Contenido

INTRODUCCIÓN AL DEVOPS

DevOps es un enfoque que integra las áreas de desarrollo de software (Development) y operaciones de TI (Operations), con el objetivo de mejorar la colaboración, la eficiencia y la entrega continua de valor al cliente.

Más que una herramienta o una metodología, DevOps es una cultura organizacional basada en la cooperación, la automatización y la mejora continua.

Su propósito principal es eliminar las barreras tradicionales entre los equipos de desarrollo y de operaciones, que históricamente trabajaban de manera aislada ("en silos"). Al hacerlo, se consigue una entrega de software más rápida, confiable y de mejor calidad.

BREVE HISTORIA DE DEVOPS

El término DevOps surge entre 2007 y 2009:

2007: Patrick Debois, consultor belga, observó los conflictos recurrentes entre los equipos de desarrollo y operaciones en un proyecto gubernamental.

2008: Durante la Agile Conference en Toronto, asistió a una sesión llamada "Agile Infrastructure" impartida por Andrew Shafer.

2009: Ambos organizaron el primer evento DevOpsDays en Bélgica, donde nació oficialmente el término DevOps y se popularizó el hashtag #DevOps.

Desde entonces, DevOps se ha expandido globalmente como un estándar de buenas prácticas para la entrega continua de software.

LA BARRERA DE CONFUSIÓN: DESARROLLO VS. OPERACIONES

Durante años, los equipos de desarrollo y operaciones tuvieron objetivos opuestos:

Desarrollo buscaba el cambio constante, la innovación y la entrega de nuevas características.

Operaciones priorizaba la estabilidad, la seguridad y la disponibilidad del sistema.

Este conflicto generó problemas como:

- Entregas manuales e infrecuentes.
- Silos organizacionales y falta de comunicación.
- Pérdida de tiempo y reprocesos.
- Entornos inconsistentes.
- "El juego de las acusaciones" cuando algo fallaba en producción.

DevOps nace para superar esta barrera de confusión y alinear a ambos equipos bajo objetivos compartidos: velocidad, estabilidad y mejora continua.

BENEFICIOS DE ADOPTAR DEVOPS

Implementar prácticas DevOps permite a las organizaciones:

- Acelerar la entrega de software, reduciendo los ciclos de desarrollo.
- Aumentar la estabilidad y disponibilidad de los entornos productivos.
- Reducir costos operativos gracias a la automatización y estandarización.
- Fomentar la innovación, al liberar tiempo para tareas de valor agregado.
- Mejorar la calidad, mediante pruebas automatizadas e integración continua.
- Facilitar la colaboración, rompiendo silos y promoviendo una cultura compartida.

PRINCIPALES PRÁCTICAS DE DEVOPS

1. Colaboración

Fomentar una comunicación abierta y continua entre desarrollo, operaciones y otras áreas del negocio.Todos los equipos comparten metas, responsabilidades y conocimiento.

2. Automatización

Automatizar las tareas repetitivas y propensas a errores, como la compilación, las pruebas, los despliegues y la gestión de infraestructura.

Ejemplos: Jenkins, GitHub Actions, Terraform, Ansible.

3. Integración Continua (CI)

El código desarrollado se integra de forma continua en un repositorio central. Esto permite detectar errores tempranamente y asegurar la calidad del producto en cada commit.

4. Entrega Continua (CD)

Garantiza que el software esté siempre listo para desplegarse en producción mediante pruebas automáticas y despliegues controlados.

5. Monitoreo Continuo

Recoge métricas y registros del software en producción. Permite detectar incidentes, optimizar el rendimiento y aprender del comportamiento real de los usuarios.

RELACIÓN DE DEVOPS CON AGILE Y LEAN TI

DevOps hereda sus principios de Agile y Lean:

Agile promueve la entrega iterativa, la colaboración y la retroalimentación constante.

Lean TI busca eliminar desperdicios, optimizar procesos y maximizar el valor entregado al cliente.

Ambas filosofías se integran en DevOps, que lleva los principios ágiles más allá del desarrollo, abarcando todo el ciclo de vida del software: planificación, codificación, construcción, prueba, liberación, despliegue y operación.

METODOLOGÍAS ÁGILES RELACIONADAS

Entre las metodologías más influyentes para DevOps encontramos:

- Scrum: gestión ágil de proyectos mediante roles, eventos y artefactos bien definidos.
- Kanban: gestión visual del flujo de trabajo para optimizar la entrega.
- Extreme Programming (XP): promueve prácticas técnicas como integración continua y desarrollo guiado por pruebas.
- Lean, Crystal y FDD: metodologías centradas en eficiencia, simplicidad y valor al cliente.

Principios comunes de Agile y DevOps

1. Comunicación y colaboración constantes.
2. Entrega incremental y continua.
3. Flexibilidad ante el cambio.
4. Orientación al cliente y retroalimentación temprana.
5. Equipos autoorganizados y multidisciplinarios.
6. Mejora continua y aprendizaje constante.

Scrum

Scrum es un proceso ágil que se puede usar para gestionar y controlar desarrollos complejos de software y productos usando prácticas iterativas e incrementales.

Scrum es un esqueleto de proceso que incluye un conjunto de prácticas y roles predefinidos. Los roles principales en Scrum son el "**ScrumMaster**" que mantiene los procesos y trabaja junto con el jefe de proyecto, el "**Product Owner**" que representa a las personas implicadas en el negocio y el "**Team**" que incluye a los desarrolladores.

Durante cada iteración (**sprint**- periodos de tiempo), típicamente un periodo de 2 a 4 semanas (longitud decidida por el equipo), el equipo crea un incremento de software operativo. El conjunto de características que entra en una iteración viene del "**backlog**" del producto, que es un conjunto priorizado de requisitos de trabajo de alto nivel que se han de hacer. Los ítems que entran en una iteración se determinan durante la reunión de planificación de la iteración. Durante esta reunión, el **Product Owner** informa al equipo de los ítems en el backlog del producto que quiere que se completen. El equipo determina entonces a cuanto de eso puede comprometerse a completar durante la siguiente iteración. Durante una iteración, nadie puede cambiar el backlog de la iteración, lo que significa que los requisitos están congelados para esa iteración. Cuando se completa una iteración, el equipo demuestra el uso del software.

Un principio clave de Scrum es el reconocimiento de que durante un proyecto los clientes pueden cambiar sus pensamientos sobre lo que quieren y necesitan, y de que los desafíos que no se pueden predecir no se pueden tratar fácilmente de una forma predictiva o planificada tradicional. Por esto, Scrum adopta un enfoque empírico, aceptando que el problema no se puede entender o definir completamente, centrándose en cambio en maximizar las habilidades del equipo para entregar rápidamente y responder a los requisitos emergentes.

Roles

1. Product Owner: define y prioriza el Product Backlog; representa al cliente.
2. Scrum Master: guía al equipo, elimina impedimentos y promueve la mejora continua.
3. Equipo de desarrollo: grupo autoorganizado que entrega incrementos funcionales del producto.

Eventos

1. Sprint Planning: planificación del trabajo a realizar.
2. Daily Scrum: reunión diaria breve para sincronizar esfuerzos.
3. Sprint Review: revisión del trabajo terminado y retroalimentación del cliente.
4. Sprint Retrospective: reflexión sobre el proceso y búsqueda de mejoras.

Artefactos

Product Backlog: lista priorizada de funcionalidades.

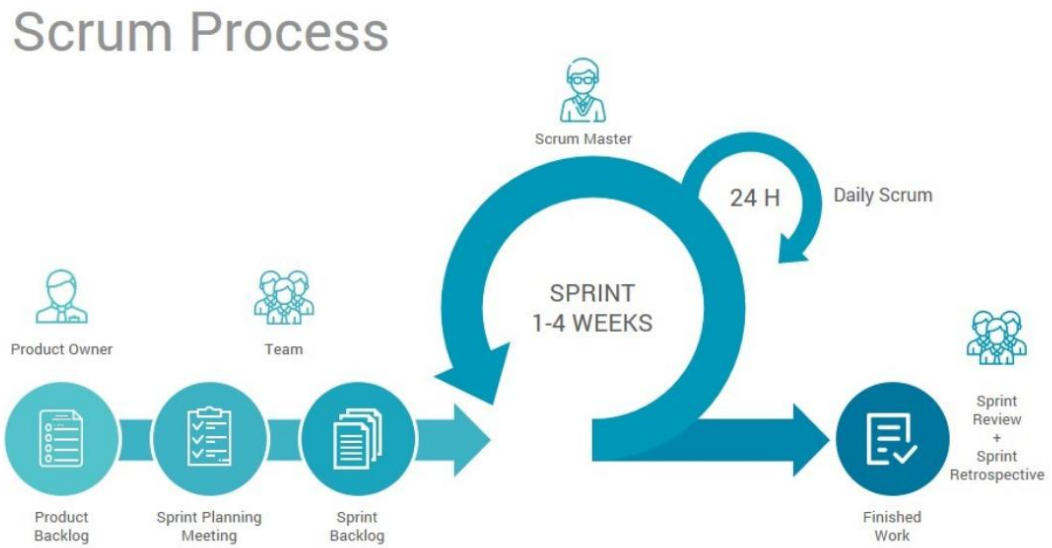
Sprint Backlog: tareas seleccionadas para el sprint actual.

Incremento: producto funcional entregable al final del sprint.

Scrum Board

Tablero visual que muestra el flujo de trabajo: Por hacer → En progreso → Hecho.

Facilita la transparencia, la priorización y la colaboración del equipo.



Extreme Programming (XP)

La programación extrema (XP) es un enfoque de la ingeniería del software formulado por **Kent Beck**. Es el más destacado de los procesos ágiles de desarrollo de software. Al igual que éstos, la programación extrema se diferencia de las metodologías tradicionales principalmente en que pone más énfasis en la **adaptabilidad** que en la previsibilidad. Los defensores de XP consideran que los cambios de requisitos sobre la marcha son un aspecto natural, inevitable e incluso deseable del desarrollo de proyectos. Creen que ser capaz de adaptarse a los cambios de requisitos en cualquier punto de la vida del proyecto es una aproximación mejor y más realista que intentar definir todos los requisitos al comienzo del proyecto e invertir esfuerzos después en controlar los cambios en los requisitos.

Se puede considerar la programación extrema como la adopción de las mejores metodologías de desarrollo de acuerdo a lo que se pretende llevar a cabo con el proyecto y aplicarlo de manera dinámica durante el ciclo de vida del software.

XP es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en el desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. XP se basa en la realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. XP se define como especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico.

Elementos de la metodología:

- Las historias de usuario: son la técnica utilizada para especificar los requisitos del software. Se trata de tarjetas de papel en las cuales el cliente describe brevemente las características que el sistema debe poseer, sean requisitos funcionales o no funcionales. El tratamiento de las historias de usuario es muy dinámico y flexible. Cada historia de usuario es lo suficientemente comprensible y delimitada para que los programadores puedan implementarlas en unas semanas. Las historias de usuario

se descomponen en tareas de programación y se asignan a los programadores para ser implementadas durante una iteración.

- Roles XP:

Programador: el programador escribe las pruebas unitarias y produce el código del sistema

Cliente: escribe las historias de usuario y las pruebas funcionales para validar su implementación. Además, asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración centrándose en apoyar mayor valor al negocio.

Encargado de pruebas (tester): ayuda al cliente a escribir las pruebas funcionales. Ejecuta las pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para las pruebas.

Encargado de seguimiento (tracker): proporciona realimentación al equipo. Verifica el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, para mejorar futuras estimaciones. Realiza el seguimiento del progreso de cada iteración.

Entrenador (coach): es el responsable del proceso global. Debe proveer guías al equipo de forma que se apliquen las prácticas XP y se siga el proceso correctamente.

Consultor: es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto, en el que puedan surgir problemas.

Gestor (big boss): es el vínculo entre clientes y programadores, ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es de coordinación.

- Proceso XP: el ciclo de desarrollo consiste en los siguientes pasos:

1) El cliente define el valor de negocio a implementar.

2) El programador estima el esfuerzo necesario para su implementación.

3) El programador construye ese valor.

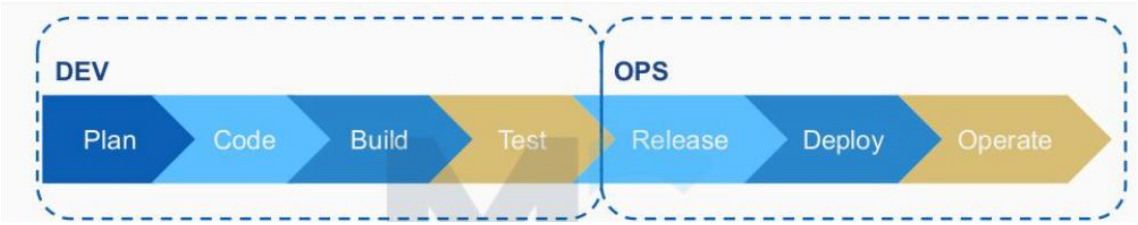
4) Vuelve al paso 1.

En todas las iteraciones de este ciclo tanto el cliente como el programador aprenden. No se debe presionar al programador a realizar más trabajo que el estimado, ya que se perderá calidad en el software o no se cumplirán los plazos. De la misma forma el cliente tiene la obligación de manejar el ámbito de entrega del producto, para asegurarse de que el sistema tenga el mayor valor de negocio posible.

El ciclo de vida ideal de XP consiste en 6 fases:

1. exploración,
2. planificación de la entrega,
3. iteraciones,
4. producción,
5. mantenimiento
6. muerte del proyecto.

CICLO DE VIDA DEVOPS



El ciclo iterativo típico en DevOps se compone de las fases:

Cada fase está soportada por herramientas específicas (Jira, Git, Jenkins, Docker, Kubernetes, Prometheus, etc.), y todas están conectadas por la automatización y la mejora continua.

Un proyecto de integración continua necesita, al menos, los siguientes componentes:

CONSTRUCCIÓN BUILD

Se detallan las herramientas agrupadas según el estado en la producción de software:

- **Desarrollar (Codificar)**: en esta categoría están aquellas herramientas encaminadas a mejorar el rendimiento del desarrollo. Las podemos dividir en subcategorías: o **IDEs (entornos de desarrollo integrado) para el desarrollo**: Eclipse, IntelliJ IDEA, NetBeans, Visual Studio (Microsoft), Xcode (Mac), Android Studio.

o **Editores de texto**: Sublime Text, Notepad++, GNU nano, GNU Emacs.

o **Sistemas de gestión de repositorios para control de versiones**: Git, GitHub, GitLab, Bitbucket, FishEye.

o **Herramientas de análisis (estático) de calidad de código**: SonarQube. El análisis nos dice cómo de bueno es nuestro código, los errores que tiene, los tipos de errores encontrados y las causas, y añade una serie de posibles soluciones. También, califica la calidad del código, valor que se actualiza continuamente tras cada modificación del código del proyecto, por lo que puede evolucionar tanto positiva como negativamente. Esto permite supervisar de forma continua la calidad del código del software de cada proyecto, y analizar tendencias. SonarQube evalúa aspectos como la complejidad ciclomática del código (número de caminos independientes), número de comentarios o las líneas de código que no se ejecutan. Asimismo, informa sobre código duplicado, estándares de codificación, pruebas unitarias, cobertura de código, posibles errores y diseño del software.

o **Revisión de código**: Crucible, Gerrit.

o **Seguimiento de errores y bugs**: Jira, Redmine, Bugzilla, MantisBT, Trac.

o **Herramientas para hacer merges (fusiones) y comparar diferencias entre versiones**: KDiff3, DiffMerge, P4Merge, JDiff.

Construcción (**Build**): en esta categoría estarían encuadradas aquellas herramientas cuya finalidad es la de preparar el código para su ejecución y prueba. El rol de estas herramientas es verificar que un desarrollo se ha validado y puede pasarse a un entorno de staging o de producción. Si se utilizan de forma correcta permiten eliminar gran parte del tiempo de pruebas que tienen que emplear los

desarrolladores y además asegura la detección de problemas inesperados introducidos en partes del desarrollo que antes del cambio funcionaban bien. Herramientas:

Integración continua (compilación y ejecución pruebas de todo el proyecto): Jenkins (open source y escrito en Java), Bamboo, CircleCI, CruiseControl, Go.CD, Travis CI.

Automatización de los builds: ▪

1. Apache Maven: gestión y construcción de proyectos software. Utiliza el Project Object Model (POM, fichero pom.xml) para describir el proyecto software a construir, sus dependencias con módulos externos y el orden de construcción de los elementos.
2. **Apache Ant**: automatización de tareas repetitivas y mecánicas como construcción de paquetes y proyectos.
3. **Make, Gradle**, Kotlin DSL, MSBuild (.NET).

- **Test**: el objetivo de las herramientas de test es reducir los casos de prueba manual al mismo tiempo que se incrementa la cobertura, y por tanto ahorrando tiempo de creación de la automatización y esfuerzo y tiempo de ejecución de los tests. Herramientas de test:

1. **Continuo web**: Selenium.
2. **Continuo móvil**: Appium.
3. **Iterativo**: Lux, Expect.
4. **Para criterios de aceptación**: FitNesse, FIT, EasyAccept.
5. **Unitarios**: XUnit, JUnit, Unit.js, ATF.
6. **Servicios web**: SoapUI y Postman (RESTful).

- **Empaquetado**: en esta categoría se agrupan las herramientas encaminadas a evitar la dependencia de proveedores y repositorios externos en los procesos de build y para poder mantener controladas las librerías propias de la organización, así como para poder compartir con clientes externos los desarrollos. Herramientas:

1. **Repositorio de artefactos**: Nexus 3, Artifactory. Útiles para subir y compartir librerías de terceros así como para publicar los artefactos (por ejemplo, WAR o JAR) de nuestra aplicación.
2. **Contenedores**: Docker.
3. **Gestión documental**: Confluence (wiki), Owncloud (alojamiento de archivos, open source).

- **Gestión de la configuración**: esta categoría agrupa aquellas herramientas diseñadas con el propósito de usar scripts, recetas, blueprints (plan software), playbooks, plantillas, etc. para la simplificación de la automatización y orquestación en los entornos de ejecución de la organización. El objetivo es proporcionar un modo estándar y consistente de realización de los despliegues y sus configuraciones. Herramientas: o Puppet, Salt, Chef, Ansible, Vagrant, Terraform y GitOps (prácticas para gestionar configuraciones de aplicaciones y las infraestructuras utilizando Git).

- **Despliegue**: dentro de esta categoría se incluyen las herramientas y servicios que permiten la ejecución sobre infraestructura tanto de las herramientas necesarias para DevOps como de los desarrollos realizados. Herramientas:

1. o **Orquestación de infraestructura privada**: Gradiant ITBox, MaaS, Openstack, Kubernetes.

2. o **Orquestación de servicios**: Docker Compose, Docker Swarm.
3. o **Orquestación multicloud**: Gradiant ITBox, Cloudify, Apache Brooklyn.
4. o **Virtualización**: Docker, KVM (Kernel-based Virtual Machine).
5. o **Cloud públicos**: AWS (Amazon Web Services), Microsoft Azure, Google CE (Compute Engine).

Monitorización: Uno de los muros que DevOps trata de romper es la falta de feedback sobre las aplicaciones puestas en producción una vez se ha realizado el despliegue por parte del departamento de operaciones. La monitorización debe ser algo transversal y debe llegar sin filtros al departamento de desarrollo que podrán detectar e interpretar los resultados del día a día de sus desarrollos.

Herramientas:

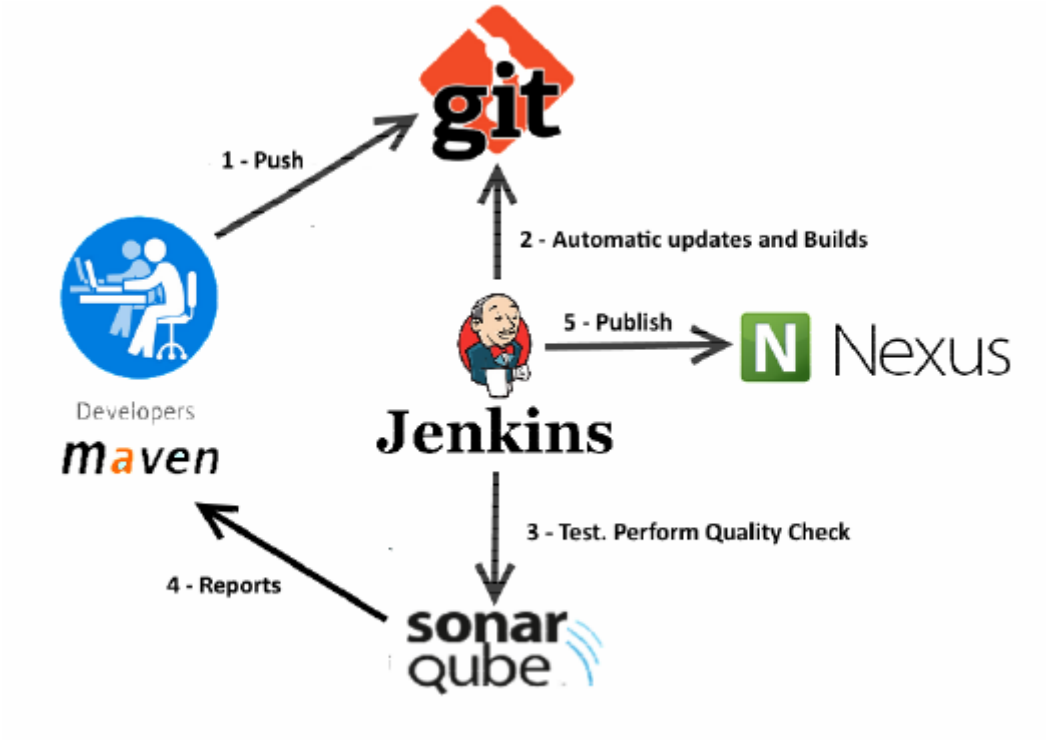
o **Rendimiento de aplicaciones**: Zabbix, Nagios, Influxdb, Telegraf, Grafana.

o **Revisión de logs**: GrayLog, Logstash, Kibana, Elasticsearch.

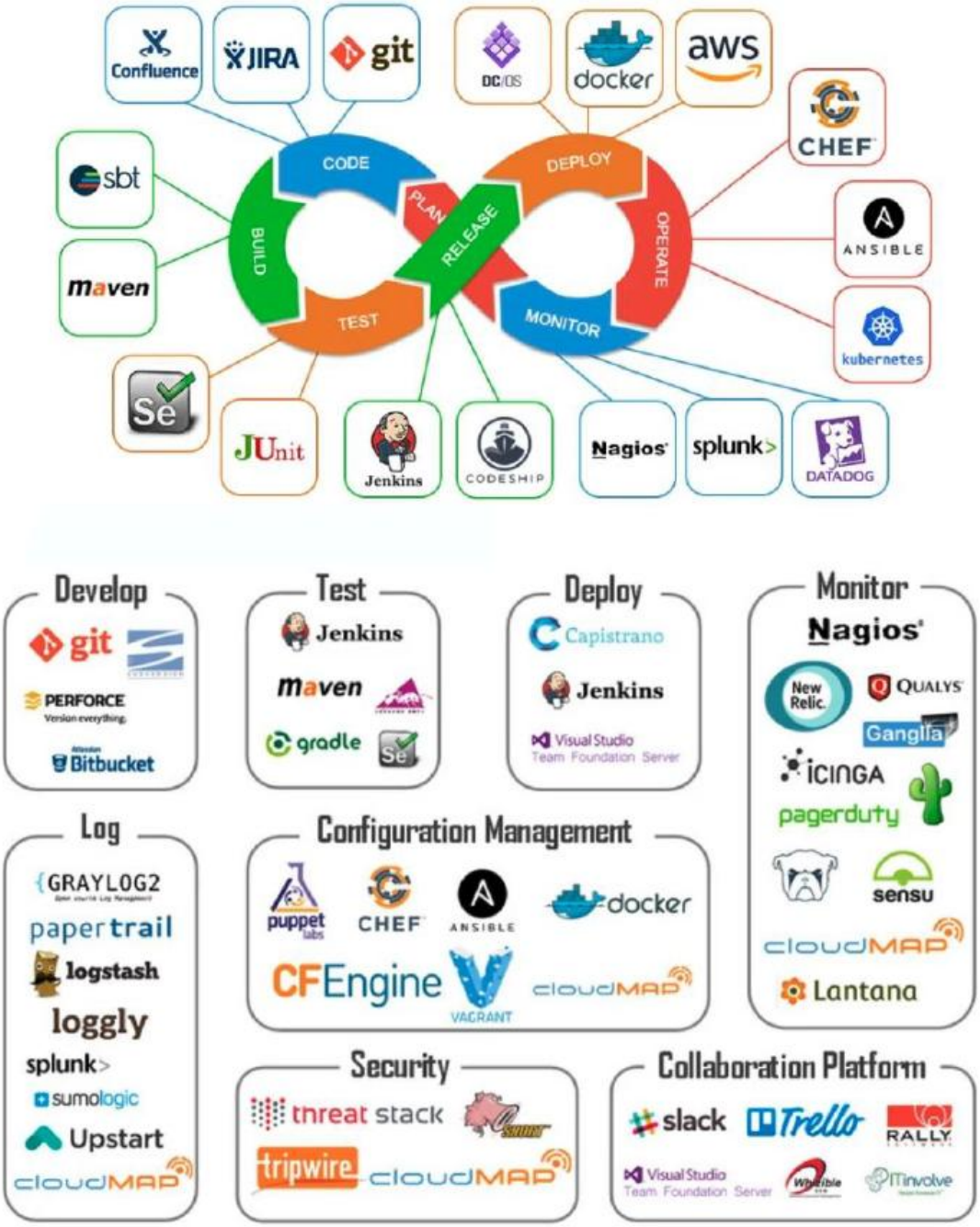
o **Experiencia de usuario y analíticas web**: Piwik PRO, Matomo (PHP/MySQL), Google Web Analytics.

Sistema de integración continua

Definimos un sistema de integración continua (compilación y ejecución pruebas de todo el proyecto) sencillo que persigue que cada vez que un programador haga un push en el repositorio remoto (en este caso, Git) y suba código a la rama master, Jenkins lo detecte, compile la aplicación, llame a SonarQube para ejecutar los test y realizar el análisis de la calidad del código y, si todo ha ido bien, publique un WAR del proyecto en Nexus



Resumen de herramientas


Confluence
JIRA
git
DC/OS
docker
aws
CHEF
sbt
Maven
CODE
PLAN
BUILD
TEST
RELEASE
DEPLOY
OPERATE
MONITOR
ANSIBLE
kubernetes
Se
JUnit
Jenkins
CODESHIP
Nagios
splunk>
DATADOG
Develop
git
PERFORCE
Bitbucket
Test
Jenkins
Maven
gradle
Deploy
Capistrano
Jenkins
Visual Studio
Team Foundation Server
Monitor
Nagios
New Relic
QUALYS
Ganglia
ICINGA
pagerduty
sensu
cloudMAP
Lantana
Log
GRAYLOG2
papertrail
logstash
loggly
splunk>
sumologic
Upstart
cloudMAP
Configuration Management
puppet
CHEF
ANSIBLE
docker
CFEngine
VAGRANT
cloudMAP
Security
threat stack
tripwire
cloudMAP
Collaboration Platform
slack
Trello
RALLY
Visual Studio
Team Foundation Server
ITinvolve

Introducción

En el desarrollo de software moderno, la calidad, la velocidad y la fiabilidad son aspectos críticos. La automatización de pruebas se ha convertido en una pieza clave para alcanzar entregas frecuentes sin comprometer la estabilidad.

¿Qué es la Pirámide de Pruebas?

La Pirámide de Pruebas es una metáfora que nos ayuda a estructurar un buen portafolio de pruebas automatizadas
En su versión original (según Mike Cohn) se compone de tres capas, de abajo arriba:

1. Pruebas de unidad (unit tests)
2. Pruebas de servicio / integración (service tests)
3. Pruebas de interfaz de usuario (UI tests)

Fowler advierte que, aunque la pirámide es un buen "regla de pulgar", su nomenclatura o estructura puede quedar corta o simplista en entornos modernos.

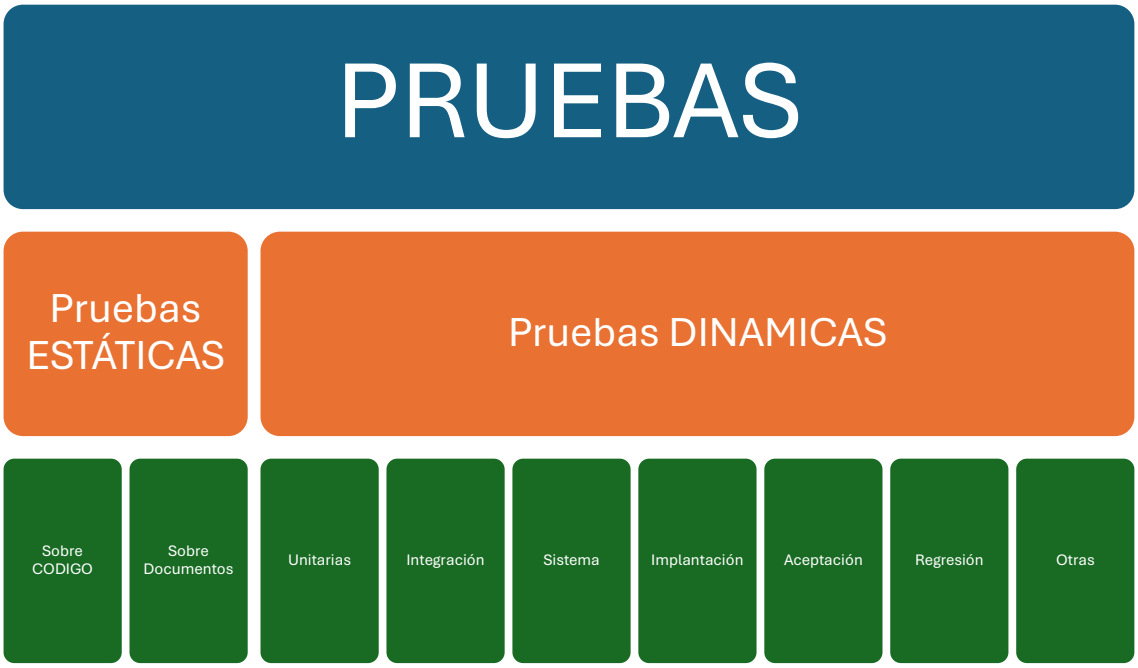
Cómo aplicar en la práctica

- Diseñar la suite de pruebas con una proporción saludable: por ejemplo, muchas unidades, un nivel medio de integración, muy pocas E2E.
- Automatizar la ejecución de las pruebas en el pipeline de integración continua / entrega continua. Recuerda el objetivo: feedback rápido y fiable.
- Elegir bien las pruebas de alto nivel: que validen lo que realmente importa.
- Evitar el "cono de pruebas" (*test ice-cream cone*): muchas pruebas lentas y superficiales en lugar de pruebas bien estructuradas.
- Mantener las pruebas limpias: no escribir pruebas que dependan excesivamente del diseño interno (evitar acoplamiento innecesario), porque dificultan la refactorización.

Las pruebas automatizadas son esenciales para la entrega continua y la calidad del software.

Debemos priorizar pruebas de unidad rápidas y numerosas; luego pruebas de integración que verifiquen dependencias; y mantener un número limitado de pruebas de extremo a extremo enfocadas en los flujos críticos.

Una buena estrategia de pruebas hará que el equipo pueda moverse rápido sin perder confianza.



INTRODUCCION

Las pruebas son prácticas para realizar en diversos momentos de la vida del sistema de información para verificar:

- El correcto funcionamiento de los componentes del sistema.

- El correcto ensamblaje entre los distintos componentes.

- El funcionamiento correcto de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistemas de información con los que se comunica.

- El funcionamiento correcto del sistema integrado de hardware y software en el entorno de operación.

- Que el sistema cumple con el funcionamiento esperado y permite al usuario de dicho sistema que determine su aceptación, desde el punto de vista de su funcionalidad y rendimiento.

- Que los cambios sobre un componente de un sistema de información no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados.

Las diversas pruebas a que debe ser sometido un sistema deben ser realizadas tanto por el equipo de desarrolladores, como por los usuarios, equipos de operación y mantenimiento en la implantación, aceptación y mantenimiento del sistema de información.

Según Métrica v3, los **tipos de pruebas** que deben realizarse son:

- Pruebas **Unitarias**.
- Pruebas de **Integración**.
- Pruebas del **Sistema**.
- Pruebas de **Implantación**.
- Pruebas de **Aceptación**.
- Pruebas de **Regresión**.

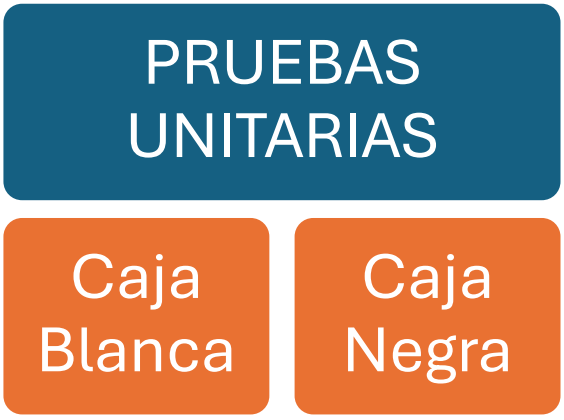
--Otras

PRUEBAS UNITARIAS

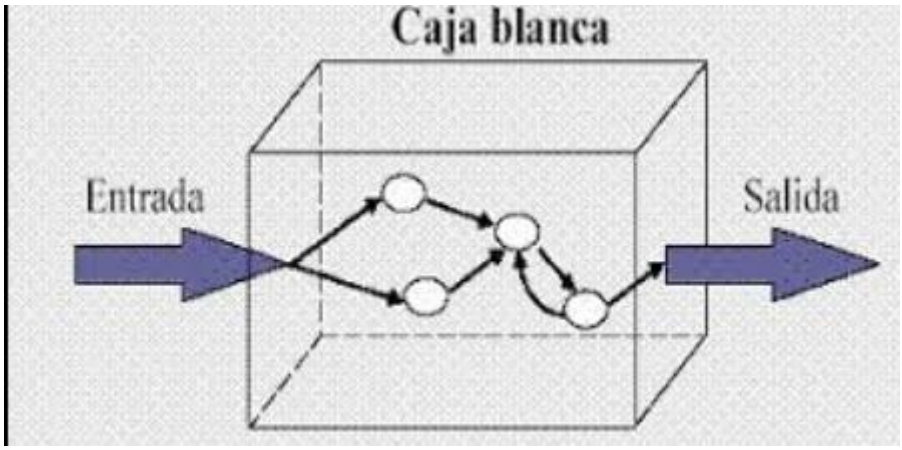
Las pruebas unitarias tienen como objetivo verificar la funcionalidad y estructura de cada componente **individualmente** una vez que ha sido codificado.

Las pruebas unitarias constituyen la prueba inicial de un sistema y las demás pruebas deben apoyarse sobre ellas.

Existen dos **enfoques** principales para el diseño de casos de prueba:

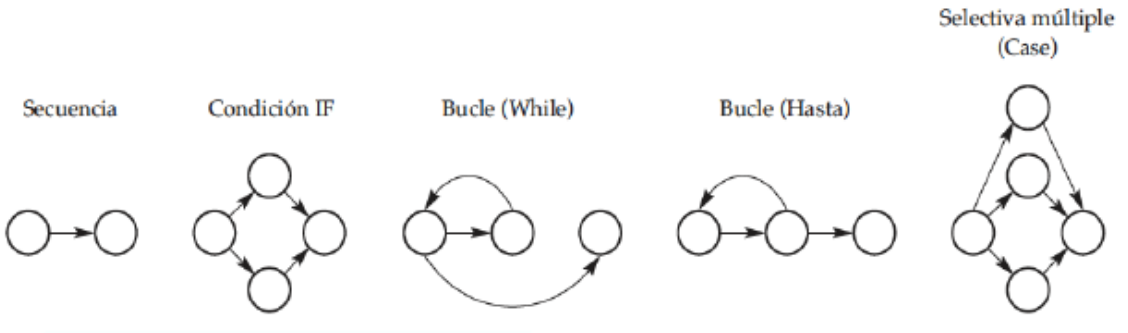


- **Enfoque estructural o de CAJA BLANCA (logic driven)**. Se verifica la estructura interna del componente con independencia de la funcionalidad establecida para el mismo. Por tanto, no se comprueba la corrección de los resultados si éstos se producen. Ejemplos de este tipo de pruebas pueden ser ejecutar todas las instrucciones del programa, localizar código no usado, comprobar los caminos lógicos del programa, etc.



Dentro de este enfoque distinguimos las siguientes **técnicas** de pruebas:

- o **De interfaz**: analiza el flujo de datos hacia y desde el componente, para asegurar que la información fluye adecuadamente.
- o **De estructuras de datos locales**: asegura la integridad de los datos durante la ejecución del código. Las pruebas que se realizan en los datos: referencias (accesos a los datos), declaraciones (definiciones), cálculos (errores de uso de datos) y comparaciones (errores en sentencias condicionales o repetitivas).
- o **Del camino básico**: permite la obtención de una medida de la complejidad lógica del diseño procedimental.



Se divide en:

▪ **Cobertura de sentencias**: se crean casos de prueba para probar todas las sentencias de un componente.

$$\text{COBERTURA DE SENTENCIAS} = \frac{\text{N.º SENTENCIAS EJECUTADAS EN LA PRUEBA}}{N.º \text{ TOTAL SENTENCIAS}}$$

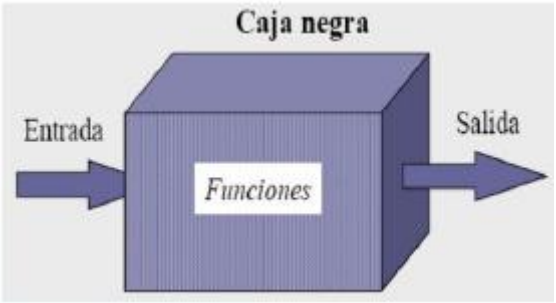
▪ **Cobertura de condiciones**: se crean casos de prueba para probar todos los posibles valores en sentencias condicionales y repetitivas.

$$\text{COBERTURA DE CONDICIONES} = \frac{\text{N.º CONDICIONES EJECUTADAS EN LA PRUEBA}}{N.º \text{ TOTAL CONDICIONES}}$$

Estos dos tipos de pruebas son complementarios entre sí y están orientados a cubrir la ejecución de cada una de las sentencias, cada una de las decisiones y cada una de las condiciones en las decisiones, tanto en su vertiente verdadera como falsa.

- o **De bucles o de condiciones límite**: se comprueba la validez de un bucle.

- **Enfoque funcional o de CAJA NEGRA (data driven)**. Se comprueba el correcto funcionamiento de los componentes del sistema de información, analizando las entradas y salidas y verificando que el resultado es el esperado. Se consideran exclusivamente las entradas y salidas del sistema sin preocuparse por la estructura interna del mismo.



Dentro de este enfoque destacamos las siguientes **técnicas** de prueba:

- o **Particiones de equivalencia**: se basa en definir juegos de prueba que representen el conjunto de todas las entradas posibles. Técnica sustentada en el concepto de clase de equivalencia, agrupación de un conjunto de valores de una determinada entrada donde el comportamiento del componente es igual, considerándose su estado como válido o inválido.

¿Cómo determinar las clases de equivalencia? Por ejemplo, si un parámetro de entrada debe estar comprendido en un cierto rango, surgen 3 clases de equivalencia: una clase para los valores por debajo del límite inferior, una clase para los valores dentro del rango y otra clase para los valores que superan el límite superior. Si es obligatorio que una entrada tome un valor de entre los de un conjunto dado, tenemos 2 clases de equivalencia: una clase para los valores dentro del conjunto y otra clase para los valores fuera del conjunto dado.

- o **Análisis de los valores límite o frontera**: parte de la base que proporciona la experiencia de que los errores suelen aparecer con mayor probabilidad en los extremos de los campos de entrada. Se seleccionan los valores justo por encima y por debajo de los límites de la clase de equivalencia. Complementa la técnica anterior y maximiza el número de errores con un menor número de pruebas.

Ejemplo: si una condición de entrada especifica un rango delimitado por los valores a y b, se deben diseñar casos de prueba para los valores a y b, y para los valores justo por debajo y justo por encima de a y b, respectivamente.

- o **Valores típicos de error y valores imposibles**: consiste en probar ciertos valores según la entrada concreta que, a priori, deben causar un error. La elección de los valores depende de la experiencia de los desarrolladores.

- o **Tabla de decisión**: se identifican las condiciones (a menudo entradas) y las acciones resultantes (a menudo salidas) del sistema. Éstas conforman las filas de la tabla, generalmente con las condiciones en la parte superior y las acciones en la parte inferior. Cada columna corresponde a una regla de decisión que define una combinación única de condiciones que resulta en la ejecución de las acciones

asociadas a esa regla. Los valores de las condiciones y acciones, normalmente se muestran como valores booleanos (verdadero o falso) o valores discretos (por ejemplo, rojo, verde, azul), pero también pueden ser números o intervalos de números. Estos diferentes tipos de condiciones y acciones pueden estar juntos en la misma tabla.

Valores para las condiciones: Verdadero o 1, Falso o 0.

Valores para las acciones: X si debe ocurrir, en blanco si no debe ocurrir.

- o **Transición de estado**: un diagrama de transición de estado muestra los posibles estados del software, así como la forma en que el software entra, sale y realiza las transiciones entre estados. Una transición se inicia con un evento (por ejemplo, la entrada de un valor por parte del usuario en un campo). El evento resulta en una transición. Si el mismo evento puede resultar en dos o más transiciones diferentes desde el mismo estado, ese evento puede estar condicionado por una condición de guarda. El cambio de estado puede provocar que el software tome una acción (por ejemplo, emitir el resultado de un cálculo o un mensaje de error).

Una tabla de transición de estado muestra todas las transiciones válidas y las transiciones potencialmente inválidas entre estados, así como los eventos, las condiciones de guarda y las acciones resultantes para las transiciones válidas. Los diagramas de transición de estado, normalmente, sólo muestran las transiciones válidas y excluyen las transiciones no válidas.

La prueba de transición de estado se utiliza para aplicaciones basadas en menús y es extensamente utilizada en la industria del software embebido. La técnica también es adecuada para modelar un escenario de negocio con estados específicos o para probar la navegación en pantalla.

- o **Caso de uso**: las pruebas se obtienen a partir de casos de uso, que son una forma específica de diseñar interacciones con elementos software, incorporando requisitos para las funciones del software representadas por los casos de uso.

- o **Grafo causa-efecto**.

- o **Pruebas de comparación**.

El enfoque que suele adoptarse para una prueba unitaria está claramente orientado al diseño de casos de caja blanca, aunque se complemente con caja negra. El hecho de incorporar casos de caja blanca se debe, por una parte, a que el tamaño del componente es apropiado para poder examinar toda la lógica y por otra, a que el tipo de defectos que se busca coincide con los propios de la lógica detallada en los componentes.

Los pasos necesarios para llevar a cabo las pruebas unitarias son los siguientes:

- **Ejecutar todos los casos de prueba** asociados a cada verificación establecida en el plan de pruebas, registrando su resultado. Los casos de prueba deben contemplar tanto las condiciones válidas y esperadas como las inválidas e inesperadas.

- **Corregir los errores** o defectos encontrados y repetir las pruebas que los detectaron. Si se considera necesario, debido a su implicación o importancia, se repetirán otros casos de prueba ya realizados con anterioridad.

La prueba unitaria se da por **finalizada** cuando se hayan realizado todas las verificaciones establecidas y no se encuentre ningún defecto, o bien se determine su suspensión.

PRUEBAS DE INTEGRACIÓN

El objetivo de las pruebas de integración es verificar el correcto **ensamblaje** entre los distintos componentes una vez que han sido probados unitariamente con el fin de comprobar que interactúan correctamente a través de sus interfaces, tanto internas como externas, cubren la funcionalidad establecida y se ajustan a los requisitos no funcionales especificados en las verificaciones correspondientes.

En las pruebas de integración se examinan las interfaces entre grupos de componentes o subsistemas para asegurar que son llamados cuando es necesario y que los datos o mensajes que se transmiten son los requeridos.

Debido a que en las pruebas unitarias es necesario crear módulos auxiliares que simulen las acciones de los componentes invocados por el que se está probando y a que se han de crear componentes "conductores" para establecer las precondiciones necesarias, llamar al componente objeto de la prueba y examinar los resultados de la prueba, a menudo se combinan los tipos de prueba unitarias y de integración.

Los **tipos** fundamentales de integración son los siguientes:

(1) Integración INCREMENTAL - **De arriba abajo (TOP-DOWN)**. El primer componente que se desarrolla y prueba es el primero de la jerarquía (A). Los componentes de nivel más bajo se sustituyen por componentes auxiliares para simular a los componentes invocados. En este caso no son necesarios componentes conductores. Una de las ventajas de aplicar esta estrategia es que las interfaces entre los distintos componentes se prueban en una fase temprana y con frecuencia.

- **De abajo arriba (BOTTOM-UP)**. En este caso se crean primero los componentes de más bajo nivel (E, F) y se crean componentes conductores para simular a los

componentes que los llaman. A continuación, se desarrollan los componentes de más alto nivel (B, C, D) y se prueban. Por último, dichos componentes se combinan con el que los llama (A). Los componentes auxiliares son necesarios en raras ocasiones.

- **Estrategias combinadas**. A menudo es útil aplicar las estrategias anteriores conjuntamente. De este modo, se desarrollan partes del sistema con un enfoque "top-down", mientras que los componentes más críticos en el nivel más bajo se desarrollan siguiendo un enfoque "bottom-up". En este caso es necesaria una planificación cuidadosa y coordinada de modo que los componentes individuales se "encuentren" en el centro.

(2) Integración NO INCREMENTAL

Se prueba cada componente por separado y posteriormente se integran todos de una vez realizando las pruebas pertinentes. Este tipo de integración se denomina también **Big-Bang** (gran explosión).

PRUEBAS DEL SISTEMA

Las pruebas del sistema tienen como objetivo ejercitar profundamente el sistema comprobando la integración del sistema de información **globalmente**, verificando el funcionamiento correcto de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistemas de información con los que se comunica.

Son pruebas de integración del sistema de información completo, y permiten probar el sistema en su conjunto y con otros sistemas con los que se relaciona para verificar que las especificaciones funcionales y técnicas se cumplen. Dan una visión muy similar a su comportamiento en el entorno de producción.

Una vez que se han probado los componentes individuales y se han integrado, se prueba el sistema de forma global. En esta etapa pueden distinguirse los siguientes **tipos** de pruebas, cada uno con un objetivo claramente diferenciado:

- **Pruebas funcionales**. Dirigidas a asegurar que el sistema de información realiza correctamente todas las funciones que se han detallado en las especificaciones dadas por el usuario del sistema.

- **Pruebas de comunicaciones**. Determinan que las interfaces entre los componentes del sistema funcionan adecuadamente, tanto a través de dispositivos remotos, como locales. Asimismo, se han de probar las interfaces hombre/máquina.

- **Pruebas de rendimiento**. Consisten en determinar que los tiempos de respuesta están dentro de los intervalos establecidos en las especificaciones del sistema.

- **Pruebas de volumen**. Consisten en examinar el funcionamiento del sistema cuando está trabajando con grandes volúmenes de datos, simulando las cargas de trabajo esperadas.

- **Pruebas de sobrecarga**. Consisten en comprobar el funcionamiento del sistema en el umbral límite de los recursos, sometiéndole a cargas masivas. El objetivo es establecer los puntos extremos en los cuales el sistema empieza a operar por debajo de los requisitos establecidos.

- **Pruebas de disponibilidad de datos**. Consisten en demostrar que el sistema puede recuperarse ante fallos, tanto de equipo físico como lógico, sin comprometer la integridad de los datos.

- **Pruebas de facilidad de uso**. Consisten en comprobar la adaptabilidad del sistema a las necesidades de los usuarios, tanto para asegurar que se acomoda a su modo habitual de trabajo, como para determinar las facilidades que aporta al introducir datos en el sistema y obtener los resultados.

- **Pruebas de operación**. Consisten en comprobar la correcta implementación de los procedimientos de operación, incluyendo la planificación y control de trabajos, arranque y rearranque del sistema, etc.

- **Pruebas de entorno**. Consisten en verificar las interacciones del sistema con otros sistemas dentro del mismo entorno.

- **Pruebas de seguridad**. Consisten en verificar los mecanismos de control de acceso al sistema para evitar alteraciones indebidas en los datos.

PRUEBAS DE IMPLANTACIÓN

El objetivo de las pruebas de implantación es comprobar el funcionamiento correcto del sistema integrado de hardware y software en el **entorno de operación**, y permitir al usuario que, desde el punto de vista de operación, realice la aceptación del sistema una vez instalado en su entorno real y en base al cumplimiento de los requisitos no funcionales especificados.

Una vez que hayan sido realizadas las pruebas del sistema en el entorno de desarrollo, se llevan a cabo las verificaciones necesarias para asegurar que el sistema funcionará correctamente en el entorno de operación. Debe comprobarse que responde satisfactoriamente a los requisitos de rendimiento, seguridad, operación y coexistencia con el resto de los sistemas de la instalación para conseguir la aceptación del usuario de operación.

Los **tipos** de pruebas de implantación son:

- Las **pruebas de SEGURIDAD** van dirigidas a verificar que los mecanismos de protección incorporados al sistema cumplen su objetivo.

- Las **pruebas de RENDIMIENTO** tienen como objetivo asegurar que el sistema responde satisfactoriamente en los márgenes establecidos en cuanto tiempos de respuesta, de ejecución y de utilización de recursos, así como los volúmenes de espacio en disco y capacidad.

- Las **pruebas de OPERACIÓN** se dirigen a comprobar que la planificación y control de trabajos del sistema se realiza de acuerdo a los procedimientos establecidos, considerando la gestión y control de las comunicaciones y asegurando la disponibilidad de los distintos recursos.

- Las **pruebas de GESTIÓN DE COPIAS DE SEGURIDAD Y RECUPERACIÓN** tienen como objetivo verificar que el sistema no ve comprometido su funcionamiento al existir un control y seguimiento de los procedimientos de salvaguarda y de recuperación de la información, en caso de caídas en los servicios o en algunos de sus componentes. Para comprobar estos últimos, se provoca el fallo del sistema, verificando si la recuperación se lleva a cabo de forma apropiada. En el caso de realizarse de forma automática, se evalúa la inicialización, los mecanismos de recuperación del estado del sistema, los datos y todos aquellos recursos que se vean implicados.

Las verificaciones de las pruebas de implantación y las pruebas del sistema tienen muchos puntos en común al compartir algunas de las fuentes para su diseño como pueden ser los casos para probar el rendimiento (pruebas de sobrecarga o de stress).

El responsable de implantación junto al equipo de desarrollo determina las verificaciones necesarias para realizar las pruebas, así como los criterios de aceptación del sistema. Estas pruebas las realiza el equipo de operación, integrado por los técnicos de sistemas y de operación que han recibido previamente la formación necesaria para llevarlas a cabo.

PRUEBAS DE ACEPTACIÓN

El objetivo de las pruebas de aceptación es validar que un sistema cumple con el funcionamiento esperado y permitir al usuario de dicho sistema que determine **su aceptación**, desde el punto de vista de su funcionalidad y rendimiento.

Las pruebas de aceptación son definidas por el usuario del sistema y preparadas por el equipo de desarrollo, aunque la ejecución y aprobación final corresponden al usuario.

Estas pruebas también son conocidas por su acrónimo en inglés, **UAT** (pruebas de aceptación de usuario).

Estas pruebas van dirigidas a comprobar que el sistema cumple los requisitos de funcionamiento esperado, recogidos en el catálogo de requisitos y en los criterios de aceptación del sistema de información, y conseguir así la aceptación final del sistema por parte del usuario.

El responsable de usuarios debe revisar los criterios de aceptación que se especificaron previamente en el plan de pruebas del sistema y, posteriormente, dirigir las pruebas de aceptación final.

La validación del sistema se consigue mediante la realización de pruebas de caja negra que demuestran la conformidad con los requisitos y que se recogen en el plan de pruebas, el cual define las verificaciones a realizar y los casos de prueba asociados. Dicho plan está diseñado para asegurar que se satisfacen todos los requisitos funcionales especificados por el usuario teniendo en cuenta también los requisitos no funcionales relacionados con el rendimiento, seguridad de acceso al sistema, a los datos y procesos, así como a los distintos recursos del sistema.

La formalidad de estas pruebas dependerá en mayor o menor medida de cada organización, y vendrá dada por la criticidad del sistema, el número de usuarios implicados en las mismas y el tiempo del que se disponga para llevarlas cabo, entre otros.

Por último, cabe destacar dentro de las pruebas de aceptación otros dos tipos de pruebas de validación:

- **Pruebas ALFA**: realizadas en las instalaciones de la organización que desarrolla, no por el equipo de desarrollo, sino por los clientes o un equipo de prueba independiente.

- **Pruebas BETA**: realizadas por los clientes en sus propias instalaciones. Estas pruebas también se conocen como pruebas de campo.

Estas pruebas pueden tener lugar después de las pruebas alfa o puede ocurrir sin que se haya realizado ninguna prueba alfa previa.

PRUEBAS DE REGRESIÓN

El objetivo de las pruebas de regresión es **eliminar el efecto onda**, es decir, comprobar que los cambios sobre un componente de un sistema de información no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados.

Las pruebas de regresión se deben llevar a cabo cada vez que se hace un cambio en el sistema, tanto para corregir un error como para realizar una mejora. No es

suficiente probar sólo los componentes modificados o añadidos, o las funciones que en ellos se realizan, sino que también es necesario controlar que las modificaciones no produzcan efectos negativos sobre el mismo u otros componentes.

Normalmente, este tipo de pruebas implica la repetición de las pruebas que ya se han realizado previamente, con el fin de asegurar que no se introducen errores que puedan comprometer el funcionamiento de otros componentes que no han sido modificados y confirmar que el sistema funciona correctamente una vez realizados los cambios.

Las pruebas de regresión pueden incluir:

- La repetición de los casos de pruebas que se han realizado anteriormente y están directamente relacionados con la parte del sistema modificada.

- La revisión de los procedimientos manuales preparados antes del cambio, para asegurar que permanecen correctamente.

- La obtención impresa del diccionario de datos de forma que se compruebe que los elementos de datos que han sufrido algún cambio son correctos.

El responsable de realizar las pruebas de regresión será el equipo de desarrollo junto al técnico de mantenimiento, quien a su vez, será responsable de especificar el plan de pruebas de regresión y de evaluar los resultados de dichas pruebas.

OTROS TIPOS DE PRUEBAS

Las **pruebas de humo** (smoke testing) son una revisión rápida de un producto de software para comprobar que funciona y no tiene defectos evidentes que interrumpan la operación básica del mismo.

Las pruebas de humo pueden resultar útiles justo después de realizar una compilación nueva para decidir si se pueden ejecutar o no pruebas más caras, o inmediatamente después de una implementación, para asegurar que la aplicación funciona correctamente en el entorno.

Las pruebas **fuzzing testing** son un conjunto de pruebas de caja negra que permiten descubrir errores de implementación mediante la introducción de datos al azar, inválidos o malformados.

El fin último de estas pruebas es provocar comportamientos inesperados, como fallos que lleguen a hacer que la aplicación deje de funcionar. Este tipo de pruebas permiten, de forma automática o semiautomática, detectar potenciales vulnerabilidades de una forma rápida. Las herramientas utilizadas para llevar a cabo estas pruebas se conocen con el nombre de **fuzzers**.

Las **pruebas de usabilidad** permiten determinar hasta qué punto el software es comprendido, aprendido, usado y atractivo para los usuarios en condiciones específicas de uso.

DevOps no es solo una metodología, sino un cambio cultural y organizacional que busca la integración total de desarrollo, operaciones y negocio.

Su adopción permite entregar valor de manera continua, ágil y confiable, adaptándose a las demandas de un entorno tecnológico en constante evolución.