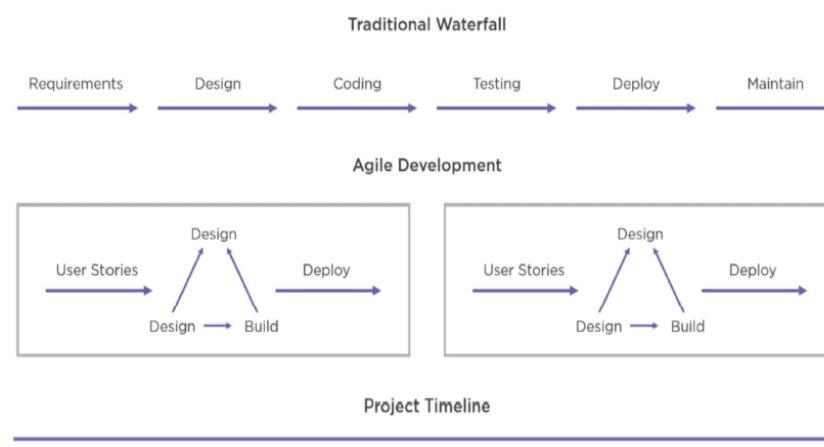
- Consideran el diseño y la implementación como las actividades centrales en el proceso del software. Incorporan otras actividades como la adquisición de requerimientos y pruebas.
- La iteración ocurre a través de las actividades. Por lo tanto, los requerimientos y el diseño se desarrollan en conjunto, no por separado.
- Salidas del proceso decididas mediante negociación durante el proceso de desarrollo de software (feedback)
- Responde al cambio



WATERFALL VS AGILE

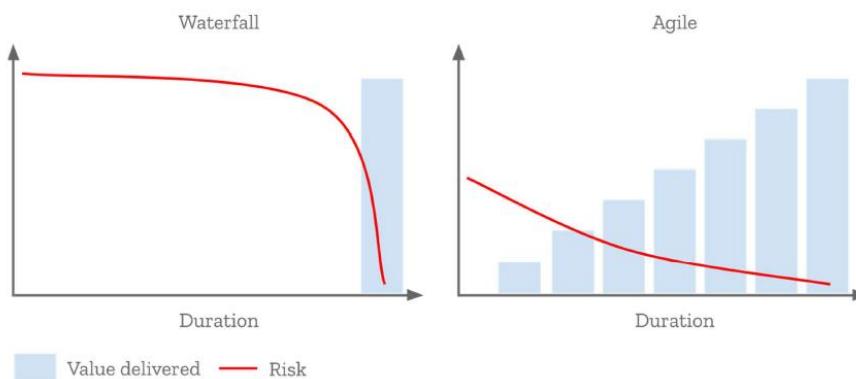
Lo primero que hay que tener en cuenta es que ambas metodologías suponen dos enfoques distintos para administrar y desarrollar un proyecto. '**Agile**' es más colaborativo y enfocado a cambios, siendo una filosofía que implica una forma distinta de trabajar y de organizarse; mientras que '**waterfall**'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

Las 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.



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.

Principio	Descripción
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.

PRINCIPIOS MANIFIESTO ÁGIL

Son 12 principios separados en 3 grupos:

→ Entrega del software en intervalos regulares y frecuentes

La mayor prioridad es satisfacer al cliente a través de la entrega temprana y continua de valioso software

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

El software de trabajo es la medida principal de progreso

Los procesos ágiles promueven el desarrollo sostenible. Los patrocinadores, desarrolladores y usuarios deberían poder mantener un ritmo constante indefinidamente

→ Comunicación en el equipo

"Los empresarios y los promotores deben trabajar juntos diariamente durante todo el proyecto"

"Construir proyectos alrededor de individuos motivados. Dales el entorno y el apoyo que necesitan, y confiar en ellos para hacer el trabajo"

"El método más eficiente y eficaz de transmitir información a y dentro de un desarrollo equipo es una conversación cara a cara"

"Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados"

"A intervalos regulares, el equipo reflexiona sobre cómo para ser más efectivo, luego sintoniza y ajusta su comportamiento en consecuencia".

→ 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

- Los equipos ágiles son equipos de desarrollo de software primero y los miembros de un departamento segundo.
- 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, así que no necesitamos programa de pares, refactor o hacer 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
- No hay ningún plan para abordar la resistencia de los empleados

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
- Los costos pueden aumentar ya que se requieren probadores todo el tiempo en lugar de al final
- Agile puede ser intenso para el equipo

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 para la intensa participación que caracteriza a los métodos ágiles.
- 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.

APLICABILIDAD DEL MÉTODO ÁGIL

- Desarrollo de productos, donde una compañía de software está desarrollando un producto de pequeño o mediano tamaño para la venta
- El desarrollo del sistema a medida dentro de una organización, donde hay un claro compromiso por parte del cliente para participar en el proceso de desarrollo y donde no hay muchas reglas y regulaciones externas que afectarán el software.
- Debido a su enfoque en pequeños equipos, bien integrados, hay problemas en la ampliación de los métodos ágiles a grandes sistemas

CUESTIONES TÉCNICAS, HUMANAS Y ORGANIZACIONALES

La mayoría de los proyectos incluyen elementos del plan guiado y procesos ágil. La decisión sobre el equilibrio depende de:

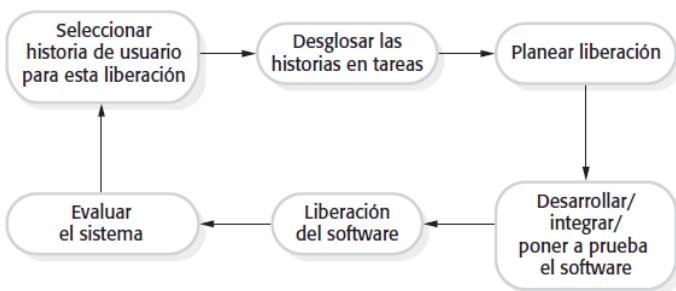
- ¿Es importante contar con una especificación y diseño muy detallado antes de pasar a la implementación?
Es una estrategia de entrega incremental, donde se entrega el software a los clientes y se obtiene una rápida retroalimentación de ellos, ¿realista?
- ¿Qué tan grande es el sistema que se está desarrollando? ¿Qué tipo de sistema se está desarrollando?
Los enfoques del plan guiado pueden ser necesarios para los sistemas que requieren una gran cantidad de análisis antes de la aplicación
- ¿Cuál es la expectativa de vida del sistema?
Los sistemas de larga vida tal vez requieran más documentación de diseño para comunicar las intenciones originales de los desarrolladores del sistema al equipo de soporte.
- ¿Qué tecnologías están disponibles para apoyar el desarrollo del sistema?
Los métodos ágiles se basan en buenas herramientas para realizar un seguimiento de la evolución de un diseño
- ¿Cómo se organiza el equipo de desarrollo?
Si el equipo de desarrollo se distribuye o si parte del desarrollo se subcontrata, entonces puede que tenga que desarrollar los documentos de diseño para comunicarse a través de los equipos de desarrollo.
- ¿Hay cuestiones culturales o de organización que puedan afectar al desarrollo del sistema?
Las organizaciones tradicionales de ingeniería tienen una cultura basada en el plan de desarrollo, ya que esta es la norma en la ingeniería.
- ¿Qué tan buenos son los diseñadores y programadores del equipo de desarrollo?
A veces se argumenta que los métodos ágiles requieren niveles más altos de capacitación que los enfoques basados en planes, en el que los programadores simplemente traducen un diseño detallado en código.
- ¿El sistema está sujeto a regulación externa?
Si un sistema tiene que ser aprobado por un regulador externo (por ejemplo, que el FAA aprueba el software, es crítico para el funcionamiento de una aeronave) entonces usted probablemente tendrá que producir detallada documentación como parte de la justificación de la seguridad del sistema.

EXTREME PROGRAMMING

La programación extrema (XP) es quizás el método ágil mejor conocido y más ampliamente usado.

Extreme programming 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 figura 3.3 ilustra el proceso XP para producir un incremento del sistema por desarrollar.



ACTIVIDADES XP

- Escribir código
- Testear el sistema
- Escuchar a los clientes y los usuarios
- Diseñar para reducir el acoplamiento

VALORES XP

- La comunicación es esencial

Con objetivo de romper las clásicas barreras entre negocio y desarrollo, XP promueve que los requisitos sean comunicados y trabajados con el equipo de desarrollo mano a mano y no a través de documentación (aspecto que el Agile Manifesto recoge como uno de sus cuatro valores).

En lo referente a la comunicación establece: "Todos son parte del equipo y nos comunicamos cara a cara todos los días. Trabajamos juntos en todo, desde los requerimientos hasta la programación. En equipo crearemos la mejor solución al problema."

En los métodos tradicionales de desarrollo de software, la comunicación de los requerimientos a los desarrolladores se realiza a través de la documentación, por ejemplo las Especificaciones de Diseño en el Rational Unified Process (RUP).

XP rompe con este esquema, la comunicación se realiza por medio de transferencia de conocimientos en reuniones frecuentes cara a cara entre usuarios y desarrolladores, lo que le da a ambos una visión compartida del sistema.

Por ello, XP favorece diseños simples, colaboración de usuarios con programadores, comunicación verbal frecuente, retroalimentación y construcción rápida del software.

- Simplicidad

La simplicidad implica que: "Desarrollaremos lo que sea solicitado y necesario, pero no más que eso. De esa forma, se maximiza el valor de la inversión realizada. Nos dirigiremos a nuestro objetivo a pasos simples y pequeños, mitigando las fallas a medida que ocurran. Crearemos algo de lo cual podamos sentirnos orgullos y que pueda mantenerse en el largo plazo a costos razonables."

En XP se comienza desarrollando las soluciones más sencillas necesarias para solucionar los problemas (requerimientos) que se están viendo en ese momento, añadiendo funcionalidad extra más tarde, en la medida en que se obtiene más información de los requerimientos. La diferencia respecto a esquemas tradicionales es que se enfoca en las necesidades de hoy en lugar de las necesidades de mañana, la semana que viene o el mes que viene.

La ventaja es que no se invierte esfuerzo en futuros requerimientos que podrían cambiar o que no se vayan a necesitar.

Asimismo, un diseño y programación simple mejora la calidad de las comunicaciones, pues es más fácil de implementar y entender por todos en el equipo.

- Aprender del feedback

Según extremeprogramming el valor de la retroalimentación establece: "Nos tomaremos seriamente los compromisos con el usuario establecidos en todas las iteraciones, entregando software en funcionamiento en cada una. Mostraremos al usuario nuestro software frecuentemente y de forma temprana, escuchando cuidadosamente sus observaciones y realizando los cambios que sean necesarios. Adaptaremos nuestros procesos al proyecto y no al contrario".

Según Wikipedia, La retroalimentación actúa en tres dimensiones:

- **Retroalimentación del sistema:** Por medio de la ejecución de pruebas unitarias y de integración, los programadores reciben retroalimentación directa del estado del sistema.
- **Retroalimentación del cliente (usuario):** Las pruebas de aceptación, son diseñadas conjuntamente por el cliente y los analistas de pruebas, obteniendo en conjunto retroalimentación del estado actual del sistema. Esta revisión puede hacerse cada 2 o 3 semanas, permitiendo así que el cliente sea quien guíe el desarrollo del software.
- **Retroalimentación del equipo:** Cuando el cliente trae nuevos requerimientos, el equipo puede directamente proporcionar la estimación del tiempo que tomará implementarlos.

Bajo este esquema, las fallas de sistema se pueden comunicar fácilmente, pues existen pruebas unitarias que demuestran que el sistema fallará si es puesto en producción. Asimismo, un cliente puede probar el sistema periódicamente, contrastando el funcionamiento con sus requerimientos funcionales o “Historias de usuario”.

→ **Tener coraje**

Establece: “Diremos la verdad en nuestros avances y estimados, no documentaremos excusas para el fracaso, pues planificamos para tener éxito. No tendremos miedo a nada pues sabemos que nadie trabaja solo. Nos adaptaremos a los cambios cuando sea que estos ocurran.”

Algunas prácticas del coraje son:

- Diseñar y programar para hoy y no para mañana, evitando así hacer énfasis en el diseño en detrimento de todo lo demás.
- Refactorizar el código siempre que sea necesario (No tener reservas al respecto).
- Inspeccionar constantemente el código y modificarlo (refactorizar), de tal manera que futuros cambios se puedan implementar más fácilmente (desarrollar rápido para atender las necesidades de hoy, pero refactorizar después para facilitar el mantenimiento).
- Desechar componentes o piezas de código cuando sea necesario, sin preocuparse del tiempo invertido (y perdido) en su creación (Es mejor desechar algo que no es útil en lugar de tratar de repararlo).
- Ser persistente en la resolución de problemas.

→ **Respetar al team y al proyecto**

El valor del respeto en XP establece: “Todos en el equipo dan y reciben el respeto que merecen como integrantes del equipo y los aportes de cada integrante son valorados valorados por todos. Todos contribuyen, así sea simplemente con entusiasmo. Los desarrolladores respetan la experticia de los clientes y viceversa. La Gerencia respeta el derecho del equipo de asumir responsabilidad y tener autoridad sobre su trabajo”.

Respeto es tanto por el trabajo de los demás como por el trabajo de uno mismo, por ejemplo, los desarrolladores nunca deben subir cambios que impidan la compilación de la versión, que hagan fallar pruebas unitarias ya realizadas o que de alguna otra forma retrasen el trabajo de sus pares, esto significa tener respeto por el trabajo (y el tiempo) de los demás.

Asimismo, los desarrolladores respetan su propio trabajo por medio de su compromiso con una alta calidad y buscando el mejor diseño para la solución por medio de la refactorización constante.

En cuanto al trabajo en equipo, nadie debe sentirse poco apreciado o ignorado, todos deben colaborar en esto, tratando con respeto a sus compañeros y mostrando respeto por sus opiniones, esto asegura altos niveles de motivación y lealtad hacia el proyecto.

PRINCIPIOS XP

→ **Feedback rápido**

La retroalimentación rápida consiste en obtener la retroalimentación, comprenderla y volver a poner el aprendizaje en el sistema lo más rápido posible.

Los desarrolladores diseñan, implementan y prueban el sistema y utilizan esa retroalimentación en segundos o minutos en lugar de días, semanas o meses.

Los clientes revisan el sistema para comprobar cuál es la mejor forma de contribuir y dan su opinión en días o semanas en lugar de meses o años.

→ **Construir con simplicidad (DTSTTCPW, KISS, YAGNI)**

Asumir la simplicidad es tratar cada problema como si pudiera resolverse con sencillez.

Tradicionalmente, se le dice que planifique para el futuro, que diseñe para su reutilización. El resultado de este enfoque puede convertirse en 'lo que el cliente requiere hoy no se cumple y lo que finalmente se entrega puede ser obsoleto y difícil de cambiar'.

'Asumir simplicidad' significa 'hacer un buen trabajo resolviendo el trabajo de hoy hoy y confiar en su capacidad para agregar complejidad en el futuro donde la necesite'. En Extreme Programming, se le dice que haga un buen trabajo (pruebas, refactorización y comunicación) enfocándose en lo que es importante hoy.

- Con buenas pruebas unitarias, puede refactorizar fácilmente su código para realizar pruebas adicionales.
- Sigue a YAGNI (No lo vas a necesitar).
- Siga el principio DRY (Don't Repeat Yourself). Por ejemplo,
 - No tenga varias copias de código idéntico (o muy similar).
 - No tenga copias redundantes de información.
 - Sin desperdicio de tiempo y recursos en lo que puede no ser necesario.

→ Cambio incremental

En cualquier situación, los grandes cambios hechos de una vez simplemente no funcionan. Cualquier problema se resuelve con una serie de cambios mínimos que marcan la diferencia.

En Programación extrema, el Cambio incremental se aplica de muchas formas.

- El diseño cambia poco a poco.
- El plan cambia poco a poco.
- El equipo cambia poco a poco.

Incluso la adopción de la Programación Extrema debe tomarse en pequeños pasos.

→ Aceptar y valorar el cambio como algo positivo

La mejor estrategia es la que conserva la mayor cantidad de opciones y al mismo tiempo resuelve su problema más urgente.

→ Trabajo de calidad

A todo el mundo le gusta hacer un buen trabajo. Intentan producir la calidad de la que están orgullosos. El equipo

- Funciona bien
- Disfruta el trabajo
- Se siente bien al producir un producto de valor.

PRÁCTICAS XP

Las prácticas de Programación Extrema se pueden agrupar en cuatro áreas:

- Retroalimentación rápida y fina -
 - Pruebas
 - Cliente in situ
 - Programación en pareja
- Proceso continuo -
 - Integración continua
 - Refactorización
 - Lanzamientos breves
- Comprensión compartida -
 - El juego de la planificación
 - Diseño simple
 - Metáfora
 - Propiedad colectiva
 - Estándares de codificación
- Bienestar del desarrollador –
 - Semana de cuarenta horas

Principio o práctica	Descripción
Planeación incremental	Los requerimientos se registran en tarjetas de historia (<i>story cards</i>) 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. Vea las figuras 3.5 y 3.6.
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.
Diseño simple	Se realiza un diseño suficiente para cubrir sólo aquellos requerimientos actuales.

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.
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.
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.
Propiedad colectiva	Los desarrolladores en pares laboran en todas las áreas del sistema, de manera que no se desarrollan islas de experiencia, ya que todos los desarrolladores se responsabilizan por todo el código. Cualquiera puede cambiar cualquier función.
Integración continua	Tan pronto como esté completa una tarea, se integra en todo el sistema. Después de tal integración, deben aprobarse todas las pruebas de unidad en el sistema.
Ritmo sustentable	Grandes cantidades de tiempo extra no se consideran aceptables, pues el efecto neto de este tiempo libre con frecuencia es reducir la calidad del código y la productividad de término medio.
Cliente en sitio	Un representante del usuario final del sistema (el cliente) tiene que disponer de tiempo completo para formar parte del equipo XP. En un proceso de programación extrema, el cliente es miembro del equipo de desarrollo y responsable de llevar los requerimientos del sistema al grupo para su implementación.

USER STORIES EN XP

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 ágiles incluyen una o dos frases escritas y, lo que es más importante, una serie de conversaciones sobre la funcionalidad deseada.

¿QUÉ ES UNA HISTORIA 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. Ellos típicamente siguen una simple plantilla:

Como un <tipo de usuario>, quiero <algún objetivo> para que <alguna razón>

Las historias de los usuarios a menudo se escriben en tarjetas de índice o notas adhesivas, guardadas en una caja de zapatos, y dispuestas en paredes o mesas para facilitar la planificación y el debate.

Cambian fuertemente el foco de atención de escribir sobre características a discutirlas. Estas discusiones son más importantes que cualquier texto que se escriba.

Ejemplos

Uno de los beneficios de las historias de usuarios ágiles es que pueden ser escritas a diferentes niveles de detalle. Podemos escribir una historia de usuario para cubrir grandes cantidades de funcionalidad. Estos grandes usuarios las historias son generalmente conocidas como épicas. Aquí hay un ejemplo de una historia épica de un usuario ágil de un escritorio producto de respaldo:

Como usuario, puedo hacer una copia de seguridad de todo mi disco duro.

Debido a que una epopeya es generalmente demasiado grande para que un equipo ágil la complete en una sola iteración, es se dividen en múltiples historias de usuarios más pequeños antes de que se trabaje en ello. La epopeya anterior podría dividirse en docenas (o posiblemente cientos), incluyendo estos dos:

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.

¿CÓMO SE AÑADEN LOS DETALLES A LAS HISTORIAS DE LOS USUARIOS?

Los detalles pueden ser añadidos a las historias de los usuarios de dos maneras:

- Dividiendo una historia de usuario en varias historias de usuario más pequeñas.
- Añadiendo "condiciones de satisfacción/exito".

Cuando una historia de usuario relativamente grande se divide en múltiples historias de usuario ágiles, más pequeñas, es natural asumiendo que se han añadido detalles. Después de todo, se ha escrito más.

Las condiciones de satisfacción/exito son simplemente una prueba de aceptación de alto nivel que será verdadero después de que la historia del usuario ágil esté completa.

¿QUIÉN ESCRIBE LAS HISTORIAS DE LOS USUARIOS?

Cualquiera puede escribir historias de usuarios. Es responsabilidad del propietario del producto asegurarse de que un de historias de usuarios ágiles existe, pero eso no significa que el propietario del producto sea el que las escribe.

En el curso de un buen proyecto ágil, deberías esperar tener ejemplos de historias de usuarios escrito por cada miembro del equipo.

Además, ten en cuenta que quien escribe una historia de usuario es mucho menos importante que quien está involucrado en las discusiones de ello.

¿CUÁNDO SE ESCRIBEN LAS HISTORIAS DE USUARIOS?

Las historias de usuarios se escriben a lo largo de todo el proyecto ágil.

Usualmente un taller de escritura de historias se lleva a cabo cerca del comienzo del proyecto ágil. Todos los miembros del equipo participan con el objetivo de crear un producto que completamente describe la funcionalidad que se añadirá en el transcurso del proyecto o de tres a seis meses ciclo de liberación dentro de ella.

Algunas de estas ágiles historias de usuarios serán sin duda épicas: las épicas se descompondrán más tarde en historias más pequeñas que encajen más fácilmente en un solo iteración.

Además, se pueden escribir nuevas historias y añadirlas al producto en cualquier momento y por nadie.

LAS HISTORIAS DE USUARIOS NO SON CASOS DE USO

- Ambos ofrecen valor comercial
- Ámbito: las historias son más pequeñas, deberían encajar en una iteración
- Nivel de completitud
- Longevidad: los casos de uso son de larga duración, las historias son para una iteración
- Caso de uso propenso a incluir detalles de la interfaz de usuario

LAS HISTORIAS DE USUARIOS NO SON DECLARACIONES DE REQUISITOS

Lista de declaraciones de "el sistema..." .

- ¿Tedioso, propenso a errores, que requiere tiempo, por adelantado, completo?
- Centrado en el sistema y no en los objetivos del usuario

Ejemplo:

3.4) El producto tendrá un motor de gasolina.

3.5) El producto tendrá cuatro ruedas.

3.5.1) El producto tendrá un neumático de goma montado en cada rueda.

3.6) El producto tendrá un volante.

3.7) El producto tendrá un cuerpo de acero.

DISEÑO EN XP

El sistema debe ser diseñado tan simple como sea posible

El diseño correcto

- Hace pasar todos los tests

- No tiene lógica duplicada
- Comunica las intenciones del desarrollador
- Tiene la menor cantidad de unidades (clases, métodos, funciones, etc)

Opuesto a “Implementa para hoy, diseña para mañana” porque “El futuro es incierto”

VENTAJAS DEL DISEÑO SIMPLE

- No se pierde tiempo en funcionalidad superflua
- Más fácil de entender
- Más fácil de “refactorizar” y de crear “propiedad colectiva” del código
- Ayuda a mantener a los programadores en Schedule

REFACTORING EN XP

XP Y EL CAMBIO

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

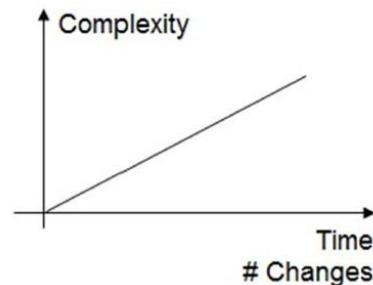
XP, sin embargo, descarta este principio ya que sostiene que los cambios no se pueden prever de forma fiable

En su lugar, propone la mejora constante de código (Refactoring/Reconstrucción).

REFACTORING

El refactoring está motivado por los smells

- Design/Code smells
- Rotting software



DESIGN SMELLS

Los cambios constantes degradan el diseño y el código. Signos de un mal diseño o un diseño degradado

- **Rigidez:** es el nivel de dificultad que tiene el software para ser modificado incluso en circunstancias o elementos relativamente sencillos; modificaciones que pueden estar estimadas para uno o dos días pueden volverse monstruosas con los cuales lidiar durante semanas.
- **Fragilidad:** es la tendencia del software a quebrarse incluso en áreas conceptualmente no relacionadas. Como pueden imaginarse es bastante problemático tratar de mejorar una aplicación si cualquier cambio produce resultados inesperados.
- **Inmovilidad:** es la incapacidad del software para ser reutilizado. Imaginemos que tenemos un proyecto en el que nos gustaría reutilizar un componente que se encuentra en otro proyecto, o incluso en el proyecto actual, pero resulta que la dependencia que tiene este componente es tan grande que tratar de abstraerlo para ser reusable representa un coste mayor al de reconstruir el componente.
- **Viscosidad:** existen varias formas de modificar una aplicación, algunas preservan el diseño, otras lo arruinan. Hace referencia a agregar funcionalidades a un código que no se encuentra ordenado, el desarrollador sumará sus líneas de código siguiendo ese caos.

CODE SMELL

Característica de un programa que indica un posible problema más profundo. Análisis subjetivo

Ejemplos:

- Métodos muy largos
- Métodos muy similares

Ciertas estructuras en el código que indican la violación de principios fundamentales de diseño e impactan de manera negativa la calidad del diseño

No es un bug

El código funciona correctamente

Indican debilidad en el diseño que pueden

- Hacer el desarrollo más lento
- Aumentar la posibilidad de la introducción de bugs en el desarrollo

A nivel de aplicación:

- **Código duplicado:** código idéntico o muy similar existe en más de un lugar.
- **Complejidad artificial:** uso forzado de un diseño demasiado complicado patrones donde un diseño más simple sería suficiente.
- **Cirugía de escopeta:** un solo cambio debe ser aplicado a múltiples clases al mismo tiempo.

A nivel de clases:

- **Clase grande:** una clase que ha crecido demasiado. Dios se opone.
- **Feature envy:** una clase que usa métodos de otra clase en exceso.
- **Intimidad inapropiada:** una clase que depende de los detalles de implementación de otra clase.
- **Solicitud denegada:** una clase que anula un método de una clase base de tal manera que el contrato de la clase base no es respetado por la clase derivada. Liskov principio de sustitución.
- **Clase perezosa:** una clase que hace muy poco.
- **Uso excesivo de los literales:** estos deben ser codificados como constantes de nombre, para mejorar y para evitar errores de programación. Además, los literales pueden y deben ser externalizado en archivos/scripts de recursos, u otros almacenes de datos como bases de datos cuando sea posible, para facilitar la localización de los programas informáticos si están destinados a ser desplegados en diferentes regiones.
- **Complejidad ciclomática:** demasiadas ramas o bucles; esto puede indicar una función necesita ser dividido en funciones más pequeñas, o que tiene potencial para simplificación.
- **Downcasting:** un tipo de molde que rompe el modelo de abstracción; la abstracción tiene que ser refactorizada o eliminados.
- **Clase huérfana variable o constante:** una clase que típicamente tiene una colección de constantes que pertenecen a otro lugar donde esas constantes deberían ser propiedad de uno de las otras clases de miembros.
- **Grupo de datos:** Ocurre cuando un grupo de variables se pasan juntas en varias partes del programa. En general, esto sugiere que sería más apropiado para agrupar formalmente las diferentes variables en un solo objeto, y en su lugar pasar sólo este objeto.

A nivel de métodos:

- **Demasiados parámetros:** una larga lista de parámetros es difícil de leer, y hace que llamar y probar la función complicada. Puede indicar que el propósito de la función está mal concebido y que el código debe ser refactorizado para que se asigne la responsabilidad de una manera más limpia.
- **Método largo:** un método, función o procedimiento que ha crecido demasiado.
- **Identificadores excesivamente largos:** en particular, el uso de convenciones de denominación para proporcionar desambiguación que debería estar implícita en la arquitectura del software.
- **Identificadores excesivamente cortos:** el nombre de una variable debe reflejar su función a menos que la función sea obvia.
- **Excesivo retorno de datos:** una función o método que devuelve más de lo que cada uno de las necesidades de sus llamantes.
- **Una línea de código excesivamente larga (o línea de Dios):** Una línea de código que es tan larga, haciendo que el código sea difícil de leer, entender, depurar, refactorizar o incluso identificar posibilidades de reutilización del software.

TECHNICAL DEBT

Los code smells son indicadores de factores que contribuyen a la técnica deuda

La deuda técnica: un concepto en el desarrollo de software que refleja el costo implícito de retrabajo causado por elegir una solución fácil en lugar de usar una mejor solución que había tomado más tiempo en desarrollarse

Se puede comparar con las deudas financieras, si la deuda técnica no se paga acumula intereses, haciendo cambios futuros más difíciles

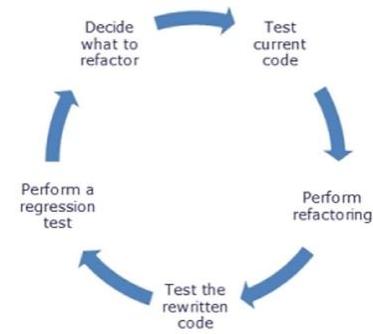
Refactorización

Los code smells son un indicador de que se necesita refactorización. La refactorización es una forma de pagar la deuda técnica.

Beneficios de la refactorización:

- Mantenimiento. Es más fácil arreglar los errores porque el código fuente es fácil de y la intención de su autor es fácil de entender.
- Extensibilidad. Es más fácil ampliar las capacidades de la aplicación si utiliza patrones de diseño reconocibles, y proporciona cierta flexibilidad cuando ninguno antes puede haber existido.

Un arnés de prueba automatizado con suficiente cobertura es obligatorio para poder hacer una buena refactorización



El refactoring se hace mediante ciclos iterativos de pequeñas transformaciones del programa y pruebas intercaladas

Técnicas de refactorización

Técnicas que permiten una mayor abstracción

- Encapsular el campo - forzar el código para acceder al campo con los métodos getter y setter
- Generalizar el tipo - crear tipos más generales para permitir más compartir el código
- Reemplazar el código de verificación de tipo con el estado/estrategia
- Sustituir el condicional por el polimorfismo

Técnicas para separar el código en piezas más lógicas

- La componentización descomponer el código en unidades semánticas reutilizables que presentan interfaces claras, bien definidas y fáciles de usar.
- Extraer la clase mueve parte del código de una clase existente a una nueva clase.
- Método de extracción, para convertir parte de un método más grande en un nuevo método. Al descomponer el código en piezas más pequeñas, es más fácil comprenderlo. Esto también es aplicable a las funciones.

Técnicas para mejorar los nombres y la ubicación del código

- Mover el método o mover el campo - mover a una clase más apropiada o archivo de origen
- Método de renombramiento o campo de renombramiento - cambiar el nombre en un nuevo uno que revela mejor su propósito
- Pull up - en la programación orientada a objetos (OOP), pasar a una superclase
- Push down - en OOP, pasar a una subclase

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 cuando se terminó la tarea
- Los tests son automatizados. 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

Pruebas antes del código => aclara los requerimientos

Pruebas == programas != texto => se pueden ejecutar de forma automática

Pruebas anteriores y nuevas =>

- 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. Por consiguiente, todo nuevo código es validado para asegurarse de que es lo que necesita el cliente.

No obstante, las personas que adoptan el papel de clientes tienen tiempo disponible limitado y por lo tanto no pueden trabajar a tiempo completo con el equipo de desarrollo. Pueden pensar que la presentación de los requerimientos es suficiente contribución y por tanto pueden ser reacios a involucrarse en el proceso de prueba.

LA AUTOMATIZACIÓN DE PRUEBAS

La 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.

Un marco de prueba automatizada (por ejemplo, junit) es un sistema que hace que sea fácil de escribir pruebas ejecutables y presentar un conjunto de pruebas para su ejecución

Como se automatiza las pruebas, siempre hay un conjunto de pruebas que puede ser rápida y fácilmente ejecutado

Siempre que se agrega alguna funcionalidad al sistema, las pruebas se pueden correr y los problemas que ha introducido el nuevo código puede ser atrapadas inmediatamente

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.

Algunas pruebas pueden ser muy difíciles de escribir de forma incremental. Por ejemplo, en una interfaz de usuario compleja, es a menudo difícil de escribir pruebas unitarias para el código que implementa la 'lógica de visualización "y flujo de trabajo entre las pantallas

Es difícil juzgar la integridad de un conjunto de pruebas. Aunque usted puede tener un montón de pruebas del sistema, la prueba de conjunto puede no proporcionar una cobertura completa.

PROGRAMACIÓN EN PAREJAS EN XP

En XP, los programadores trabajan en parejas,

- Sentados juntos a desarrollar código.
- Los pares se crean dinámicamente

Ventajas

- Ayuda a desarrollar la propiedad común de código
- Difunde el conocimiento a través del equipo.
 - Sirve como un proceso de revisión informal ya que cada línea de código es visto por más de 1 persona.
- Alienta a la reconstrucción

Productividad programación en pareja similar a dos personas trabajando de forma independiente

Puntos clave

- Los métodos ágiles son métodos incrementales de desarrollo que se centran en el desarrollo rápido, versiones frecuentes del software, reduciendo los gastos generales del proceso.
- Implican al cliente directamente en el proceso de desarrollo.

La decisión sobre si se debe utilizar un enfoque ágil o un enfoque dirigido por un plan para el desarrollo debe depender del tipo de software que está desarrollado, las capacidades del equipo de desarrollo y la cultura de la empresa que desarrolla el sistema.

La programación extrema es un método ágil bien conocido que integra una serie de buenas prácticas de programación tales como versiones frecuentes del software, el software de mejora continua y participación del cliente en el equipo de desarrollo.

SCRUM

GESTIÓN DE PROYECTOS ÁGILES

La principal responsabilidad de los directores de proyectos de software es la gestión del proyecto para que el software se entregue a tiempo y dentro del presupuesto previsto para el proyecto

El enfoque estándar para la gestión de proyectos es el direccionado por plan. Los gerentes elaboran un plan para el proyecto mostrando qué debería ser entregado, cuándo debería ser entregado y quién trabajara en el desarrollo del proyecto

La gestión de proyectos ágil requiere un enfoque diferente, que está adaptado para desarrollo incremental y las fortalezas particulares de los métodos ágiles.

MANIFIESTO ÁGIL

- Valores y Principios
- Énfasis en colaboración y comunicación, software funcionando, etc.
- En contraste con el PMBOK
- No provee pasos concretos

Se busca una metodología o marco de trabajo ágil: generalmente scrum más prácticas de XP u otras metodologías

Scrum es un marco de trabajo ágil iterativo e incremental para gestionar el desarrollo de un producto. Define una estrategia flexible donde el equipo de desarrollo trabaja como una unidad para alcanzar una meta común.

Un marco de trabajo por el cual

- Se puede acometer problemas complejos adaptativos
- Entregar productos del máximo valor posible productiva y creativamente
- Ligero
- Fácil de entender
- Extremadamente difícil de llegar a dominar

No es un proceso o una técnica para construir productos. Es un marco de trabajo dentro del cual se pueden emplear varias técnicas y procesos

Scrum muestra la eficacia relativa de las prácticas de gestión de producto y las prácticas de desarrollo, de modo que podamos mejorar.

Consiste de equipo, Roles, Eventos y Artefactos

Teoría del control de proceso empírica o empirismo

- Tomar decisiones

- basados en hechos reales y no especulaciones
- basados en la experiencia

→ Iterativo e incremental



→ Pilares del control empírico:

- Transparencia
- Inspección (en busca de variaciones)
- Adaptabilidad

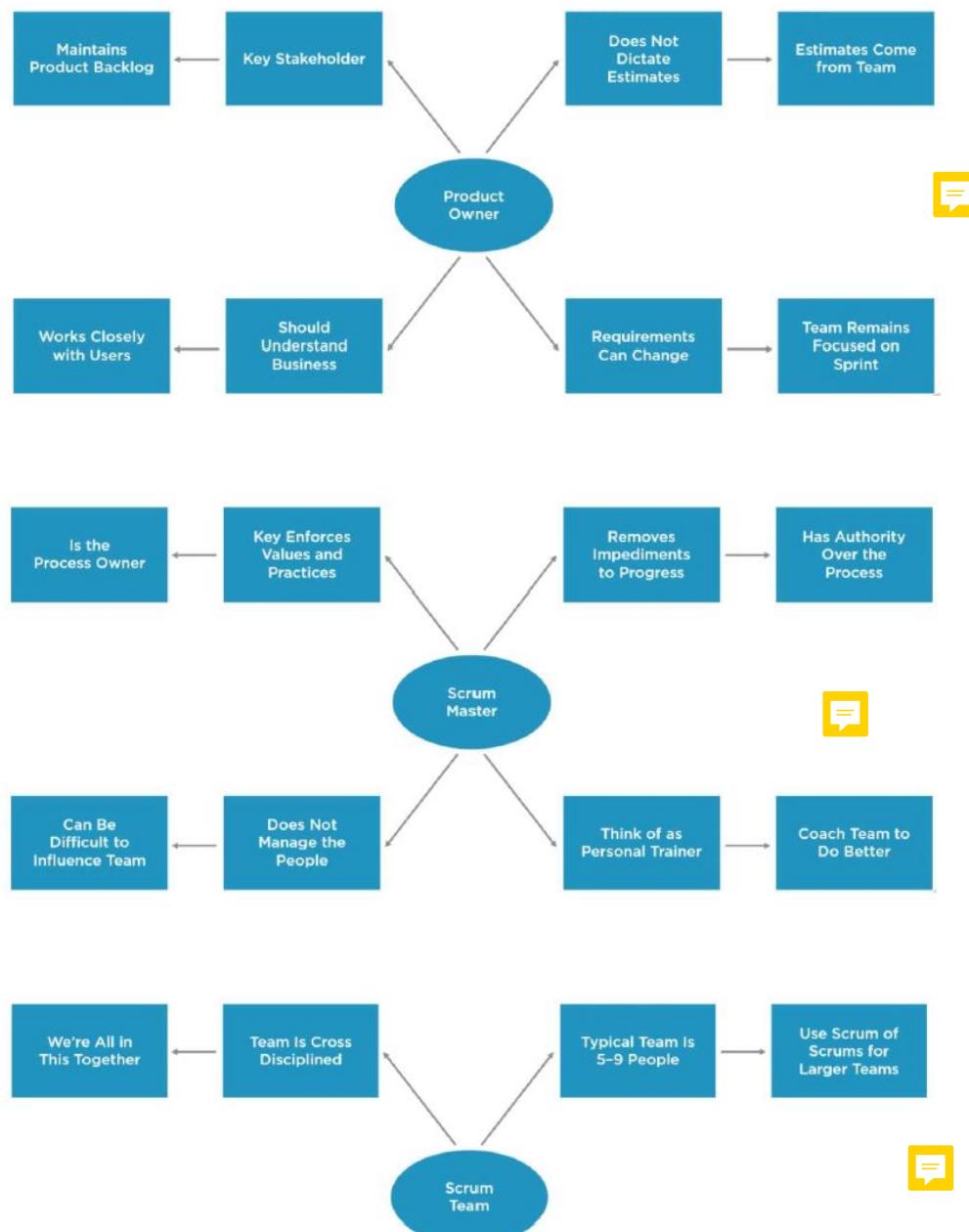
Ciclos de feedback “inspect and adapt”

- El tiempo se divide en ciclos cortos conocidos como sprints
- El producto está en estado “potencialmente entregable” en todo momento
- Al final de cada ciclo se muestran los resultados y se planea el próximo

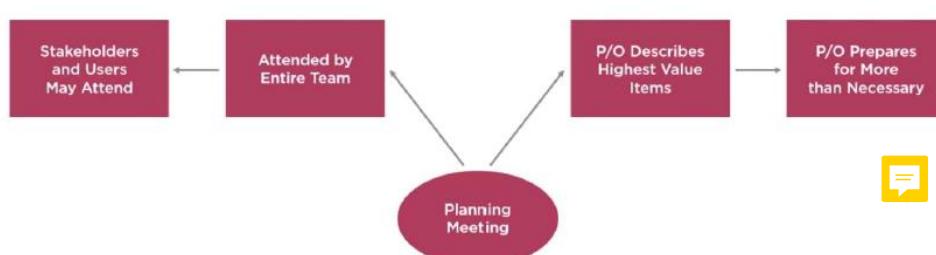
Eventos formales:

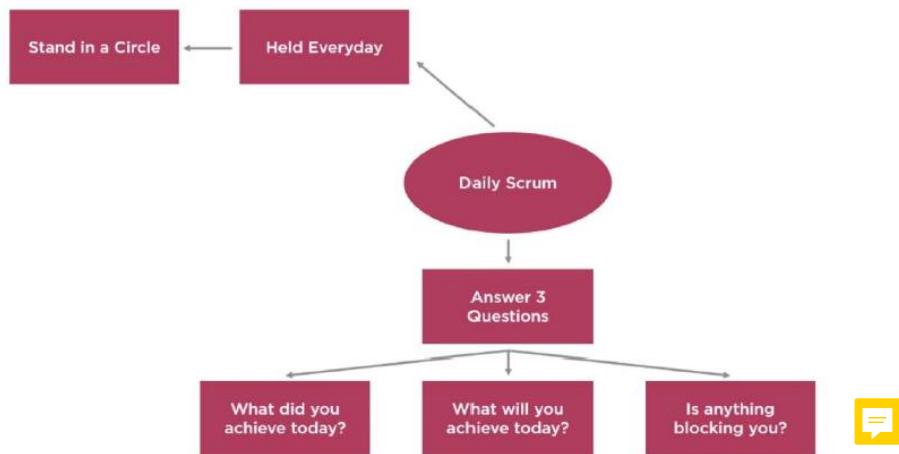
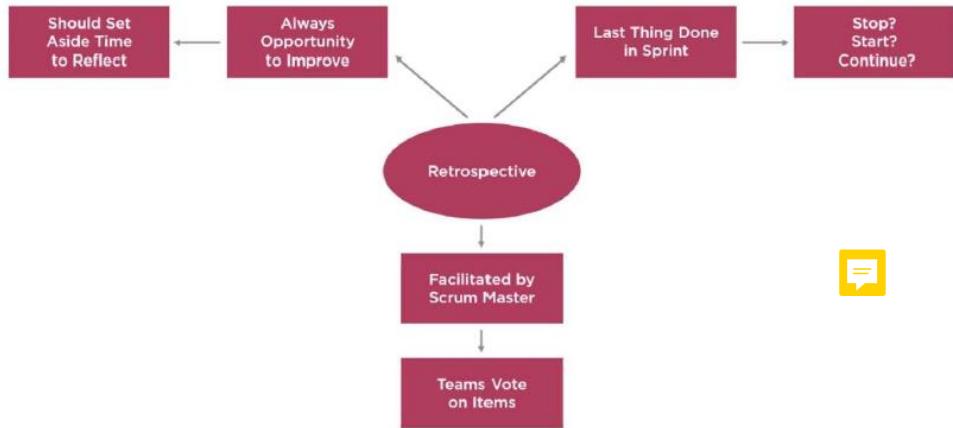
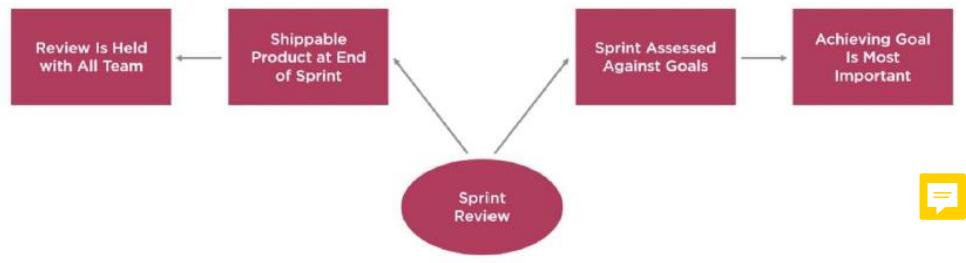
- Reunión de Planificación del Sprint (Sprint Planning Meeting)
- Scrum Diario (Daily Scrum)
- Revisión del Sprint (Sprint Review)
- Retrospectiva del Sprint (Sprint Retrospective)

ROLES

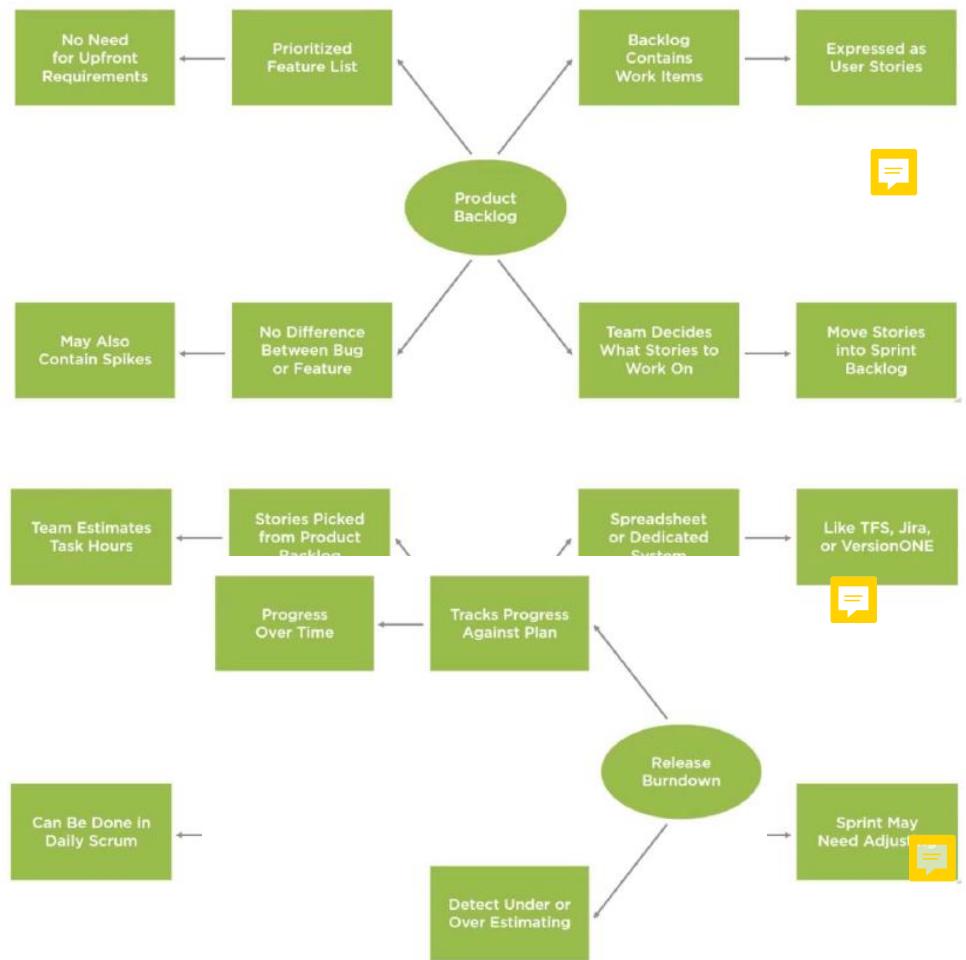


EVENTOS

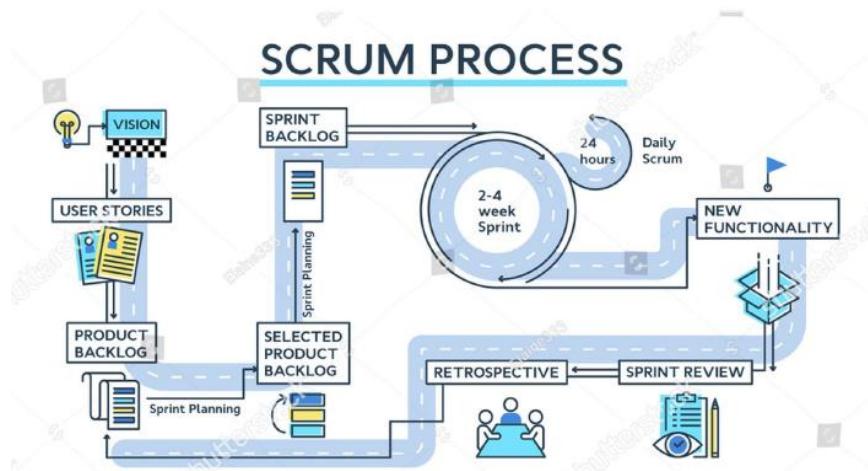




ARTIFACTS



SCRUM PROCESS



SCRUM TERMINOLOGY

Scrum term	Definition
Development team	Un grupo auto-organizado de desarrolladores de software, que no debe ser más que 7 personas. Son responsables de desarrollar el software y otros documentos esenciales del proyecto.
Potentially shippable product increment	El incremento de software que se entrega de un sprint. La idea es que este debería ser "potencialmente transportable", lo que significa que está en un estado final y no se necesita ningún trabajo adicional, como pruebas, para incorporarlo en la producto. En la práctica, esto no siempre es posible.
Product backlog	Esta es una 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.
Product owner	Un 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.
Srum	Una reunión diaria del equipo Scrum que revisa el progreso y prioriza el trabajo a realizar ese día. Idealmente, esto debería ser un corto cara a cara reunión que incluye a todo el equipo.
ScrumMaster	El ScrumMaster es responsable de asegurar que el proceso de Scrum sea siguió y guía al equipo en el uso efectivo de Scrum. Él o ella es responsable de interactuar con el resto de la compañía y de asegurar que el equipo de Scrum no se desvíe por interferencias externas. El Scrum los desarrolladores son inflexibles en que el ScrumMaster no debe ser pensado como director de proyecto. Sin embargo, a otros no siempre les resulta fácil ver la diferencia.
Potentially shippable product increment	El incremento de software que se entrega de un sprint. La idea es que este debería ser "potencialmente transportable", lo que significa que está en un estado final y no se necesita ningún trabajo adicional, como pruebas, para incorporarlo en la producto. En la práctica, esto no siempre es posible.
Sprint	La iteración del desarrollo. Los sprints suelen durar de 2 a 4 semanas.
Velocity	Una medida de la cantidad de esfuerzo de atraso de producto que un equipo puede cubrir en un solo sprint. Entender la velocidad de un equipo les ayuda a estimar lo que puede cubrirse en un sprint y proporciona una base para medir la mejora de la actuación.

EL CICLO DE 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.

- Estimaciones
- Punto de Historia
- Poker planning
- Velocidad

→ 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. Basados en:

- Prioridades
- Esfuerzo estimado
- Velocidad del team

Una vez que éstos están de acuerdo, los equipos se organizan 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 presentó a las partes interesadas. El siguiente ciclo de 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 DE 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.

UNIDAD 5

AUTOMATIZACIÓN DE PRUEBAS

INTRODUCCIÓN

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

En la práctica, el proceso de prueba por lo general requiere una combinación de pruebas manuales y automatizadas.

- En las primeras **pruebas manuales**, un examinador opera el programa con algunos datos de prueba y compara los resultados con sus expectativas. Anota y reporta las discrepancias con los desarrolladores del programa.
- En las **pruebas automatizadas**, éstas se codifican en un programa que opera cada vez que se prueba el sistema en desarrollo. Comúnmente esto es más rápido que las pruebas manuales, sobre todo cuando incluye pruebas de regresión, es decir, aquellas que implican volver a correr pruebas anteriores para comprobar que los cambios al programa no introdujeron nuevos bugs.

El uso de pruebas automatizadas aumentó de manera considerable durante los últimos años. Sin embargo, las pruebas nunca pueden ser automatizadas por completo, ya que esta clase de pruebas sólo comprueban que un programa haga lo que supone que tiene que hacer.

BUILD QUALITY IN

“No depender en inspecciones masivas para alcanzar la calidad, mejorar el proceso y construir con la calidad incluida en el producto desde el principio”

Testing es una actividad que involucra a todo el equipo y debería ser continua desde el principio del proyecto

“Building quality in” para testing significa escribir tests automatizados a diferentes niveles:

- Unidad
- Componente
- Aceptación

Ejecutarlos como parte del deployment pipeline el cual es iniciado cada vez que se cambia el código, la configuración, o el entorno dónde se ejecuta

PROYECTO IDEAL

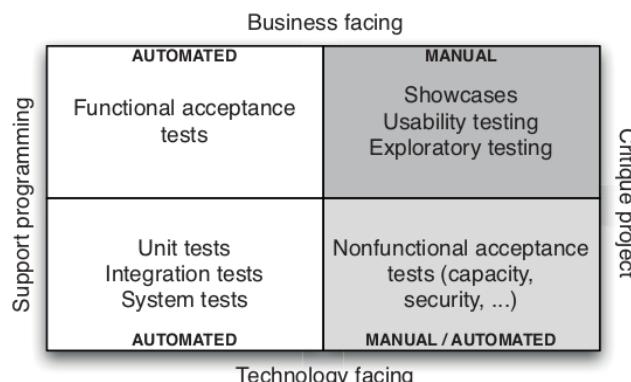
Colaboración entre testers, developers, usuarios para escribir tests automatizados desde el comienzo del proyecto

Los tests se escriben antes que los desarrolladores comiencen a trabajar en la feature que los tests testean

El conjunto de tests forman una especificación ejecutable del comportamiento del sistema. Cuando pasan demuestran que la funcionalidad requerida por el cliente ha sido implementada completamente y correctamente. El conjunto de tests es ejecutado por el sistema de CI, por lo tanto, también sirve como tests de regresión

También se incluyen tests automatizados para los aspectos “no funcionales” del sistema

TIPOS DE TESTS



SOPORTE DEL PROCESO DE DESARROLLO, DE CARA AL NEGOCIO

Los tests funcionales de aceptación aseguran que el criterio de aceptación de una story se cumple. Deberían ser escritos y automatizados antes que el desarrollo de la story comience.

Son críticos en un entorno ágil responden a preguntas

- ¿Cómo sé cuándo terminé la implementación? -> desarrollador
- ¿Conseguí lo que quería? -> usuario

En un proyecto ideal los usuarios deberían escribir el criterio de éxito de una story que debería traducirse a un test de aceptación y los testers y developers trabajan en su implementación. Cucumber es un ejemplo de tool que le permite al usuario escribir el test script (en su lenguaje). La tool provee el mecanismo para mantenerlos sincronizados

SOPORTE DEL PROCESO DE DESARROLLO, DE CARA AL NEGOCIO

Modelo given-when-then para tests.

Happy path:

Given [estado inicial],

when [acciones],

then [nuevo estado] will result

Alternative path para variaciones del estado inicial, acciones y o estado final, o particiones equivalentes y boundary values para determinar cuáles tests implementar. Como es de esperarse, los sad paths causan errores

Automatizar los tests de aceptación

- Costo
- Retroalimentación más rápida
- Reducen la carga de los testers
- Los testers se pueden centrar en el test exploratorio
- El conjunto de tests de aceptación también representan un conjunto de test de regresión poderoso
- Usando nombres legibles por los humanos en los tests, se puede generar la documentación del sistema a partir de los tests
- Usados para regresión

SOPORTE DEL PROCESO DE DESARROLLO, DE CARA A LA TECNOLOGÍA

Escritos y mantenidos exclusivamente por desarrolladores

- Unit tests
- Component tests
- Deployment tests

Pruebas de unidad

- Prueban una parte en particular del software aislada
- 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

- Prueban un cluster de funcionalidad mayor, para poder encontrar problemas que surgen de la interacción
- Son más lentos
- A veces se los conoce como tests de integración

Pruebas de Despliegue

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

Pruebas de aceptación

Son manuals. 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 cuánto 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

CRITICAN EL PROYECTO DE CARA A LA TECNOLOGÍA

Los tests de aceptación vienen en dos categorías: funcionales y no funcionales

- **No funcionales:** todas las características de un sistema diferentes a la funcional: por ejemplo: capacity, availability, security, etc. El usuario muchas veces no especifica los requerimientos no funcionales desde un principio pero los sobreentiende muchas veces. Requieren una cantidad considerable de recursos y de tools específicas. Son más lentos y tienden a correrse más hacia el final del deployment pipeline

Test en el Commit Stage

La mayoría deberían ser test de unidad: deben ser rápidos y deben tener buena cobertura (~80% o más)

Los unit tests por sí solos no son suficiente para asegurar que la aplicación funciona, para eso está el resto del deployment pipeline.

¿Cómo lograr que los test de unidad sean rápidos?

- Evitar la interfaz de usuario
 - 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
 - Buen diseño, permiten introducción de test doubles
- Evitar la base de datos
 - cualquier otro subsistema que no sea el testeado
 - Debería ser sencillo de evitar
- Evitar el asincronismo
 - En el commit stage debe ser evitada
 - Dividir los tests

¿Cómo lograr que los test de unidad sean rápidos?

- 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
 - Injectar una implementación que simula el tiempo
- Fuerza bruta
 - Paralelizar la ejecución
 - Separar los tests suites para poder paralelizar

Test doubles

Una parte clave del test automatizado involucra reemplazar parte del sistema con una versión simulada. Esto se hace para lograr tener más control sobre el comportamiento. Estas partes simuladas se denominan “test doubles”

Tipos:

- Dummy objects, se pasan pero nunca se usan
- Fake objects, tienen implementaciones funcionales, pero tomando atajos que los hacen no usables en producción - in-memory db
- 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

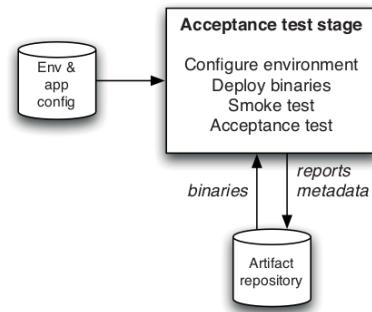
TESTS EN EL ACCEPTANCE TEST STAGE

Test de Aceptación Automatizado

- Son una parte crucial en el deployment pipeline
- Lleva a los equipos más allá del simple CI
- Testean el criterio de aceptación del negocio
- Se corren contra toda versión de la aplicación que pase el commit stage

¿Por qué son esenciales?

- El costo de testear manualmente es alto, dado que tiene que repetirse con cada nueva versión para comprobar si puede ser reseñada o no
- Testing manual suele hacerse al último
 - Los defectos se capturan tarde
 - Suele planificarse incorrectamente
- Los tests de unidad y de componente no se basan en testear escenarios por lo que no son suficiente
- Los tests de aceptación proveen una protección contra cambios grandes en el diseño o la arquitectura
 - Los tests de unidad y de componente tienen que rehacerse pero no los tests de aceptación
- Proveen una presión saludable para tener requerimientos de calidad



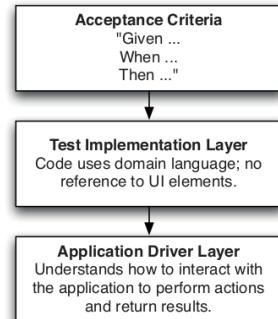
Test de Aceptación Automatizados Mantenibles

- Están basados en los criterios de aceptación por lo que estos deben escribirse con la automatización en mente
- Una vez que se tiene el COS se debe automatizar
- Los tests de aceptación deben siempre ser implementados en capas

Primera capa

- Criterio de aceptación
- Herramientas como cucumber permiten poner el COS directamente en el test y linkearlo automáticamente con la implementación
- También puede utilizarse los nombres de los métodos para lograr esto
- Es fundamental que se haga en lenguaje del dominio y no contenga detalles de la interacción con la aplicación porque se vuelven frágiles
- Los tests de aceptación no deberían ir contra la UI
 - Si bien la UI es la forma de interactuar del usuario, no debería contener lógica de negocios
- Tests contra la misma API que utiliza la UI es preferible

Figure 8.2 Layers in acceptance tests



Business Analyst / Product Owner / Representante del cliente y los usuarios

- Trabajan con los usuarios para identificar y priorizar los requerimientos
- Los developers para asegurarse que los developers entienden bien la aplicación y la perspectiva del usuario y para asegurarse que las stories entreguen el valor de negocio esperado
- Los testers para asegurar que el CoS está especificado apropiadamente y para asegurar que la funcionalidad desarrollada cumple con el COS y entrega el valor esperado
- Este es un rol, no una persona

COS COMO ESPECIFICACIÓN EJECUTABLE

En un entorno ágil, los tests automatizados no solo sirven a su propósito de testear el software. También son una especificación ejecutable del comportamiento del software que se está desarrollando

BDD -> CoS debe ser escrita en la forma de expectativas del usuario respecto del comportamiento del sistema. Entonces, es posible ejecutarla directamente contra la aplicación para verificar que cumple con la especificación

Ventajas

No quedan out-of-date porque fallan en su ejecución apenas lo hacen

- El deployment pipeline las verifica para cada cambio
- Con los enfoques tradicionales para especificar un sistema las especificaciones desfanan rápidamente y quedan atrasadas

Los test de aceptación son business-facing. CoS captura el valor de negocio y debe usarse para validar

Cucumber permite escribir el CoS de una manera que puede automatizarse su validación

Given some initial context, // Given: Estado inicial necesario para iniciar el test

When an event occurs, // When: Describe la interacción entre el usuario y la aplicación

Then there are some outcomes. // Then: Describe el estado de la aplicación después que la interacción se ha completado

Ejemplo

- Feature: Placing an order
- Scenario: User order should debit account correctly
- Given there is an instrument called bond
- And there is a user called Dave with 50 dollars in his account
- When I log in as Dave
- And I select the instrument bond
- And I place an order to buy 4 at 10 dollars each
- And the order is successful
- Then I have 10 dollars left in my account

EN RESUMEN

Discutir el criterio de aceptación de la story con el cliente

Escribirlo en un formato ejecutable

Escribir la implementación del tests que usa solo lenguaje del dominio y accede a la capa de aplicación “driver”

Crear la capa de aplicación “driver” que encapsula la forma de hablar con el sistema que está siendo testeado

Esto representa un avance muy grande respecto de mantener criterios de aceptación en documentos word y tener que trackear que tests los valida

Application Driver Layer

- Es la capa que entiende como hablar con la aplicación
- La interfaz de esta capa está expresada en lenguaje del dominio
- Si está bien diseñada podría incluso obviarse la capa de CoS y expresar el COS en la capa de implementación
- Ejemplo de capa de implementación usando un driver layer bien diseñada
- Si está bien diseñada hace más fácil la lectura
- DSL abstrae la complejidad
- Suele utilizarse aliases en el driver layer que permite ejecución de los tests en paralelo

Windows Driver Pattern

- La parte del application driver layer que interactúa con la GUI se conoce como windows driver
- Permite desacoplar los tests de aceptación de la GUI del sistema
- Hace los tests más robustos, aislando los tests de los detalles de la UI
- Un driver para cada parte de la GUI, equivalente a los device drivers
- Los códigos de los tests solo hablan con la GUI a través del driver apropiado

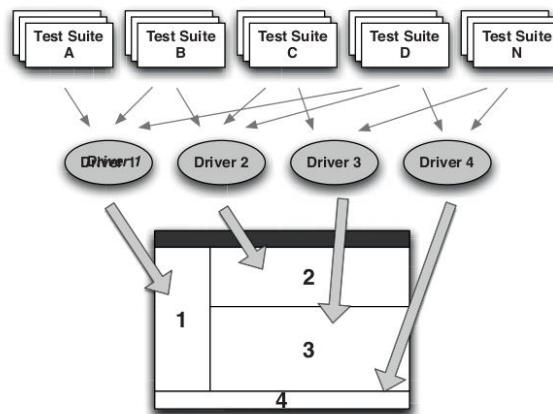


Figure 8.3 The use of the window driver pattern in acceptance testing

Beneficios de los drivers:

- Si cambia algo actualizo solo el driver que se ve afectado. Si no funciona cambio el driver y vuelvo todo a la versión anterior.
- Encapsulamiento, permite cambiar la implementación y un solo lugar para cambios.

MANEJANDO EL ESTADO

Los unit tests deberían ser stateless, pero no es posible lograrlo con los tests de aceptación

La intención de los tests de aceptación es

- simular las interacciones del usuario con el sistema
- ejercitar el sistema y
- probar que este cumple con los requerimientos del negocio

Para poder probar el comportamiento de la aplicación el test depende de que la aplicación esté en un estado específico inicial. Esto puede ser difícil

Recomendaciones:

- evitar data dumps de producción para inicializar la base de datos de tests
- mantener un set mínimo, coherente y controlado de datos que permita explorar el comportamiento del sistema
- punto de comienzo conocido
- colección de scripts guardados en el VCS que se aplican al comienzo de la ejecución de los tests
- debería usarse la API pública de la aplicación para poner la aplicación en el estado correcto para evitar correr los scripts directamente contra la base de datos
- El test ideal debería ser atómico: crear lo que necesita para su ejecución y luego “limpiar” y revertir la aplicación al estado anterior
- tratar de usar características de la aplicación para aislar los tests, por ejemplo usuarios, áreas de trabajo, particiones, etc

ENCAPSULACIÓN Y TESTING

- los test de aceptación ponen una presión en hacer el diseño mejor
- si hay que romper el encapsulamiento para poder acceder a información para el test, entonces se está perdiendo un diseño mejor
- No debería tener que crearse código específico para que los tests puedan acceder a estado interno de la aplicación
- Resistir la tentación de crear
 - Backdoors for testing
 - Code for testing mode
- ¿Se pueden usar magic values? Solo para simular cosas que no se controlan

ASINCRONÍA Y TIMEOUTS

En los unit tests se debe evitar los tests asincronos, cada parte se debe testear por separado, esto no es posible en los tests de aceptación

Los test de aceptación se deben ejecutar en un entorno como el de producción. No debería incluir integración con sistemas externos, sino más bien, el entorno debería ser controlable, es decir poder controlar el estado de los tests

Por otro lado queremos testear la integración con otros sistemas, hay una tensión en qué hacer: crear tests doubles para los sistemas externos

Pero también tener tests suites pequeños y específicos para testear los puntos de integración, que deberán correrse en un entorno contra los sistemas reales

Usar tests doubles permite

- controlar el estado
- controlar el comportamiento
- simular fallas de comunicación
- simular respuestas de error

Un buen diseño debería minimizar el acoplamiento entre la aplicación y el sistema externo. Generalmente un solo componente de nuestra aplicación maneja la comunicación con un sistema externo:

- Un adapter o gateway por sistema externo
- Permite implementar patrones como circuit breaker

Desacoplar nuestro Sistema del Sistema externo agrega una capa donde podemos agregar control

- Retries

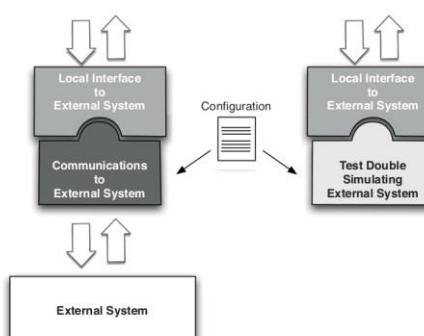


Figure 8.4 Test doubles for external systems

- Circuit breaking
- Timeouts

Testear los puntos de integración externos

- No testear el sistema externo
- Testear la integración

Testar puntos de interacción

- Dependerán del estado del sistema externo
 - maduro en producción
 - bajo desarrollo active
- Solo testear lo necesario para la interacción
 - no enfocarse en testear la interfaz del sistema externo completo
- Adoptar un approach empírico, crear tests que cubran los problemas que se van descubriendo

DEPLOYMENT TESTS

El entorno para los acceptance tests debe ser lo más parecido a producción que se pueda. En lo posible idénticos

Esto es una excelente oportunidad para testear los instaladores en un entorno como el de producción

Además, se debe testear que los componentes se pueden comunicar correctamente

Deberían sumarse como otro tests suite al principio que hagan que el stage del pipeline falle si estos fallan de lo contrario puede que todos los tests esperen el máximo timeout antes de fallar

UNIDAD 6

MÉTRICAS

GESTIÓN DE LA CALIDAD

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

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

1. A nivel de organización, la gestión de calidad se ocupa de establecer un marco de proceso y estándares de organización que conducirán a software de mejor calidad. Esto supone que el equipo de gestión de calidad debe tener la responsabilidad de definir los procesos de desarrollo del software a usar, los estándares que deben aplicarse al software y la documentación relacionada, incluyendo los requerimientos, el diseño y el código del sistema.
2. A nivel del proyecto, la gestión de calidad implica la aplicación de procesos específicos de calidad y la verificación de que continúen dichos procesos planeados; además, se ocupa de garantizar que los resultados del proyecto estén en conformidad con los estándares aplicables a dicho proyecto.
3. A nivel del proyecto, la gestión de calidad se ocupa también de establecer un plan de calidad para un proyecto. El plan de calidad debe establecer metas de calidad para el proyecto y definir cuáles procesos y estándares se usarán.

Los términos aseguramiento de calidad y control de calidad se utilizan ampliamente en la industria manufacturera.

El **aseguramiento de calidad** (QA, por las siglas de quality assurance) es la definición de procesos y estándares que deben conducir a la obtención de productos de alta calidad y, en el proceso de fabricación, a la introducción de procesos de calidad.

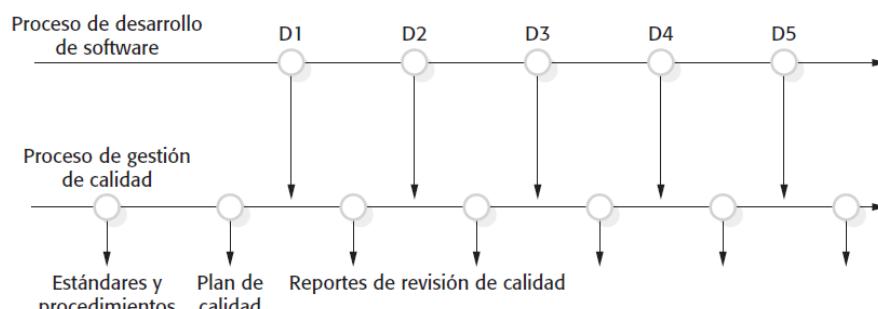
El control de calidad es la aplicación de dichos procesos de calidad para eliminar aquellos productos que no cuentan con el nivel requerido de calidad.

En la industria de software, diversas compañías y sectores industriales interpretan de maneras diferentes el aseguramiento de calidad y el control de calidad. En ocasiones, el aseguramiento de calidad representa simplemente la definición de procedimientos, procesos y estándares cuyo objetivo es asegurar el logro de calidad del software. En otros casos, el aseguramiento de calidad incluye también todas las actividades de gestión de configuración, verificación y validación aplicadas después de que un equipo de desarrollo entrega un producto.

En la mayoría de las compañías, el equipo QA es el responsable de administrar el proceso de pruebas de liberación. El equipo es responsable de comprobar que las pruebas del sistema cubren los requerimientos y de mantener los registros adecuados del proceso de pruebas.

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

De preferencia, el equipo de gestión de calidad no debe asociarse con algún grupo de desarrollo particular; sin embargo, tiene la responsabilidad ante toda la organización por la administración de la calidad. El equipo debe ser independiente y reportarse ante la administración ubicada sobre el nivel del administrador del proyecto.



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

Al usar medición de software, un sistema podría valorarse preferentemente mediante un rango de métricas y, a partir de dichas mediciones, se podría inferir un valor de calidad del sistema. Si el software alcanzó un umbral de calidad requerido, entonces podría aprobarse sin revisión. Cuando es adecuado, las herramientas de medición pueden destacar también áreas del software susceptibles de mejora. Sin embargo, aún se está lejos de esta situación ideal y no hay señales de que la valoración automatizada de calidad será en el futuro una realidad previsible.

Una métrica de software es una característica de un sistema de software, documentación de sistema o proceso de desarrollo que puede medirse de manera objetiva.

TIPOS DE MÉTRICAS

Las métricas de software pueden ser métricas de control o de predicción.

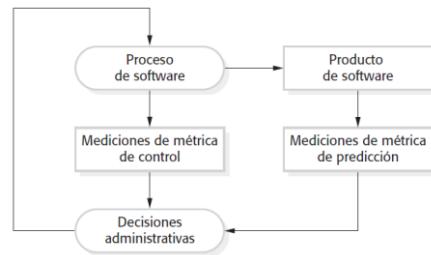
→ Las **métricas de control o de proceso** apoyan la gestión del proceso, y las métricas de predicción ayudan a predecir las características del software. Se asocian por lo general con procesos de software. Ejemplos:

- El esfuerzo promedio.
- El tiempo requerido para reparar los defectos reportados.
- El tiempo que lleva completar un proceso particular: calendar time, etc.
- Los recursos requeridos para un proceso particular: cuantas 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

→ Las **métricas de predicción**, o métricas de producto, se asocian con el software en sí, ayudan a predecir características del software. Ejemplos:

- Complejidad ciclomática
- Longitud promedio de los identificadores
- Número de atributos y operaciones que tiene una clase

Tanto las métricas de control como las de predicción pueden influir en la toma de decisiones administrativas, como se muestra en la figura. Los managers usan *mediciones de proceso* para decidir si deben hacerse cambios al proceso, y las *métricas de predicción* ayudan a estimar el esfuerzo requerido para hacer cambios al software.



USO DE MEDICIONES

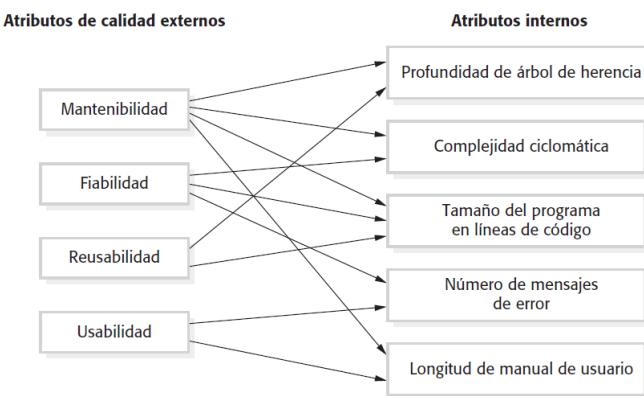
Existen dos formas en que pueden usarse las mediciones de un sistema de software:

1. Para **asignar un valor a los atributos de calidad del sistema**. Al medir las características de los componentes del sistema, como su complejidad ciclomática, y luego agregar dichas mediciones, es posible valorar los atributos de calidad del sistema, tales como la mantenibilidad.
2. Para **identificar los componentes del sistema cuya calidad está por debajo de un estándar**. Las mediciones pueden identificar componentes individuales con características que se desvían de la norma. Por ejemplo, es posible medir componentes para descubrir aquéllos con la complejidad más alta. Éstos tienen más probabilidad de tener bugs porque la complejidad los hace más difíciles de entender.

Lamentablemente, es difícil hacer mediciones directas de muchos de los atributos de calidad del software. Los atributos de calidad, como mantenibilidad, comprensibilidad y usabilidad, son atributos externos que se refieren a cómo los desarrolladores y usuarios experimentan el software. Se ven afectados por factores subjetivos, como la experiencia y educación del usuario, y, por lo tanto, no pueden medirse de manera objetiva. Para hacer un juicio sobre estos atributos, hay que medir algunos atributos internos del software (como tamaño, complejidad, etcétera) y suponer que éstos se relacionan con las características de calidad por las que uno se interesa.

La figura 24.10 muestra algunos atributos externos de calidad del software y atributos internos que podrían, intuitivamente, relacionarse con ellos. Aunque el diagrama sugiere que pueden existir relaciones entre atributos externos e

internos, no dice cómo se relacionan dichos atributos. Si la medida del atributo interno debe ser un factor de predicción útil de la característica externa del software, deben sostenerse tres condiciones



1. El atributo interno debe medirse con exactitud. Esto no siempre es un proceso directo y tal vez se requiera de herramientas de propósito especial para hacer las mediciones.
2. Debe existir una relación entre el atributo que pueda medirse y el atributo de calidad externo que es de interés. Esto es, el valor del atributo de calidad debe relacionarse, en alguna forma, con el valor del atributo que puede medirse.
3. Esta relación entre los atributos interno y externo debe comprenderse, validarse y expresarse en términos de una fórmula o un modelo. La formulación de un modelo implica identificar la manera funcional del modelo (lineal, exponencial, etcétera) mediante el análisis de datos recopilados, identificar los parámetros que se incluirán en el modelo, y calibrar dichos parámetros usando los datos existentes.

O se puede utilizar data-mining para descubrir relaciones y hacer predicciones sobre los atributos externos.

En la década de 1990, numerosas y grandes compañías introdujeron programas de métricas. Hicieron mediciones de sus productos y procesos y las usaron durante sus procesos de gestión de calidad. La mayor parte de la atención se centró en la recolección de métricas sobre los defectos de programa y los procesos de verificación y validación.

Existe escasa información disponible al público concerniente al uso actual en la industria de la medición sistemática del software. Muchas compañías reúnen información referente a su software, como el número de peticiones de cambio de requerimientos o el número de defectos descubiertos en las pruebas. Sin embargo, no es claro si usan entonces dichas mediciones de manera sistemática para comparar productos y procesos de software o para valorar el efecto de los cambios sobre los procesos y las herramientas de software. Existen algunas razones por las que esto se dificulta:

- Es imposible cuantificar la rentabilidad de la inversión de introducir un programa de métricas organizacional. En años pasados existieron significativas mejoras en la calidad del software sin el uso de métricas, así que es difícil justificar los costos iniciales de introducir medición y valoración sistemáticas del software.
- No hay estándares para las métricas de software o para los procesos estandarizados para medición y análisis. Muchas compañías son renuentes a introducir programas de medición hasta que se hallan disponibles tales estándares y herramientas de apoyo.
- En gran parte de las compañías, los procesos de software no están estandarizados y se encuentran mal definidos y controlados. Por lo tanto, hay demasiada variabilidad de procesos dentro de la misma compañía para que las mediciones se usen en una forma significativa.
- Buena parte de la investigación en la medición y métricas del software se enfoca en métricas basadas en códigos y procesos de desarrollo basados en un plan. Sin embargo, ahora cada vez más se desarrolla software mediante la configuración de sistemas ERP o COTS, o el uso de métodos ágiles. Por consiguiente, no se sabe si la investigación previa es aplicable a dichas técnicas de desarrollo de software.
- La introducción de medición representa una carga adicional a los procesos. Esto contradice las metas de los métodos ágiles, los cuales recomiendan la eliminación de actividades de proceso que no están directamente relacionadas con el desarrollo de programas. En consecuencia, es improbable que las compañías que adoptaron los métodos ágiles aprueben un programa de métricas.

MÉTRICAS DEL PRODUCTO

Las métricas del producto son métricas de predicción usadas para medir los atributos internos de un sistema de software. Por desgracia, como se explicó anteriormente, las características del software que pueden medirse fácilmente, como el tamaño y la complejidad ciclomática, no tienen una relación clara y consistente con los atributos de calidad como comprensibilidad y mantenibilidad. Las relaciones varían dependiendo de los procesos de desarrollo, la tecnología empleada y el tipo de sistema a diseñar.

Las métricas del producto se dividen en dos clases:

- **Métricas dinámicas**, que se recopilan mediante mediciones hechas de un programa en ejecución. Dichas métricas pueden recopilarse durante las pruebas del sistema o después de que el sistema está en uso. Un ejemplo es el número de reportes de bugs o el tiempo necesario para completar un cálculo.
- **Métricas estáticas**, las cuales se recopilan mediante mediciones hechas de representaciones del sistema, como el diseño, el programa o la documentación. Ejemplos de mediciones estáticas son el tamaño del código y la longitud promedio de los identificadores que se usaron.

Estos tipos de métrica se relacionan con diferentes atributos de calidad. Las métricas dinámicas ayudan a valorar la eficiencia y fiabilidad de un programa. Las métricas estáticas ayudan a valorar la complejidad, comprensibilidad y mantenibilidad de un sistema de software o de los componentes del sistema.

Por lo general, existe una relación clara entre métricas dinámicas y características de calidad del software. Es muy sencillo medir el tiempo de ejecución requerido para funciones particulares y valorar el tiempo requerido con la finalidad de iniciar un sistema. Éstos se relacionan directamente con la eficiencia del sistema. De igual modo, el número de fallas del sistema y el tipo de fallas pueden registrarse y relacionarse directamente con la fiabilidad del software.

MÉTRICAS ESTÁTICAS

Métrica de software	Descripción
Fan-in/Fan-out	Fan-in (abanco de entrada) es una medida del número de funciones o métodos que llaman a otra función o método (por ejemplo, X). Fan-out (abanco de salida) es el número de funciones a las que llama la función X. Un valor alto para fan-in significa que X está estrechamente acoplado con el resto del diseño y que los cambios a X tendrán extensos efectos dominó. Un valor alto de fan-out sugiere que la complejidad global de X puede ser alta debido a la complejidad de la lógica de control necesaria para coordinar los componentes llamados.
Longitud de código	Ésta es una medida del tamaño de un programa. Por lo general, cuanto más grande sea el tamaño del código de un componente, más probable será que el componente sea complejo y proclive a errores. Se ha demostrado que la longitud del código es una de las métricas más fiables para predecir la proclividad al error en los componentes.
Complejidad ciclomática	Ésta es una medida de la complejidad del control de un programa. Tal complejidad del control puede relacionarse con la comprensibilidad del programa. En el capítulo 8 se estudia la complejidad ciclomática.
Longitud de identificadores	Ésta es una medida de la longitud promedio de los identificadores (nombres para variables, clases, métodos, etcétera) en un programa. Cuanto más largos sean los identificadores, es más probable que sean significativos y, por ende, más comprensible será el programa.
Profundidad de anidado condicional	Ésta es una medida de la profundidad de anidado de los enunciados if en un programa. Los enunciados if profundamente anidados son difíciles de entender y proclives potencialmente a errores.
Índice Fog	Ésta es una medida de la longitud promedio de las palabras y oraciones en los documentos. Cuanto más alto sea el valor del índice Fog de un documento, más difícil será entender el documento.

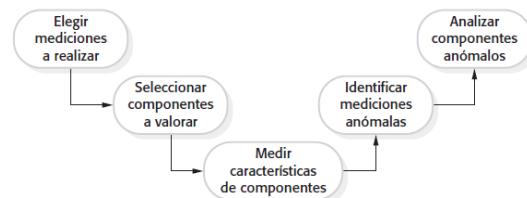
MÉTRICAS ORIENTADAS A OBJETOS (OO)

Métrica de software	Descripción
Fan-in/Fan-out	Fan-in (abanco de entrada) es una medida del número de funciones o métodos que llaman a otra función o método (por ejemplo, X). Fan-out (abanco de salida) es el número de funciones a las que llama la función X. Un valor alto para fan-in significa que X está estrechamente acoplado con el resto del diseño y que los cambios a X tendrán extensos efectos dominó. Un valor alto de fan-out sugiere que la complejidad global de X puede ser alta debido a la complejidad de la lógica de control necesaria para coordinar los componentes llamados.
Longitud de código	Ésta es una medida del tamaño de un programa. Por lo general, cuanto más grande sea el tamaño del código de un componente, más probable será que el componente sea complejo y proclive a errores. Se ha demostrado que la longitud del código es una de las métricas más fiables para predecir la proclividad al error en los componentes.
Complejidad ciclomática	Ésta es una medida de la complejidad del control de un programa. Tal complejidad del control puede relacionarse con la comprensibilidad del programa. En el capítulo 8 se estudia la complejidad ciclomática.
Longitud de identificadores	Ésta es una medida de la longitud promedio de los identificadores (nombres para variables, clases, métodos, etcétera) en un programa. Cuanto más largos sean los identificadores, es más probable que sean significativos y, por ende, más comprensible será el programa.
Profundidad de anidado condicional	Ésta es una medida de la profundidad de anidado de los enunciados if en un programa. Los enunciados if profundamente anidados son difíciles de entender y proclives potencialmente a errores.
Índice Fog	Ésta es una medida de la longitud promedio de las palabras y oraciones en los documentos. Cuanto más alto sea el valor del índice Fog de un documento, más difícil será entender el documento.

ANÁLISIS DE COMPONENTES DE SOFTWARE

En la figura 24.13 se ilustra un proceso de medición que puede ser parte de un proceso de valoración de calidad del software. Cada componente del sistema puede analizarse por separado mediante un rango de métricas. Los valores de dichas métricas pueden compararse entonces para diferentes componentes y, tal vez, con datos de medición históricos recopilados en proyectos anteriores. Las mediciones anómalas, que se desvian significativamente de la norma, pueden implicar que existen problemas con la calidad de dichos componentes.

Las etapas clave en este proceso de medición de componentes son:



1. **Elegir las mediciones a realizar** Deben formularse las preguntas que la medición busca responder, y definir las mediciones requeridas para responder a tales preguntas. Deben recopilarse las mediciones que no son directamente relevantes para dichas preguntas.
2. **Seleccionar componentes a valorar** Probablemente usted no necesite estimar valores métricos para todos los componentes en un sistema de software, ya que en ocasiones podrá seleccionar una muestra representativa de componentes para medición, que le permitirá realizar una valoración global de la calidad del sistema. En otras circunstancias, tal vez desee enfocarse en los componentes centrales del sistema que están casi en uso constante. La calidad de dichos componentes es más importante que la de aquellos componentes que sólo se usan muy pocas veces.
3. **Medir las características de los componentes** Se miden los componentes seleccionados y se calculan los valores de métrica asociados. Por lo general, esto implica procesar la representación de los componentes (diseño, código, etcétera) mediante una herramienta de recolección automatizada de datos. Esta herramienta puede escribirse especialmente o ser una característica de las herramientas de diseño que ya están en uso.
4. **Identificar mediciones anómalas** Después de hacer las mediciones de componentes, se comparan entonces unas con otras y con mediciones anteriores que se hayan registrado en una base de datos de mediciones. Hay que observar los valores inusualmente altos o bajos para cada métrica, pues éstos sugieren que podría haber problemas con el componente que muestra dichos valores.
5. **Analizar componentes anómalos** Cuando identifique los componentes con valores anómalos para sus métricas seleccionadas, debe examinarlos para decidir si dichos valores de métrica anómalos significan que la calidad del componente se encuentra o no comprometida. Un valor de métrica anómalo para la complejidad (al parecer) no necesariamente significa un componente de mala calidad. Podría haber alguna otra razón para el valor alto, por lo que no necesariamente significa que haya problemas con la calidad del componente.

MÉTRICAS

- **Response Time / Tiempo de Respuesta**
Tiempo que un sistema requiere para procesar un pedido visto desde fuera
- **Responsiveness / Tiempo de reacción**
Cuán rápido un sistema acepta (hace acknowledge) un pedido
¿Cuándo el tiempo de reacción es igual al tiempo de respuesta?
- **Latency / Latencia**
El tiempo mínimo requerido para obtener cualquier tipo de respuesta
Sistemas locales vs sistemas remotos
- **Throughput / Rendimiento**
Cuanto trabajo puede hacer el sistema en una cantidad dada de tiempo
Bytes/s – Transactiones/s
- **Load / Carga / Stress**
Es una medida de cuan estresado/cargado están un sistema
Usando generalmente como contexto de otra medida de performance
Tiempo de respuesta 0.5 s con 10 usuarios y 2 s con 20 usuarios
- **Load Sensitivity / Sensibilidad a la Carga**
Una expresión que indica cómo varía el tiempo de respuesta con la carga
Sistema A: 0.5 s con 10 a 20 usuarios
Sistema B: 0.2 s con 10 usuarios y 2 s con 20 usuarios
- **Efficiency / Eficiencia**
Es la performance dividido los recursos empleados
Sistema A: 30 tps con 2 CPUs
Sistema B: 40 tps con 4 CPUs (iguales a la anterior)

→ **Scalability / Escalabilidad**

Es una medida que indica como se ve afectada la performance cuando se agregan nuevos recursos

Scale Up: Agregar más o mejores recursos a un servidor

Scale Out: Agregar más servidores

→ **Capacidad**

Máxima carga o máximo rendimiento efectivos.

Máximo absoluto o un punto donde la performance se vuelve inaceptable