

Relays 602 in 033 failed
in Relay
Relays changed
Started Cosine Tape (Sine check)
- started Multi Adder Test.

Relay #70 Panel F
(moth) in relay.

Fundamentos de las pruebas continuas de software

Emanuel Irrazábal · Maximiliano Mascheroni



First actual case of bug being found.
17:30 Antangut started.
17:00 closed down.



Fundamentos de las pruebas continuas de software

Emanuel Irrazábal · Maximiliano Mascheroni

Irrazabal, Emanuel

Fundamentos de las pruebas continuas de software/Emanuel Irrazabal;
Maximiliano A. Mascheroni. - 1a edición para el alumno - Corrientes:
Editorial de la Universidad Nacional del Nordeste EUDENE, 2022.
Libro digital, PDF/A - (Ciencia y técnica)

Archivo Digital: descarga
ISBN 978-950-656-209-0

1. Software. 2. Diseño de Software. 3. Ingeniería de Software. I. Mascheroni,
Maximiliano A. II. Título.
CDD 005.12

Edición: Natalia Passicot

Corrección: Irina Wandelow

Diseño y diagramación: Ma. Belén Quiñonez



© EUDENE. Secretaría de Ciencia y Técnica,
Universidad Nacional del Nordeste, Corrientes, Argentina, 2022.

Queda hecho el depósito que marca la ley 11.723.
Reservados todos los derechos.

25 de Mayo 868 (CP 3400) Corrientes, Argentina.
Teléfono: (0379) 4425006
eudene@unne.edu.ar / www.eudene.unne.edu.ar

*Para mis hermanos
Emanuel I.*

*Para mi mamá
Agustín M.*

| | |
|--|----------|
| ORIGEN Y DESTINO POSIBLE DE ESTE TEXTO | 8 |
| RESUMEN DE CONTENIDOS | 9 |
| Capítulo 1. Las primeras pruebas de software | |
| 1.1. Las pruebas en el desarrollo ágil de aplicaciones | 11 |
| 1.2. Áreas de conocimiento implicadas | 13 |
| Capítulo 2. Las pruebas y la calidad del software | |
| 2.1. La calidad en el desarrollo de software | 14 |
| 2.2. Modelos para la medición de la calidad | 16 |
| 2.3. La norma ISO/IEC 9126 | 16 |
| 2.4. La norma ISO/IEC 25010 | 19 |
| 2.5. Modelos de procesos orientados a las pruebas | 20 |
| 2.6. Metodologías ágiles | 21 |
| 2.6.1. El Manifiesto ágil | 21 |
| 2.6.2. Comparación con las metodologías tradicionales | 23 |
| Capítulo 3. Integración continua de software | |
| 3.1. El proceso de integración continua | 25 |
| 3.2. Integración continua de bases de datos | 28 |
| 3.3. Feedback continuo | 30 |
| Capítulo 4. Despliegue y entrega continua de software | |
| 4.1. Despliegue tradicional y despliegue continuo | 34 |
| 4.2. Etapas del despliegue continuo | 35 |
| 4.3. Entrega continua y despliegue continuo | 37 |
| 4.4. Componentes de entrega continua | 38 |
| 4.5. El conducto de despliegue | 39 |

| | |
|---|----|
| Capítulo 5. Tipos de pruebas | |
| 5.1. Inspecciones de código | 44 |
| 5.2. Pruebas unitarias | 45 |
| 5.3. Pruebas de integración | 48 |
| 5.4. Pruebas funcionales y no funcionales | 50 |
| 5.4.1. Pruebas funcionales | 51 |
| 5.4.2. Pruebas no funcionales | 52 |
| 5.4.3. Pruebas de aceptación del usuario | 55 |

| | |
|---|----|
| Capítulo 6. Definición y fases de las pruebas continuas de software | |
| 6.1. Importancia de las pruebas continuas de software | 57 |
| 6.2. Los problemas que tienen las pruebas en el despliegue continuo de software | 60 |
| 6.3. ¿Existe una definición válida y aceptada para las pruebas continuas? | 62 |
| 6.4. Niveles de prueba en proyectos de desarrollo de software continuo | 64 |

| | |
|--|----|
| Capítulo 7. Problemas en las pruebas continuas | |
| 7.1. Problemas detectados en la literatura académica | 67 |
| 7.2. Problemas detectados por la industria | 70 |
| 7.3. Resumen de los problemas | 71 |

| | |
|---|----|
| Capítulo 8. Soluciones en las pruebas de desarrollo continuo | |
| 8.1. Pruebas no deterministas | 73 |
| 8.2. Pruebas que consumen mucho tiempo de ejecución | 75 |
| 8.2.1. Pruebas unitarias | 75 |
| 8.2.2. Pruebas funcionales | 77 |
| 8.2.3. Pruebas automatizadas de interfaz gráfica | 78 |
| 8.2.4. Pruebas ambiguas | 79 |

| | |
|---|----|
| Capítulo 9. Uso de prácticas del desarrollo continuo de software en la industria | |
| 9.1. Buenas prácticas en el desarrollo de las pruebas | 81 |
| 9.2. Colaboración en el equipo de trabajo | 90 |




| | |
|--|------------|
| Capítulo 10. Automatización de las pruebas de software | |
| 10.1. Pruebas unitarias | 92 |
| 10.1.1. Implementación de pruebas unitarias automatizadas | 93 |
| 10.1.2. Agrupamiento de pruebas unitarias | 95 |
| 10.1.3. Cobertura de pruebas unitarias | 96 |
| 10.2. Pruebas funcionales | 97 |
| 10.2.1. Librerías de automatización de pruebas funcionales | 98 |
| 10.2.2. Agrupamiento de pruebas funcionales | 102 |
| | |
| Capítulo 11. Servidores para la construcción de pruebas continuas de software | |
| 11.1. Selenium Grid | 104 |
| 11.2. Conducto de integración continua | 106 |
| 11.2.1. Instalación del servidor de integración continua | 106 |
| 11.2.2. Configuración del conducto de integración continua | 107 |
| 11.2.3. Creación del conducto de integración continua en Jenkins | 107 |
| | |
| Anexo | 110 |



Origen y destino posible de este texto

Este libro resume el trabajo de cinco años en el Grupo de Investigación en Calidad de Software de la Universidad Nacional del Nordeste. En particular, el desarrollo de las investigaciones de una tesis doctoral y dos tesis de maestría que buscaron mejorar la forma en la que los equipos que construyen aplicaciones software llevan adelante las pruebas.

Su objetivo es dar una introducción a los conceptos básicos de una nueva metodología de trabajo: las pruebas continuas de software. Es un contenido desafiante y, aunque sea introductorio, están descritos aquí los resultados más relevantes de esas investigaciones. Se trata de un texto técnico, con un amplio contenido teórico y algunos ejemplos prácticos, especialmente en los dos últimos capítulos. 

Los autores consideramos que será útil en asignaturas relacionadas con la Ingeniería del software para el último año de una carrera de grado o como parte de un curso de posgrado. En ambos casos, sugerimos complementar este contenido con ejemplos prácticos basados en una o más tecnologías.

Puede ser utilizado asimismo en el ámbito de una empresa privada, especialmente si se ha medido el costo de «no hacer pruebas». La implementación de los conceptos, como cualquier otra metodología de trabajo, necesita de una inversión inicial de tiempo y recursos humanos altamente calificados. Los autores opinamos que bien vale la pena y que los beneficios van aumentando con el tiempo.

Finalmente, animarlos a disfrutar de la lectura y a que se pregunten luego de cada capítulo: ¿Cómo puedo incorporarlo a mi vida profesional?

Resumen de contenido

El concepto de integración continua del desarrollo de software como mecanismo moderno para la construcción de aplicaciones se define en el capítulo introductorio. Luego, en el segundo capítulo, la relación existente entre las actividades de prueba y la mejora en la calidad del producto software resultante. En el tercer capítulo, profundizamos las definiciones del capítulo 1, desde el punto de vista filosófico y técnico, lo que ayudará a insertar los conceptos posteriores de prueba continua.

Los pasos para la implementación y entrega actual de las aplicaciones software y su relación con las pruebas continuas se desarrollan en el cuarto capítulo para, después, en el quinto capítulo, describir los principales tipos de prueba tanto automatizadas como manuales, información que complementa los tres capítulos anteriores, en los que se habló de los pasos para lograr continuidad en la integración, el despliegue y la entrega del software.

Habiendo transitado los capítulos introductorios del libro, en el sexto capítulo se detallan los principales conceptos de las pruebas continuas de software para, posteriormente, en el séptimo capítulo, detallar los problemas actuales de las pruebas continuas de software. En el octavo y en el noveno capítulo, a continuación, se enumeran las principales técnicas utilizadas por las empresas a nivel mundial y los siguientes pasos para su utilización masiva en el marco del desarrollo continuo de software. Aquí se publican los resultados de las investigaciones y encuestas llevadas adelante por el Grupo de Investigación en Calidad de Software de la Universidad Nacional del Nordeste.

Finalmente, en los dos últimos capítulos, se detallan ejemplos de tecnologías y los pasos principales a tener en cuenta para implementar las actividades de pruebas continuas en un equipo de trabajo que cuente con buenas prácticas de trabajo colaborativo.



Capítulo 1. Las primeras pruebas de software

El 9 de septiembre de 1947, en la Universidad de Harvard, tuvo lugar el primer defecto informático, cuando una polilla ingresó entre las dos lengüetas de un relé. Este incidente quedó registrado (puede verse la descripción y a la responsable pegada con cinta adhesiva en la Figura 1). Desde ese momento, el término *bug* («bicho» en inglés) es utilizado para referirse a los errores en el código fuente de un software. Como respuesta a ello, se llevaron adelante tareas de inspección y corrección de estos errores: las aplicaciones software necesitaban ser probadas.

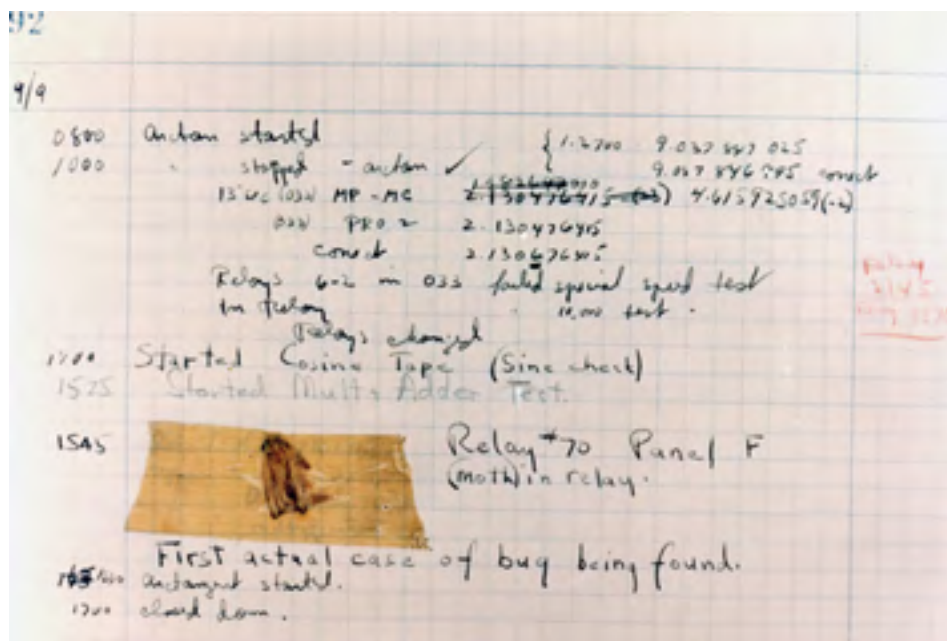


Figura 1. Registro de una falla de software en 1947 (Naval Surface Warfare Center, 1988, en US Naval Historical Center Online Library Photograph).

Allí nacería una nueva disciplina, las pruebas de software, que tienen como fin observar la ejecución de un sistema para validar si se comporta de la forma prevista o, en caso contrario, detectar sus fallas. Estas abarcan un amplio espectro de actividades, técnicas y actores, desde la prueba de una pequeña parte del código a cargo del desarrollador (prueba unitaria) hasta la validación que hace el cliente de todo el sistema (prueba de aceptación).

En todas las etapas, los escenarios de prueba se diseñan con objetivos diferentes como, por ejemplo, cumplir los requisitos del usuario, evaluar la conformidad de una especificación estándar, analizar la robustez ante condiciones de sobrecarga o medir atributos dados, como el rendimiento, la usabilidad o la estimación de la confiabilidad operativa. Además, las pruebas se llevan a cabo de acuerdo con un procedimiento formal que requiere planificación y documentación.

Como consecuencia de esta variedad de objetivos y alcances, las pruebas de software plantean muchos desafíos. Estudios tempranos en esta disciplina estimaron que las pruebas pueden consumir el cincuenta por ciento, o incluso más, de los costos de desarrollo (Beizer y Vinter, 1990). Y si revisamos encuestas realizadas a varios sectores industriales de Estados Unidos, los impactos económicos de una infraestructura de pruebas inadecuada son de miles de millones de dólares (Tassey, 2003).

Se han desarrollado y propuesto diferentes herramientas para hacer frente a estas problemáticas. Entre ellas, las pruebas automatizadas, que reducen el tiempo de ejecución y reemplazan la verificación manual que puede generar errores. Por este motivo, en los procesos de desarrollos actuales, donde el tiempo es un factor fundamental, se las ha elegido para aumentar la velocidad en la ejecución de las pruebas y así disminuir el tiempo para lanzar una nueva versión del producto al mercado. Sin embargo, las pruebas automatizadas aún presentan debilidades y características por mejorar. En este contexto, mantener el software de acuerdo a determinados niveles de calidad, en entornos de desarrollo modernos, es todavía un desafío.

1.1. LAS PRUEBAS EN EL DESARROLLO ÁGIL DE APLICACIONES

Tradicionalmente, el software se construía utilizando metodologías secuenciales, donde una etapa no podía comenzar hasta que la anterior no terminaba. En este tipo de metodologías, la ejecución de pruebas era una de las últimas etapas, de tal manera que, si se detectaban defectos en el código fuente, se generaban mayores costos para la organización y retrasos en los tiempos de entrega. Es bien conocido el crecimiento exponencial de estos costos de acuerdo a la fase en donde se encontraba el defecto. En la Figura 2 puede verse un ejemplo tomado de los desarrollos de software realizados por la Nasa, cuyos costos aumentaban entre 29 y 1615 veces, dependiendo de la etapa del ciclo de vida en donde se detectaba el error. Asimismo, puede verse que incluso en la segunda etapa, la del diseño, el factor de multiplicación era de 4 a 8 veces.

En este sentido, el desarrollo ágil de software ha incorporado una gran variedad de buenas prácticas y procesos para producir software de manera más veloz y eficiente (Cohn, 2009). Una de las más conocidas es la integración continua, una práctica de la construcción de software donde los desarrolladores, que trabajan con la copia local del

| | METHOD 1 | METHOD 2 | METHOD 3 |
|--------------|---------------------|---------------------|---------------------|
| | COST FACTORS | COST FACTORS | COST FACTORS |
| Requirements | 1X | 1X | 1X |
| Design | 8X | 3X - 4X | 4X |
| Build | 16X | 13X - 16X | 7X |
| Test | 21X | 61X - 78X | 28X |
| Operations | 29X | 157X - 186X | 1615X |

Figura 2. Evolución de los costos asociados a un defecto detectado a lo largo de un proyecto de desarrollo de software (Haskins *et al.*, 2004).

código fuente en su computadora, pasan las versiones de lo desarrollado al servidor de versiones de manera diaria. Esta frecuencia de integración hace indispensable la revisión automatizada del código (Fowler y Foemmel, 2006) que, hacia los 2000, estaba soportada solamente por herramientas de pruebas unitarias automáticas. Con la aparición de nuevas herramientas, aumentaron los tipos de prueba automatizables, siendo así introducidos en el proceso de integración continua. Sin embargo, para ejecutar pruebas de manera automática sobre una versión del código en ejecución, es necesario el despliegue en un ambiente de desarrollo. Esto dio lugar a la aparición de nuevas herramientas que automatizan el despliegue de código e incorporan este proceso, junto con la ejecución de pruebas automatizadas, a la integración continua.

Por un lado, algunas empresas o equipos de trabajo buscaban automatizar el proceso completo, desde la integración del código fuente hasta el despliegue a producción del producto software resultante. Mientras otros, por otro lado, decidieron automatizar todo el flujo de construcción, pero el despliegue a producción se realizaría de manera automática, al presionar un botón cuando el cliente lo indique. El primer enfoque se denomina actualmente *despliegue continuo* (Rodríguez, Piattini y Ebert, 2019) y el segundo recibe el nombre de *entrega continua* (Humble y Farley, 2010).

En las tres prácticas antes mencionadas, las pruebas automatizadas cumplen un rol fundamental, ya que deben brindar a los desarrolladores confiabilidad en cada versión del producto que se construye. Para ello, los especialistas en el despliegue continuo proponen un patrón llamado conducto o tubería de despliegue, más conocido en inglés como *deployment pipeline* (Humble y Farley, 2010). En una tubería de despliegue, el flujo de las tareas se divide en diferentes etapas y cada una aumenta la confiabilidad a la par que el tiempo de ejecución.

Encontrar un equilibrio entre un tiempo de ejecución pequeño y un conjunto de pruebas que aseguren la calidad de cada aplicación de software entregable es una tarea difícil. Además de ello, en la literatura pueden encontrarse revisiones y casos de estudio que reportan problemas relacionados con el despliegue continuo, que afectan la calidad del software construido.

Finalmente, tanto la industria como los investigadores afirman que aún no existe un modelo formal para la implementación del proceso de pruebas al adoptar enfoques de desarrollo continuo (Rodríguez, Piattini y Ebert, 2019).

Estimado lector, en este libro trataremos sobre estos puntos realizando una introducción a cada idea y técnica, así como también introduciremos el concepto de pruebas continuas en este marco y daremos ejemplos de técnicas para ámbitos específicos.

1.2. ÁREAS DE CONOCIMIENTO IMPLICADAS

Las áreas de conocimiento vinculadas con este libro son diversas, aunque las describimos a continuación, teniendo en cuenta el Cuerpo de Conocimiento de la Ingeniería del Software o *Software Engineering Body of Knowledge*, más conocido como Swebok (Bourque y Fairley, 2014). El documento representa una guía del conocimiento en el área de la Ingeniería del Software, como un paso esencial hacia el desarrollo de la profesión, ya que representa un amplio consenso respecto de los contenidos de la disciplina y ordena todo el conocimiento existente en una ontología. Su última versión ha sido publicada en 2014. Según el Swebok, el conocimiento en Ingeniería del Software está organizado en 15 áreas, una de las cuales es la «Prueba del software».

Tal como ilustra la Figura 3, y de acuerdo con el Swebok, los temas de este libro se integran a diferentes áreas y subáreas. Esta aproximación puede ser de ayuda a los docentes de las asignaturas que tomen este libro como bibliografía de referencia.

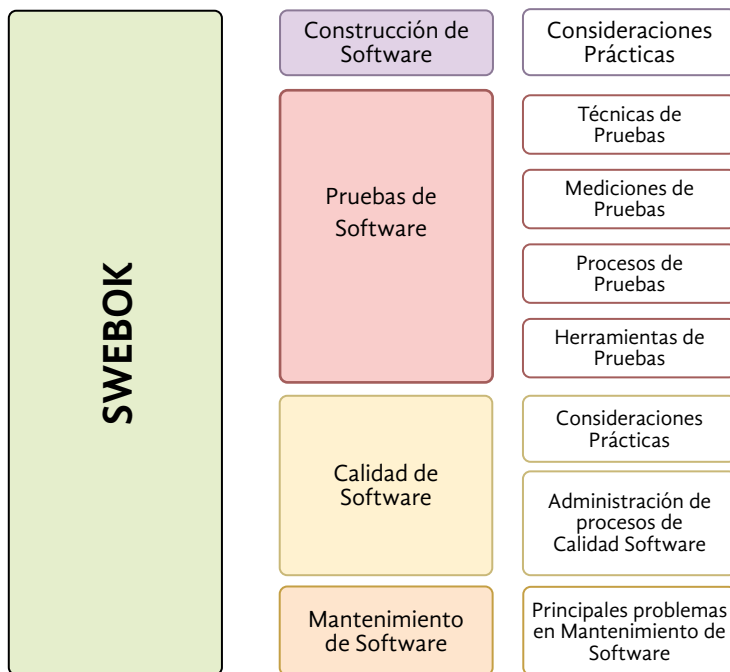


Figura 3. Áreas y subáreas de Swebok tomadas como referencia.

Capítulo 2. Las pruebas y la calidad del software

Las buenas prácticas de pruebas de software como parte de la integración y el despliegue continuo tienen el objetivo principal de mantener una calidad mínima en la construcción de la aplicación. Este capítulo brinda las bases para enmarcar a las pruebas continuas en la disciplina de la calidad del software.

2.1. LA CALIDAD EN EL DESARROLLO DE SOFTWARE



De acuerdo con Parnas (2001, citado en Hoffman y Weiss, 2001), en la Ingeniería del Software se trabaja mayormente con la «construcción de aplicaciones software multi-versión». Por lo tanto, muchas de las actividades asociadas con una aplicación software provocan revisiones para mejorar la funcionalidad o para corregir errores. Esto mismo aseguraba Lehman (1996) en sus leyes, haciendo referencia al incremento de la complejidad y el crecimiento continuo. En la Tabla 1 se detalla una revisión de las últimas descripciones de las leyes de Lehman sobre la evolución del software. Estas leyes hacen referencia a los sistemas del tipo E, que se describen como los sistemas software cons-truidos para solucionar un problema o implementar una aplicación computacional en el mundo real.

De forma resumida, se puede concluir que las aplicaciones software tienden a crecer, tanto en complejidad como en funcionalidad, y es necesario ejecutar pruebas de software para, por lo menos, conservar su calidad actual. Esto se refleja especialmente en la séptima ley de Lehman: «disminución de la calidad».

En este sentido, la calidad es un concepto complejo y multidimensional (Kitchenham y Pfleeger, 1996), es decir, puede ser descrita desde diferentes perspectivas. David Garvin (1987) ha hecho un intento por ordenar las diferentes perspectivas de calidad:

- La perspectiva trascendental: donde la calidad es reconocida, pero no descrita. Se realizan definiciones subjetivas y no cuantificables.

- La perspectiva del usuario: la calidad está directamente relacionada con la satisfacción de las necesidades del usuario. Características como la fiabilidad, el rendimiento o la eficiencia son tenidas en cuenta en esta perspectiva.
- La perspectiva de la fabricación o del proceso: se centra en la conformidad con las especificaciones y la capacidad para producir software de acuerdo con el proceso de desarrollo implementado en la empresa. En este caso, se tienen en cuenta características como la tasa de defectos o los costes de retrabajo.
- La perspectiva de producto: especifica que las características de calidad del producto se definen a partir de las características de sus partes. Por ejemplo, líneas de código, complejidad algorítmica, características en el diseño.
- La perspectiva basada en aspectos económicos: miden la calidad de acuerdo a los costos, precios, productividad, etc.

En el caso del desarrollo software, la calidad puede estudiarse desde el punto de vista de la calidad del producto software y la calidad del proceso de desarrollo software (Robson y McCartan, 2016).

Tabla 1. Leyes de Lehman, formulación de 1996

| AÑO | NOMBRE Y EXPLICACIÓN |
|-----------------------|---|
| 1974 | Cambio continuo. Un software de tipo E que sea utilizado deberá ser adaptado continuamente, sino se volverá progresivamente menos satisfactorio. |
| 1974 | Incremento de la complejidad. La evolución de un programa conlleva un incremento en su complejidad, si no se realiza un trabajo para mantener o reducir este nivel de complejidad. |
| 1974 | Autorregulación. El proceso de evolución del software se encuentra autorregulado con una distribución semejante a la distribución normal que presentan las mediciones de atributos de procesos y producto. |
| 1978 | Conservación de la inestabilidad organizacional (ratio de trabajo invariante). El promedio del ratio de trabajo global efectivo en un sistema que evoluciona es invariante respecto a su tiempo de vida. |
| 1991 | Conservación de la familiaridad. Durante el tiempo de vida activo de un sistema que evoluciona, el contenido de versiones sucesivas es estadísticamente invariante. |
| 1991 | Crecimiento continuo. El contenido funcional de un programa debe ser continuamente incrementado para mantener la satisfacción del usuario. |
| 1996 | Disminución de la calidad. La ausencia de un mantenimiento riguroso y una respuesta adaptativa a los cambios en el entorno operacional se percibirá como una disminución de la calidad del software de tipo E. |
| 1971 revisado en 1996 | Sistema de realimentación. Los desarrollos software de tipo E constituyen sistemas de realimentación multibucle y multinivel, y deben ser tratados de esa manera para asegurar modificaciones y mejoras exitosas. |

Fuente: Lehman (1996).

2.2. MODELOS PARA LA MEDICIÓN DE LA CALIDAD

Jan Bosch define a los atributos de calidad orientados al desarrollo como aquellos de particular relevancia desde la perspectiva de la Ingeniería del Software (Bosch, 2000) y pone como ejemplos a la mantenibilidad y la reutilización, entre otros. A continuación, se comentarán los modelos de calidad de software más comunes que hagan referencia a los atributos de calidad orientados al desarrollo y su perspectiva desde el punto de vista de las pruebas.

Pfleeger y Atlee (1998) hacen referencia a los modelos de calidad de McCall, Boehm y la normativa ISO/IEC 9126 como los más importantes. A su vez, la norma ISO/IEC 25010, publicada en marzo de 2011 y reemplazada por la norma ISO/IEC 9126, puede ser tenida en cuenta en este grupo de modelos.

El modelo de McCall fue creado en 1977 y consiste en un árbol jerárquico donde los factores externos de calidad se relacionan con los criterios de calidad del producto. Los tres factores externos tenían que ver con la operación, revisión y transición de los productos de software. En particular, una de las subcaracterísticas de la revisión es la facilidad con la que se prueba el software, tanto desde el punto de vista del código fuente como de la aplicación final.

Un año después, el modelo de Boehm fue publicado, en 1978, y difiere del modelo de McCall ya que agrega características de rendimiento del hardware. Asimismo, el modelo de Boehm categoriza a los atributos de acuerdo con una perspectiva de utilidad. La norma ISO/IEC 9126 fue publicada por primera vez en 1991, con la intención de unificar los criterios de los modelos anteriores. Este modelo, al igual que los de McCall y Boehm, es jerárquico y conecta a los atributos de calidad a las características. En este caso, la relación de herencia es estricta, pues separa las características y las subcaracterísticas, por lo que una subcaracterística no se relaciona con más de una característica.

Finalmente, dos son los modelos más destacados en el área de la calidad del producto software. Estos se corresponden con la norma ISO/IEC 9126 y su sucesora, la norma ISO/IEC 25010. A continuación, se detallan ambas normas.

2.3. LA NORMA ISO/IEC 9126

La norma ISO/IEC 9126 establece un modelo de calidad en el que se recogen las investigaciones de otros modelos propuestos por los investigadores durante los últimos 30 años para la caracterización de la calidad del producto software.

Este estándar propone tres aproximaciones o enfoques de calidad:

- Interna: la calidad del código fuente mientras se está desarrollando o manteniendo. Esta es especialmente tenida en cuenta por el equipo de programadores y en directa relación con los reportes y herramientas del entorno de desarrollo.
- Externa: calidad de la aplicación final desarrollada. En este caso se pueden tener dos usuarios principales del enfoque; por un lado, el desarrollador y, por otro, el supervisor o jefe de proyectos.

- En uso: las características relacionadas con los diferentes contextos de uso del sistema en explotación. En este caso, la cantidad de usuarios diferentes a la vez, la rapidez de la respuesta o incluso los aspectos ambientales deben ser tenidos en cuenta.

Estas tres aproximaciones se influyen entre sí, tal y como se puede ver en la Figura 4. Así, la calidad interna influye a la externa y esta, a la calidad en uso.

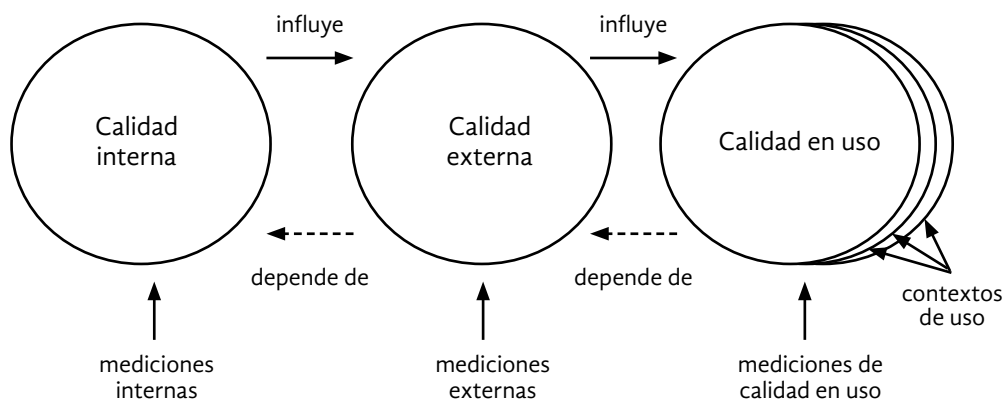


Figura 4. Las tres perspectivas de calidad de la norma ISO/IEC 9126.

En este caso, el modelo establece diez características, seis de ellas comunes a las vistas interna y externa, y las otras cuatro, propias de la vista en uso. Las características que definen las vistas interna y externa se muestran en la Figura 5 y son:

- **Funcionalidad:** capacidad del software de proveer los servicios necesarios para cumplir con los requisitos funcionales. Las pruebas funcionales son las encargadas de medir el grado en el que se ha alcanzado la funcionalidad. Como se verá más adelante, puede ser difícil encontrar mecanismos automáticos para hacerlo.
- **Fiabilidad:** capacidad del software de mantener las prestaciones requeridas del sistema, durante un tiempo establecido y bajo un conjunto de condiciones definidas. Este es un atributo típico a ser medido por las pruebas no funcionales y su eficacia dependerá del grado con el que se pueda cuantificar la «fiabilidad» requerida. Las pruebas de carga en particular acompañan a este tipo de predicciones.
- **Eficiencia:** relación entre las prestaciones del software y los requisitos necesarios para su utilización.
- **Usabilidad:** esfuerzo requerido por el usuario para utilizar el producto satisfactoriamente. En este caso, las pruebas de usabilidad o de interfaz de usuario pueden ser responsables de aumentar la usabilidad de un desarrollo.
- **Mantenibilidad:** esfuerzo necesario para adaptarse a las nuevas especificaciones y requisitos del software. Este atributo puede ser medido de forma indirecta mediante

el análisis estático, las inspecciones de código fuente o la cobertura de las pruebas unitarias.

- **Portabilidad:** capacidad del software para ser transferido de un entorno a otro. Este es otro atributo típico a ser medido por las pruebas no funcionales y también tenido en cuenta para la automatización del despliegue continuo del software.

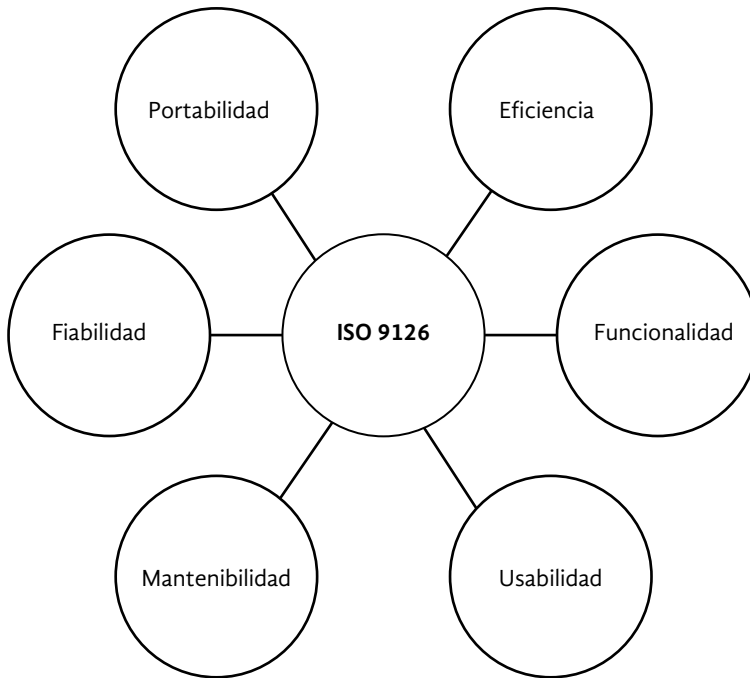


Figura 5. Características de calidad según la norma ISO/IEC 9126.

Por otro lado, las cuatro características propias de la vista en uso se muestran en la Figura 6 y son las siguientes:

- **Efectividad:** capacidad del software de facilitar al usuario alcanzar objetivos con precisión y completitud.
- **Productividad:** capacidad del software de permitir a los usuarios gastar la cantidad apropiada de recursos con relación a la efectividad obtenida.
- **Seguridad:** capacidad del software para cumplir con los niveles de riesgo permitidos tanto para posibles daños físicos como para posibles riesgos de datos.
- **Satisfacción:** capacidad del software de cumplir con las expectativas de los usuarios en un contexto determinado.

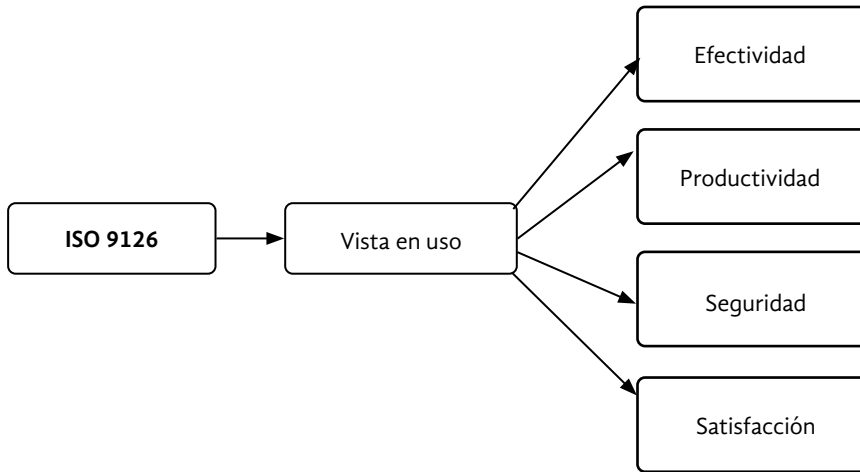


Figura 6. Características de la vista en uso según la norma ISO/IEC 9126.

Aquí, las pruebas están directamente relacionadas con un plan y ejecuciones manuales de las mismas por parte de los usuarios finales. Por ello, son los mecanismos de planificación y reporte los que deben ser tenidos en cuenta.

2.4. LA NORMA ISO/IEC 25010

Desde 2011 existe una nueva versión de la norma ISO/IEC 9126, que reemplaza a la anterior, aunque esta sigue siendo una referencia para los textos científicos. La norma actual es la ISO/IEC 25010, que se encuentra englobada en la serie de normas ISO/IEC 25000.

El objetivo de esta serie es guiar el desarrollo de productos software de acuerdo con una especificación y evaluación de requisitos de calidad. Asimismo, establece criterios para la especificación de requisitos de calidad de productos software, sus métricas y su evaluación. Se definen las siguientes divisiones de normas dentro de la serie 25000:

- ISO/IEC 2500n. Las normas que forman esta división definen todos los modelos comunes, términos y referencias a los que se alude en las demás divisiones de la familia ISO/IEC 25000.
- ISO/IEC 2501n. División del modelo de calidad. Las normas que conforman esta división presentan un modelo de calidad detallado, incluyendo características para la calidad interna, externa y en uso.
- ISO/IEC 2502n. División de mediciones de calidad. Las normas pertenecientes a esta división incluyen un modelo de referencia de calidad del producto software, definiciones matemáticas para las mediciones de calidad y una guía práctica para su aplicación. Presenta aplicaciones de métricas para la calidad externa, interna y en uso del producto software.

- ISO/IEC 2503n. División de requisitos de calidad. Las normas que forman esta división ayudan a especificar los requisitos de calidad. Estos requisitos pueden ser usados en el proceso de especificación de requisitos de calidad para un producto software que va a ser desarrollado o como entrada para un proceso de evaluación.
- ISO/IEC 2504n. División de evaluación de la calidad. Estas normas proporcionan requisitos, recomendaciones y guías para la evaluación de un producto software.

La serie 25000 no establece los niveles de calidad deseables para cada proyecto, pero sí recomienda que los requisitos de calidad deben ser proporcionales a las necesidades de la aplicación y a lo crítico que sea el correcto funcionamiento del sistema implementado. Por lo tanto, será trabajo del jefe del proyecto determinar el nivel de calidad final que un producto software deberá alcanzar y, por lo tanto, el conjunto de pruebas asociadas.

En particular, el principal objetivo de la norma ISO/IEC 25010 es proporcionar un marco para definir y evaluar la calidad del software. Para lograr esto, la norma define dos modelos. El primero es el modelo de la calidad del producto que proporciona un conjunto de características relacionadas con las propiedades estáticas y las propiedades dinámicas del software. Para ello, identifica ocho características que componen los modelos para evaluar la calidad del producto software: idoneidad funcional, eficiencia en el rendimiento, compatibilidad, usabilidad, confiabilidad, seguridad, mantenibilidad y portabilidad.

Este nuevo modelo agrega seguridad y compatibilidad a la norma ISO/IEC 9126, pero ambas deben ser tenidas en cuenta por las pruebas no funcionales.

Para concluir esta sección, en los diferentes modelos de calidad del software se tienen en cuenta características que pueden ser evaluadas y mejoradas con las pruebas de software.

2.5. MODELOS DE PROCESOS ORIENTADOS A LAS PRUEBAS

La calidad de los productos software está fuertemente influenciada por la calidad del proceso que los generó (Humphrey, Kitson y Kasse, 1989). En efecto, los procesos de pruebas contribuyen a la calidad del producto y representan un esfuerzo significativo en proyectos de desarrollo de software.

La mejora de procesos se ha vuelto cada vez más importante a lo largo de los años, con muchas organizaciones tratando de reducir sus costos de producción, mejorando la eficiencia de sus procesos de desarrollo. Esta comprensión está marcada por una progresión de modelos de procesos de software. De todos ellos, el CMMI se considera el estándar de la industria más usado para la mejora de procesos de software (CMMI Production Team SEI, 2010).

Del mismo modo, diferentes modelos de procesos orientados a las pruebas se han desarrollado para gestionar la etapa de pruebas en las organizaciones que construyen software. En este sentido, García, Dávila y Pessoa (2014) llevaron a cabo una revisión sistemática de la literatura que estudia los modelos de procesos de pruebas existentes en la actualidad, siendo los dos más utilizados el TMMi y el TPI.

TMMi es un modelo desarrollado por el TMMi Foundation¹, evolución de TMM, que fue previamente creado por el Instituto Tecnológico de Illinois en 1996. Ambos modelos están basados en el CMMI y también definen cinco niveles de madurez específicos para las pruebas de software. Por otro lado, el TPI es un modelo de pruebas creado por la compañía española Sogeti², orientado a la capacidad de los procedimientos y no a la madurez general de la empresa o el equipo de trabajo.

Teniendo el marco de los modelos de calidad y los modelos de prueba, ahora introduciremos los conceptos relacionados con la forma de llevar adelante las pruebas en el contexto de un proyecto de desarrollo de software moderno.

2.6. METODOLOGÍAS ÁGILES

En febrero de 2001, tras una reunión celebrada en la ciudad de Utah, Estados Unidos, nació el término «ágil» aplicado al desarrollo de software. De esta reunión participó un grupo de 17 expertos de la industria del software, incluyendo algunos de los creadores e impulsores de metodologías de software anteriores. Su objetivo fue esbozar los valores y principios que deberían permitir a los equipos desarrollar software rápidamente y responder a los cambios que puedan surgir a lo largo del proyecto. Se pretendía ofrecer una alternativa a los procesos de desarrollo de software tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada uno de los pasos previos.

Luego de esta reunión, se creó la denominada Alianza Ágil, una organización sin fines de lucro dedicada a promover los conceptos relacionados con el desarrollo ágil de software y que ayuda a las organizaciones a adoptar dichos enfoques. El punto de partida fue un documento que resumía esta filosofía y que se llamó el Manifiesto ágil (Beck et al., 2001).

Finalmente, las metodologías ágiles trajeron consigo un conjunto de buenas prácticas y enfoques para mejorar los procesos de desarrollo (Cohn, 2009). Entre ellos se encuentran la integración continua de los componentes del software, la programación en pares, la documentación mediante historias de usuario y la importancia de los equipos multidisciplinarios, entre otros.

2.6.1. El Manifiesto ágil

Los valores definidos en el Manifiesto ágil no se centran en prácticas, metodologías o procedimientos de trabajo, sino que persiguen un cambio de cultura organizacional basada en estos cuatro pilares:

El individuo y las interacciones del equipo de desarrollo antes que el proceso y las herramientas. Las personas son el principal factor de éxito en un proyecto software. Es más importante construir un buen equipo que construir el entorno. Muchas veces se comete el error de construir primero el entorno y esperar que el equipo se adapte auto-

1. Véase con más detalle en <http://www.tmmifoundation.org/>

2. Véase con más detalle en <https://www.sogeti.es/>

máticamente. Es mejor empezar por el equipo y que este configure su propio entorno de desarrollo y de pruebas sobre la base de sus necesidades.

El software funcional con la documentación necesaria. La regla a seguir es: no producir documentos a menos que sean necesarios de forma inmediata para tomar una decisión importante. Desde este punto de vista, las pruebas son una forma de asegurar la funcionalidad del software.

La colaboración con el cliente en lugar de la negociación basada en un contrato. Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. Esta colaboración entre ambos será la que marque la marcha del proyecto y asegure su éxito. La construcción de la comunicación se realiza también con los resultados parciales del desarrollo, debidamente probados.

La respuesta a los cambios en lugar de seguir estrictamente un plan. La habilidad de responder a los cambios que puedan surgir a lo largo del proyecto determina también el éxito o fracaso del mismo. Por lo tanto, la planificación no debe ser estricta, sino flexible y abierta. El software es construido de manera iterativa e incremental, incluyendo pruebas en todo momento.

Los valores mencionados anteriormente son la base de los doce principios del manifiesto. Estos principios son características que diferencian un proceso ágil de uno tradicional:

1. La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporten valor.
2. Dar la bienvenida a los cambios. Se capturan los cambios para que el cliente tenga una ventaja competitiva.
3. Entregar frecuentemente software que funcione desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.
4. El cliente y los desarrolladores deben trabajar juntos a lo largo del proyecto.
5. Construir el proyecto junto a individuos motivados. Darles el entorno y el apoyo que necesitan y confiar en ellos para conseguir finalizar el trabajo.
6. El diálogo cara a cara es el método más eficiente y efectivo para comunicar información dentro de un equipo de desarrollo.
7. El software que funciona es la medida principal de progreso.
8. Los procesos ágiles promueven un desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz constante.
9. La atención continua a la calidad técnica y al buen diseño mejora la agilidad.
10. La simplicidad es esencial.
11. Las mejores arquitecturas, requisitos y diseños surgen de los equipos organizados por sí mismos.
12. En intervalos regulares, el equipo reflexiona respecto a cómo llegar a ser más efectivo y, según esto, ajusta su comportamiento.

2.6.2. Comparación con las metodologías tradicionales

La Tabla 2 recoge esquemáticamente las principales diferencias de las metodologías ágiles con respecto a las tradicionales o «no ágiles». Estas diferencias afectan no solo al proceso en sí, sino también al contexto del equipo y a su organización.

Tabla 2. Diferencias entre metodologías ágiles y no ágiles

| METODOLOGÍAS ÁGILES | METODOLOGÍAS TRADICIONALES |
|---|---|
| Basadas en heurísticas provenientes de prácticas de producción de código. | Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo. |
| Especialmente preparados para cambios durante el proyecto. | Cierta resistencia a los cambios. |
| Impuestas internamente (por el equipo). | Impuestas externamente. |
| Proceso menos controlado, con pocos principios. | Proceso mucho más controlado, con numerosas políticas/normas. |
| No existe un contrato tradicional, sino un contrato ágil no prefijado y flexible. | Existe un contrato prefijado. |
| El cliente es parte del equipo de desarrollo. | El cliente interactúa con el equipo de desarrollo mediante reuniones. |
| Grupos pequeños (menor a 10 integrantes) y trabajando en el mismo sitio. | Grupos grandes y posiblemente distribuidos. |
| Pocos roles. | Más roles. |
| Menos énfasis en la arquitectura del software. | La arquitectura del software es esencial y se expresa mediante modelos. |



Finalmente, la forma de trabajo propuesta por las metodologías ágiles se materializa en una programación ágil donde cobran vital importancia las pruebas a lo largo de todo el desarrollo. El software es construido mediante iteraciones e incrementos rápidos, que deben ser integrados de manera continua. Esta integración implica también el desarrollo de pruebas rápidas a lo largo de todo el proceso de desarrollo. En el siguiente capítulo profundizamos en la integración continua del software.

Capítulo 3. Integración continua de software

La actividad de integrar las diferentes unidades de construcción o componentes del software está presente tanto en pequeños proyectos a cargo de una sola persona como en un equipo experimentado trabajando en un gran proyecto. En este sentido, la complejidad de la integración aumenta junto con la cantidad de personas del equipo, la cantidad de dependencias y la relación con otros sistemas relacionados. En procesos tradicionales, la integración se realiza al finalizar el proyecto; sin embargo, esperar hasta el final del proyecto para realizar la integración puede generar problemas en términos de calidad, los que a su vez son costosos y producen demoras en los tiempos de entrega (Chow y Cao, 2008). La «integración continua» es una de las prácticas utilizadas para resolver estos inconvenientes.

El término «integración continua» fue mencionado por primera vez en el libro de Grady Booch (1995: 75). En el mismo, Booch expresa que «el proceso macro del desarrollo orientado a objetos, consiste en un proceso de integración continua». Además, menciona que, en intervalos regulares de tiempo, el proceso de integración continua produce nuevas versiones del software ejecutables de funcionalidad creciente. Finalmente, Booch explica que, gracias a estos hitos, se puede medir el progreso y la calidad y, de esta manera, anticipar, identificar y luego atacar los riesgos de forma continua.

De acuerdo a Paul Duvall (2007), la integración continua es simplemente un avance en la evolución del proceso de integración del software. Al principio, la práctica de ejecutar construcciones de código nocturnas automatizadas (*nightly builds*) fue considerada como la mejor solución durante muchos años. Sin embargo, otras prácticas similares han sido propuestas en diferentes libros y artículos. En el libro «Microsoft Secrets» (Cusumano y Selby, 1988) se menciona la práctica de ejecutar construcciones diarias (*daily builds*) en Microsoft. Asimismo, Steve McConnell (1997) recomienda la práctica de ejecutar construcciones diarias y pruebas de humo como parte de los proyectos de desarrollo de software.

Con la venida de las metodologías ágiles, los equipos de desarrollo comenzaron a notar que la construcción no debería ser diaria, sino «continua». Es decir, evitar los procesos secuenciales de desarrollo y prueba al final de día, al final de la semana o al final de mes.



Finalmente, Martin Fowler describe a la integración continua de la siguiente manera (Fowler y Foemmel, 2006: 2):

La integración continua es una práctica del desarrollo de software, donde miembros de un equipo integran su trabajo con frecuencia. Usualmente, cada persona realiza al menos una integración en el día, lo que conduce a múltiples integraciones diarias. Cada integración es verificada por un proceso de construcción¹ automatizado (que incluye pruebas), para detectar errores de integración tan pronto como sea posible.

3.1. EL PROCESO DE INTEGRACIÓN CONTINUA

Un escenario de integración continua comienza cuando un desarrollador realiza un *commit*² de sus cambios hacia el repositorio donde se encuentra el código fuente. Sin embargo, este *commit* no solo se limita al código fuente del programa principal, también abarca cambios en esquemas de bases de datos, archivos de configuración o documentación. La Figura 7 muestra un escenario completo de integración continua y sus componentes.

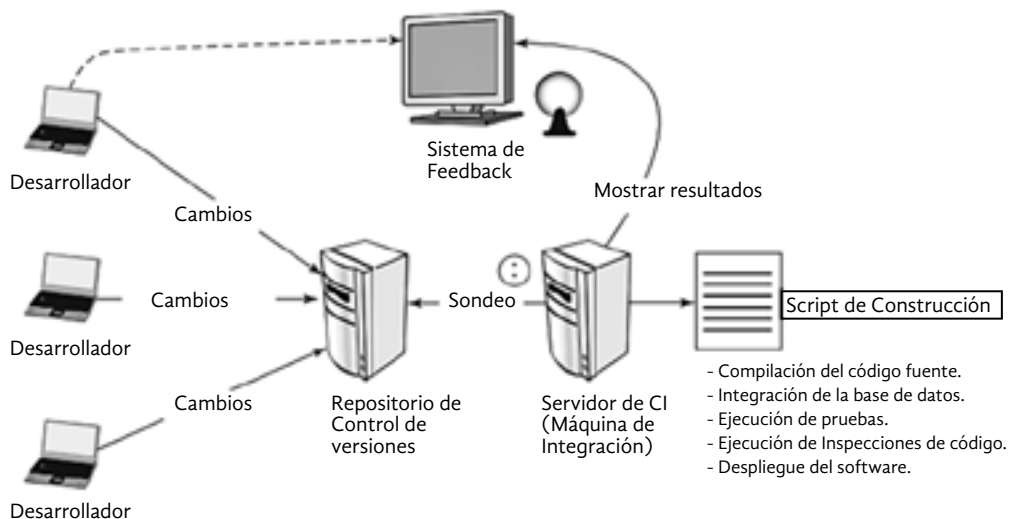


Figura 7. Escenario de una integración continua (Duvall, Matyas y Glover, 2007).

1. Más conocido en inglés como *build*.

2. El *commit* es el proceso para confirmar un conjunto de cambios de forma permanente desde el entorno de trabajo del desarrollo hacia un repositorio común, donde se encuentra el código fuente del proyecto.

Normalmente, los pasos en el proceso de integración continua siguen el siguiente flujo (Duvall, Matyas y Glover, 2007):

1. Un desarrollador introduce un cambio en el repositorio de control de versiones. Mientras tanto, el servidor de integración continua en la máquina de integración se encuentra sondeando este repositorio para detectar nuevos cambios (por ejemplo, cada 15 segundos).
2. Cuando el servidor de integración continua detecta los cambios introducidos en el repositorio de control de versiones, los obtiene y ejecuta un script de construcción (*build script*) que los integra al software.
3. El servidor de integración continua reporta el resultado de la construcción mediante el envío de un correo electrónico a los miembros del proyecto o personas involucradas.
4. Vuelta al punto 1.

A continuación, se describen los actores y componentes involucrados en la Figura 7.

- **Desarrollador:** la persona que modifica el código fuente y realiza el *commit*. Sin embargo, antes de enviar los cambios al repositorio de control de versiones, debe ejecutar una construcción privada (*private build*) en su entorno local (Vafaei y Arvidsson, 2015) para detectar cualquier problema de compilación o defecto antes de la integración. La construcción privada puede realizarse en cualquier momento, sin afectar a los pasos siguientes del proceso de integración continua, ya que el mismo comienza una vez que los cambios son introducidos en el repositorio de control de versiones.
- **Repositorio de control de versiones:** es simplemente un repositorio que tiene integrado un sistema de control de versiones como por ejemplo Git o Subversion.
- **Servidor de integración continua:** ejecuta la integración cuando nuevos cambios son introducidos en el repositorio de control de versiones. Normalmente, está configurado para detectar la introducción de estos cambios cada cierto periodo de tiempo. El servidor obtiene el código fuente y ejecuta los pasos de la construcción a partir de una secuencia de comandos, también llamado script de construcción. Asimismo, estos servidores pueden ser configurados para ejecutar el proceso de integración continua con cierta frecuencia (como por ejemplo cada hora) en lugar de realizarlo con cada cambio introducido, pero esto puede no ser considerado integración continua para algunos autores. Existe una gran variedad de herramientas para operar como servidores de integración continua entre los que se encuentran Jenkins, Bamboo, CruiseControl o TravisCI.
- **Script de construcción:** puede ser uno o varios scripts, que se utilizan para compilar, probar, inspeccionar y desplegar el software. Maven, Ant, NAnt, MSBuild y Rake son ejemplos de herramientas que sirven para crear un script de construcción, pero no proveen integración continua por sí mismas. Algunos desarrolladores utilizan simplemente su entorno de desarrollo para construir el software (Wiegand *et al.*, 2004); sin embargo, de acuerdo a Duvall (2007), esto solo es considerado integración continua siempre y cuando se pueda ejecutar la misma construcción sin usar el entorno de desarrollo.

- Sistema de *feedback*: uno de los principales propósitos de la integración continua es producir información de control al realizar las integraciones, ya que esto logra identificar la introducción de errores con el último *commit*. El sistema de *feedback* permite enviar mensajes a los responsables de la construcción, con los resultados de la misma, tan pronto como finalice el proceso para responder inmediatamente a los problemas de integración del código fuente.

El proceso de integración continua, posterior a la recepción del código fuente, abarca una serie de etapas (Fowler y Foemmel, 2006): compilación del código fuente, integración de la base de datos, ejecución de las pruebas, ejecución de inspecciones y despliegue del software. Este proceso es puesto en marcha con cada *commit* y sus etapas son ejecutadas como parte del *script* de construcción. Es por ello que el *script* de construcción tiene que ser capaz de ejecutarse con solo una línea de comando. Paul Duvall (2007) resume esto imaginando un «botón de integración» (ver Figura 8).



Figura 8. Visualización del botón de integración (Duvall, Matyas y Glover, 2007).

La compilación del código fuente es una de las etapas más comunes en un proceso de integración continua, incluye la creación del código ejecutable a partir del código fuente y, además, es el primer paso. Luego de que el código fuente es compilado, se realiza la integración de la base de datos. La base de datos es una parte de toda la aplicación y su integración debe ser parte del proceso de integración continua. Ejemplos de scripts de integración de base de datos son: scripts de eliminación y creación de bases de datos o tablas, scripts para la inserción de datos de pruebas y scripts para aplicar procedimientos o disparadores (*triggers*).

Posteriormente, se ejecuta una serie de pruebas unitarias. El objetivo es probar pequeños componentes del código para asegurar que la integración se haya realizado con éxito. En paralelo, antes o luego de la ejecución de estas pruebas, se realiza el análisis estático del código fuente para identificar problemas relacionados a la mantenibilidad del código como, por ejemplo, su complejidad, acoplamiento, cohesión o dependencia (Irrazábal y Garzás, 2010).

Una vez que los cambios introducidos pasaron por todas las etapas mencionadas, se lleva a cabo un despliegue del software a un ambiente formal de pruebas a disposición de un equipo especializado en pruebas manuales o pruebas automatizadas de aceptación.

Este flujo puede verse en la Figura 9, en la que las primeras tres etapas deben ser tareas automáticas, pero tanto el despliegue como la ejecución de pruebas siguientes pueden ser manuales. Sin embargo, y como se verá en los siguientes capítulos, para implementar despliegue continuo, el despliegue también debe ser automático.

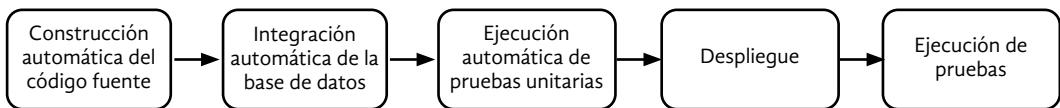


Figura 9. Flujo de pasos de la integración continua.

3.2. INTEGRACIÓN CONTINUA DE BASES DE DATOS

Existe también un proceso para realizar la integración continua en los activos de base de datos. Su objetivo es reconstruir la base de datos y verificar los datos en cualquier momento que se haga un cambio al repositorio de control de versiones, y se lo utiliza para solucionar los siguientes tipos de problemas:

- El código fuente y la base de datos tienen diferentes procesos a lo largo del ciclo de desarrollo de software.
- La base de datos solo puede ser modificada durante días fijos y no en cualquier momento.
- La base de datos solo puede ser modificada por un equipo de infraestructura u operaciones.
- Riesgos elevados de pérdida de información valiosa al realizar cambios en la base de datos.

Este proceso tiene presente que el código de base de datos no es diferente del código fuente de la aplicación o el software que se está construyendo. Por ello:

- Debe residir en un sistema de control de versiones.
- Debe ser verificado e inspeccionado.
- Se puede generar utilizando los mismos scripts de construcción.

Es por esa razón que la construcción de la base de datos puede ser incorporada en el sistema o flujo de integración continua. Inclusive, los cambios en la base de datos también podrían ejecutar el proceso completo de integración continua (Sadlage, 2007), afectando la compilación y verificación del código fuente del sistema.

Las construcciones de la base de datos pueden incluir la creación del esquema de la base de datos desde cero, sobre todo cuando son bases de datos de desarrollo y prueba, así como también pruebas de regresión y análisis estático del contenido de la base de datos.

En muchos proyectos, el administrador de la base de datos concentra una gran cantidad de tareas y puede convertirse en un cuello de botella. Es por ese motivo que las tareas repetitivas pueden automatizarse y así, tanto el tiempo del administrador como el de los miembros de equipo, es utilizado en tareas más complejas. En la Tabla 3 se presentan las tareas repetitivas que pueden ser automatizadas.

Tabla 3. Actividades repetitivas en integración de bases de datos

| ACTIVIDAD | DESCRIPCIÓN |
|--|---|
| Borrado de la base de datos. | Borrar la base de datos y su contenido para crear nuevamente una con el mismo nombre. |
| Creación de la base de datos. | Crear la base de datos utilizando un lenguaje de definición de datos. |
| Inserción de datos de sistema. | Insertar cualquier dato inicial que el sistema requiera para funcionar como, por ejemplo, tablas de <i>lookup</i> . |
| Inserción de datos de prueba. | Insertar datos de pruebas en diferentes instancias de pruebas. |
| Migración de la base de datos. | Migrar el esquema de base de datos y los datos de manera periódica. |
| Configuración de instancias de bases de datos en diferentes ambientes. | Establecer bases de datos separadas para diferentes ambientes (desarrollo, pruebas, etc.) |
| Modificación de restricciones y atributos de columna. | Modificar atributos de columna en la tabla y sus restricciones basados en requerimientos y refactorización. |
| Modificación de datos de prueba. | Modificar los datos de prueba como sea necesario para los diferentes ambientes. |
| Modificación de procedimientos. | Modificar y verificar los procedimientos almacenados todas las veces durante el desarrollo, incluyendo funciones y gatillos de ejecución. |
| Obtención de acceso a diferentes ambientes. | Acceder a los diferentes ambientes de base de datos utilizando un ID, contraseña y un identificador de la base de datos. |
| Respaldo y restauración de grandes volúmenes de datos. | Crear funciones especializadas para grandes volúmenes de datos o para toda la base de datos. |

Fuente: Duvall, Matyas y Glover (2007).

Una vez que las actividades mencionadas en la Tabla 3 se automaticen, será posible la implementación de la integración continua. Sin embargo, los miembros del equipo no deben utilizar estas bases de datos de cada ambiente para realizar tareas de desarrollo. En lugar de ello, es preferente el uso de bases de datos locales para evitar la introducción de problemas en bases de datos que son compartidas por varios usuarios y aplicaciones.

3.3. FEEDBACK CONTINUO

La principal razón por la que una empresa implementa este conjunto de herramientas es porque necesita detectar errores lo antes posible y esto solo se logra con reportes rápidos. Por ejemplo, si se desconoce el resultado de la ejecución de las pruebas de aceptación hasta luego de unas horas de la ejecución, no se pueden tomar acciones inmediatas y solucionar el problema antes de que se propague.

El propósito de la retroalimentación o *feedback* es crear un sistema de notificación que genere una acción lo más rápida y precisa posible. Además, la información tiene que ser la correcta, dirigirse a las personas indicadas, en el momento exacto y a través del medio correcto. Actualmente, los servidores de integración continua permiten hacerlo, en consecuencia, el feedback es continuo. La Figura 10 ilustra este circuito.

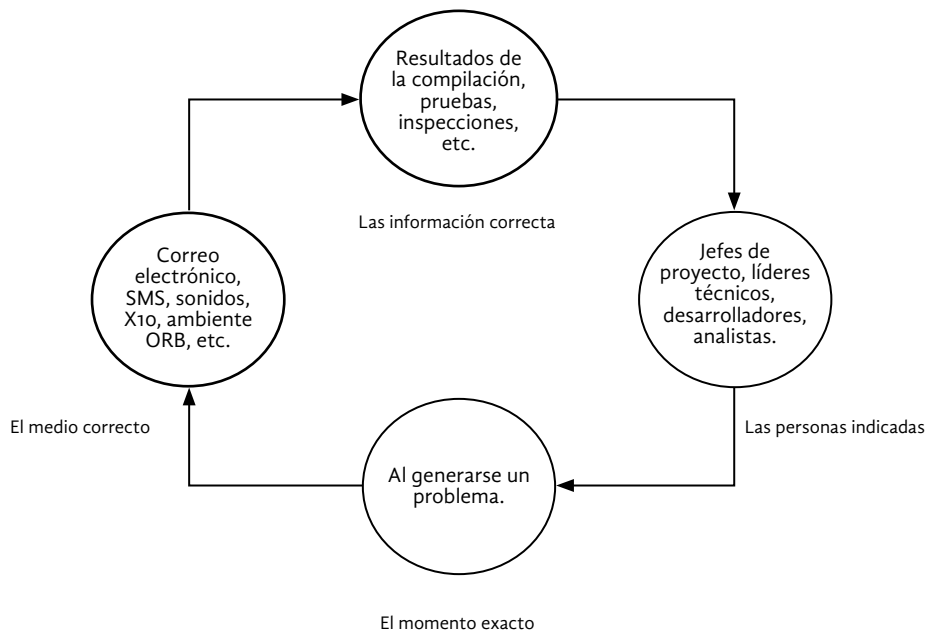


Figura 10. Feedback continuo (Duvall, Matyas y Glover, 2007).

Capítulo 4. Despliegue y entrega continua de software

El producto software debe llegar a los usuarios dentro de un periodo de tiempo establecido, ya sea por el cliente que lo solicitó o por las necesidades de la empresa que lo desarrolla para competir en el mercado. El conjunto de pasos para lograr que un producto software realmente funcione y esté disponible se denomina despliegue. Este ambiente no necesariamente es el entorno final donde los usuarios lo van a operar, sino que puede ser un ambiente de desarrollo o de prueba. Sin el despliegue, se podría concluir, el código fuente no termina de convertirse en un producto software.

La integración continua, como vimos en el capítulo anterior, favorece una mayor frecuencia de despliegues en el ambiente productivo. Por ello, es necesario profesionalizarlo y tener en cuenta diferentes técnicas, especialmente orientadas a su automatización.

La idea de desplegar el software de forma inmediata a producción se viene realizando hace más de 15 años (Coupaye y Estublier, 2000). Sin embargo, utilizar esta práctica como una extensión de la integración continua es una propuesta más reciente. Paul Duvall (2007: 190) define al despliegue continuo como «una culminación de prácticas y pasos que permiten lanzar software funcional en cualquier momento y lugar con el menos esfuerzo posible».

Los principios del despliegue continuo se encuentran en los procesos de desarrollo de software ágil y la filosofía *Lean*, que busca maximizar la eficiencia y minimizar los desperdicios. Como se ha comentado en el capítulo 2, la tendencia actual de las metodologías de desarrollo de software utilizadas en la industria es el desarrollo de software ágil. Desde el punto de vista de la organización, el principio del desarrollo ágil más importante es el siguiente: «la principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software de valor» (Martin, 2002: 7).

Si bien los desarrolladores de software buscan comprender cómo entregar software de trabajo con frecuencia, ha habido poca investigación sobre cómo utilizar metodologías y procesos ágiles para lograrlo (Claps *et al.*, 2015). El proceso de despliegue continuo fue la primera práctica propuesta que busca responder este interrogante, al permitir que los desarrolladores desplieguen el software con frecuencia y, por lo tanto, lo envíen a producción más rápidamente.



Por otro lado, el desarrollo de software Lean es otro de los promotores del despliegue continuo. Poppendieck y Poppendieck (2003) proponen siete principios que forman la base de esta filosofía:

1. Eliminar el desperdicio
2. Amplificar el aprendizaje
3. Decidir lo más tarde posible
4. Entregar lo más rápido posible
5. Empoderar al equipo
6. Construir integridad
7. Ver el todo

El despliegue continuo parece adherirse fuertemente a cuatro de los siete principios de *Lean* (Claps *et al.*, 2015): eliminar el desperdicio, ampliar el aprendizaje, entregar lo más rápido posible y empoderar al equipo.

De esta manera, los principios *Lean* en conjunto con los métodos ágiles ayudan a los equipos de desarrollo a entregar software lo más rápido posible. Esta relación puede verse resumida en la Figura 11. Finalmente, como ejemplo, las grandes empresas de servicios digitales como Amazon, Google o eBay son algunas de las primeras organizaciones que comenzaron a lanzar software funcional a producción con frecuencia.

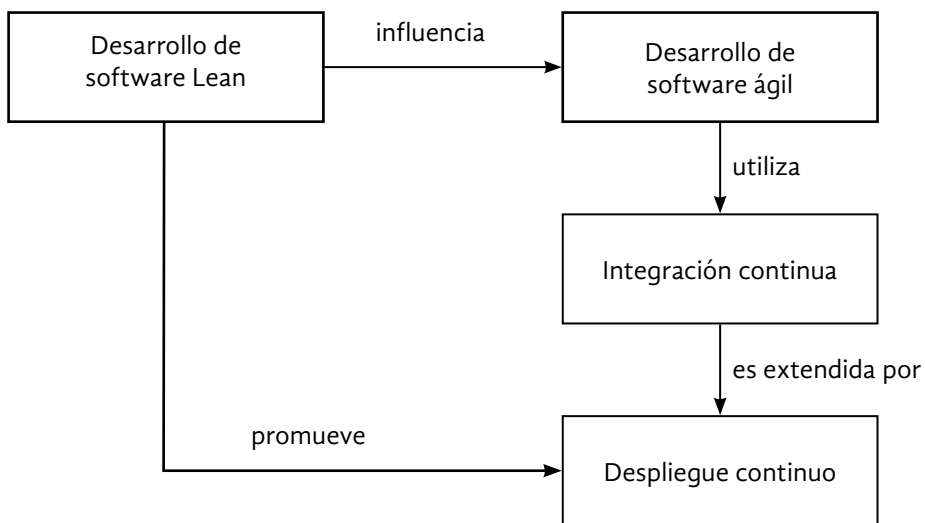


Figura 11. Mapa conceptual del proceso de despliegue continuo (Claps *et al.*, 2015).

4.1. DESPLIEGUE TRADICIONAL Y DESPLIEGUE CONTINUO

La diferencia entre el despliegue continuo y los despliegues de software tradicionales es, principalmente, su frecuencia (Claps *et al.*, 2015). Otras diferencias clave entre estos dos tipos de despliegue se resumen en la Tabla 4.

Tabla 4. Comparación entre despliegue de software tradicional y el despliegue continuo

| CARACTERÍSTICA | DESPLIEGUE TRADICIONAL | DESPLIEGUE CONTINUO |
|--|---|--|
| Frecuencia de los despliegues | Puede ir de 1 a 6 meses, incluso más. La frecuencia ha ido aumentando, de meses a semanas, de acuerdo con la metodología de desarrollo (Feitelson, Frachtenberg y Beck, 2013). | Diaria. Empresas como IMVU despliegan el software a producción hasta 50 veces al día. Otro ejemplo es Facebook, que lo hace una vez al día (Feitelson, Frachtenberg y Beck, 2013). |
| Riesgo al realizar el despliegue | Alto. Dado que el despliegue es poco frecuente y que el software que se va a lanzar contiene muchos cambios, existe una mayor probabilidad de que el mismo contenga defectos (Humble y Farley, 2010). | Como los cambios son pequeños y los despliegues frecuentes, el riesgo es menor (Claps <i>et al.</i> , 2015). |
| Feedback del cliente/ usuarios finales | Se lo obtiene luego de largos periodos de tiempo, que dependen de la frecuencia de los despliegues (Agarwal, 2011). | Se lo obtiene muy rápidamente, ya que los clientes y usuarios perciben los cambios de manera constante (Lacoste, 2009; Van der Storm, 2008). |
| Funciones o características que no se terminan de desarrollar | Según The Standish Group (2013), solo el 69% de las funciones o características que son solicitadas al inicio del proyecto se desarrollan. | Ayuda a disminuir el desarrollo de características innecesarias debido a la obtención de <i>feedback</i> frecuente (Poppendieck y Poppendieck, 2003; Ries, 2011). |

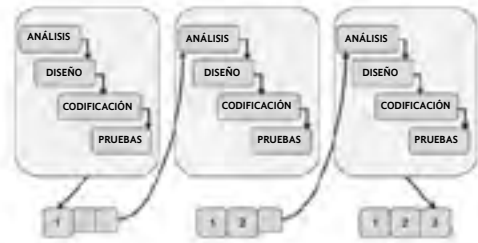
Las empresas evolucionan sus prácticas de desarrollo de software con el correr del tiempo. Normalmente, existe un patrón que la mayoría de ellas siguen en su evolución y que algunos autores denominan la «escalera al cielo».

El primer paso es ir de un enfoque tradicional a uno ágil. En el enfoque tradicional, el cliente tendrá el producto software al final del proceso de desarrollo. Esto puede tardar mucho tiempo y, por lo tanto, se descubren desviaciones entre los requerimientos iniciales y el resultado obtenido.

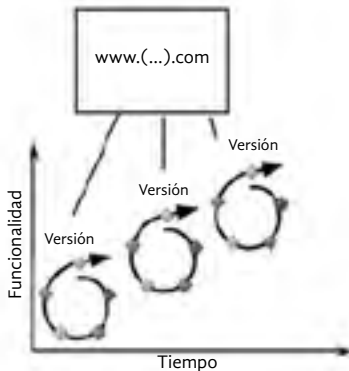
El segundo paso es la institucionalización de las prácticas ágiles, incorporando procesos como la integración continua. En este caso, el desarrollo se ha vuelto iterativo e incremental. Finalmente, una vez que la integración es alcanzada correctamente, el equipo de desarrollo es capaz de entregar pequeños componentes del software al cliente durante las reuniones de presentación o «demos». Es allí donde se ve la oportunidad de tener estos componentes funcionales en producción mediante la adopción del despliegue continuo. Un resumen de lo dicho anteriormente puede verse en la siguiente figura.



Cascada: El cliente tiene el producto al final del proceso de desarrollo.



Iterativo-Incremental sin despliegue continuo: El cliente tiene partes del producto funcionales listas, pero no lo tiene en producción.



Iterativo-Incremental con despliegue continuo: El cliente tiene diferentes versiones del producto en producción al finalizar cada iteración.

Figura 12. Evolución del despliegue continuo con las metodologías de desarrollo de software.

4.2. ETAPAS DEL DESPLIEGUE CONTINUO

El primer requisito para hacer un despliegue frecuente del software es implementar la integración continua, de tal manera que la construcción del software y la ejecución de pruebas en todos los niveles sean tareas automáticas y repetibles. Sin embargo, además de esto, el despliegue de software a cualquier ambiente abarca las siguientes etapas (Duvall, Matyas y Glover, 2007):

1. Etiquetado de los activos que serán desplegados: se realiza para facilitar la identificación y seguimiento de los activos que van a ser desplegados en producción y para agruparlos por una característica similar como, por ejemplo, un componente en una página web. Los activos no solo son archivos del código fuente, sino también esquemas de bases de datos, scripts de pruebas automatizadas, archivos de configuración o elementos multimedia.

2. Generación de un ambiente limpio: un error común es desplegar el software en un ambiente que tiene instalado una versión diferente del sistema operativo, base de datos o servidor de aplicaciones. Es importante generar un ambiente de despliegue limpio para tener todos los parámetros de configuración deseados. El script de generación de ambiente será el encargado de producir las siguientes capas:
 - a. Sistema operativo requerido.
 - b. Configuraciones del sistema operativo, como conectividad de red, usuarios o *firewall*.
 - c. Servicios, como servidor de aplicación, servidor de base de datos, servidor de mensajería, etc.
 - d. Configuración del servidor.
 - e. Herramientas de terceros.
 - f. Software personalizado adicional.
3. Etiquetado de los artefactos binarios: primeramente, el código en el repositorio requiere de una etiqueta. Sin embargo, para diferenciar la versión del software que está funcionando en un ambiente determinado, es necesario etiquetar también el artefacto resultante de la construcción del código en el repositorio, es decir, los binarios. El etiquetado de binarios permite identificar problemas y asociar soluciones a una determinada versión.
4. Instalación del artefacto binario en el servidor correspondiente: una vez que se tiene el artefacto binario etiquetado, se lo mueve al ambiente generado en la etapa 2 y luego se lo instala para que comience a funcionar.
5. Ejecución de todos los tipos de pruebas necesarias y en todos los niveles: mientras ciertas etapas en el desarrollo del software requieren de la ejecución de algunas pruebas determinadas (por ejemplo, las pruebas unitarias al inicio), cuando el software es puesto en marcha en un ambiente, todas las pruebas deben ser ejecutadas. Es fundamental ejecutar previamente pruebas que detecten problemas en el ambiente, antes de ejecutar las pruebas unitarias, de integración y funcionales. Si bien algunas pruebas se ejecutan de forma automática, se requiere una etapa de pruebas manuales ejecutadas por un equipo especializado de pruebas.
6. Creación de reportes de *feedback*: los reportes de *feedback* deben ser generados para ver los resultados del despliegue y fáciles de entender por cualquier integrante del proyecto, incluyendo los roles no técnicos.
7. Si fuera necesario realizar la vuelta atrás utilizando las versiones etiquetadas en el repositorio de control de versiones: se debe tener la capacidad de poner en ejecución la última versión estable del software.

Para que sea un despliegue continuo, todas estas etapas deben ser ejecutadas con tan solo un comando, es decir, deben estar automatizadas. Existen herramientas para realizar despliegues automáticos que se utilizan en combinación con las herramientas de integración continua. También hay herramientas que permiten implementar tanto la integración como el despliegue continuo.

4.3. ENTREGA CONTINUA Y DESPLIEGUE CONTINUO

Cuando en 2001 se escribe el Manifiesto ágil, se sabía que era importante la entrega continua de software, pero algo muy difícil de lograr (Fowler, 2014). De hecho, una de las primeras frases del manifiesto expresa: «nuestra principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software con valor».

La entrega continua es un enfoque de la Ingeniería del Software en el que los equipos construyen software funcional en ciclos muy cortos de tiempo, de tal manera que pueda ser liberado o lanzado a producción en cualquier momento (Chen, 2017).

Una de las características más importantes del despliegue continuo es la confiabilidad al liberar una nueva funcionalidad o cambio en el software (Fowler, 2014). Por ese motivo, cada cambio introducido debe someterse a un proceso riguroso de pruebas. Así, cualquier introducción, modificación o borrado de una funcionalidad es lanzada a producción cuando se lo desee con un riesgo muy bajo de errores.

Las expresiones *despliegue* y *entrega continua* son utilizadas por error como sinónimas (Fowler, 2014). Sin embargo, el despliegue continuo consiste en enviar a producción cualquier cambio que el desarrollador integre al repositorio principal. En cambio, en la entrega continua, la decisión final de ir a producción es una decisión de negocio, no una decisión técnica. La Tabla 5 muestra las principales diferencias y similitudes entre estos dos enfoques.

Tabla 5. Diferencias y similitudes entre despliegue y entrega continua

| DESPLIEGUE CONTINUO | ENTREGA CONTINUA |
|---|--|
| Se enfoca en el despliegue a producción en el tiempo más rápido posible. | Se enfoca en la confiabilidad de introducir cambios a producción. |
| La decisión de lanzar una versión del software a producción no depende del negocio. | El negocio decide el momento en el que una nueva funcionalidad sale a producción. |
| El proceso de despliegue a cualquier ambiente es automatizado. | |
| Ejecución de las pruebas más importantes para acelerar el tiempo del despliegue. | Ejecución de pruebas rigurosas para asegurar que el software no contiene errores. |
| No hay intervención humana en el flujo. | Una persona debe presionar un botón (ejecución de un comando) para hacer el despliegue a producción. |
| Es un complemento de la integración continua. | |

Así, el despliegue continuo y la entrega continua no se pueden hacer al mismo tiempo, aunque sean complementarios a la integración continua. La diferencia es que el despliegue continuo se dispara inmediatamente luego de finalizar una etapa anterior (por ejemplo, la etapa de pruebas). En cambio, en la entrega continua, el despliegue se realizará cuando se presiona el botón de despliegue. Estas diferencias se pueden ver en la descripción de pasos de la Figura 13.

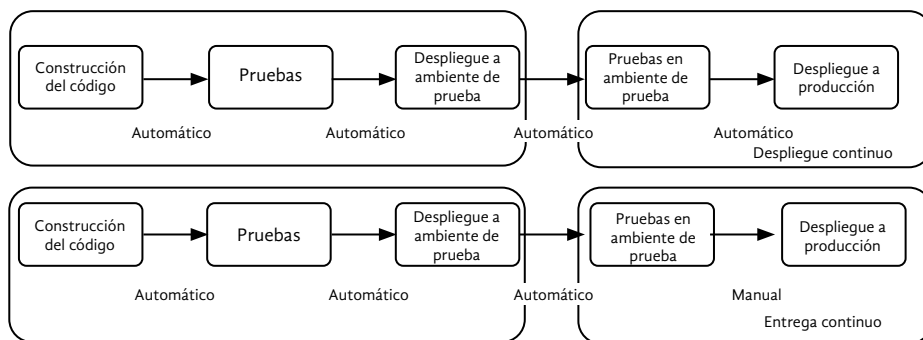


Figura 13. Comparación en los flujos de actividades del despliegue y la entrega continua.

La estrategia de pruebas también es diferente, aunque en la Figura 13 la única diferencia apreciable sea el despliegue a producción. En el despliegue continuo se busca aumentar la velocidad ejecutando solo las pruebas necesarias e involucrando al usuario final también como una etapa de pruebas más. Por otro lado, en la entrega continua, se ejecuta la mayor cantidad de pruebas posibles asegurando que el producto puesto en producción esté libre de errores (Fowler, 2014). Aun así, la velocidad de las pruebas es, en todos los casos, un factor clave para lograr la continuidad en el trabajo.

4.4. COMPONENTES DE ENTREGA CONTINUA

Uno de los principios de la entrega de software continua es la creación de un proceso de despliegue del software a producción repetible y confiable (Humble y Farley, 2010). Este principio se basa en que el lanzamiento de una versión del software a producción debe ser una tarea sencilla porque el mismo ya ha atravesado por diferentes niveles de pruebas. De esto modo, realizarlo «sería tan simple como presionar un botón» (Humble y Farley, 2010: 11). Para lograrlo, es necesario automatizar todas las tareas involucradas posibles y mantener todos los artefactos en el sistema de control de versiones.

Como se ha indicado anteriormente, la mayoría de las tareas son automatizables: la compilación y la construcción del código, los diferentes niveles y tipos de pruebas, la integración de la base de datos, el despliegue a diferentes ambientes, la generación de los ambientes de prueba o las notificaciones (Duvall, Matyas y Glover, 2007). Sin embargo, algunas tareas no pueden ser automatizadas. Entre ellas se encuentran:

- La ejecución de pruebas exploratorias realizada por un equipo de expertos y especialistas en pruebas.
- La demostración del software funcional a los responsables y dueños del producto.
- La aprobación por parte del cliente.

En general y en cualquier tipo de proyecto, la lista de tareas que no se puede automatizar es mucho menor que la lista de las tareas automatizables (Humble y Farley, 2010). La automatización es un requisito indispensable para el proceso de entrega continua, ya que es la única manera de garantizar que las personas puedan tener una versión del producto con solo presionar un botón.

Por otro lado, la gestión de la configuración también es un requisito fundamental. Todo artefacto que se utilice para las tareas principales de construcción, despliegue y prueba debe ser parte del sistema de control de versiones: el código fuente, los documentos de requerimientos, los casos de pruebas, los casos de pruebas automatizados, los scripts de configuración de red, los scripts de despliegue, los scripts de procesos en la base de datos, los archivos de configuración de la aplicación, las librerías y dependencias o la documentación técnica. Los artefactos producto del despliegue de igual modo deben ser versionados (Duvall, Matyas y Glover, 2007; Humble y Farley, 2010), ya que, si se produce un error en el despliegue o en cualquier momento del flujo, la detección del artefacto y la versión que lo causó debe ser simple de identificar (Fowler, 2014). Es por eso que los nombres de los *commits* deben ser claros y representativos de los cambios que se están introduciendo en el repositorio.

Otro factor relacionado con los despliegues del software es la infraestructura. En la mayoría de las organizaciones existen equipos de desarrollo y equipos de infraestructura u operaciones. Si las tareas relacionadas al manejo de infraestructura y ambientes son realizadas por ambos equipos de manera colaborativa desde el inicio del proyecto, serán llevadas a cabo más fácilmente (Humble y Farley, 2010). Este foco en la colaboración es uno de los principios centrales del movimiento DevOps, que tiene como objetivo agilizar los procesos de administración de sistemas y operaciones.

Esta forma de trabajo puede considerarse un cambio organizativo en el que los diferentes roles de desarrollo trabajan en conjunto para lograr entregas continuas de funciones operativas (Virmani, 2015). Este enfoque ayuda a entregar valor de forma más rápida y fluida, como así mismo a reducir los problemas debido a la falta de comunicación entre los miembros de los diferentes equipos y acelerar la resolución de problemas. Inclusive, algunas empresas adoptaron el término DevOps como un nuevo rol en la organización, refiriéndose a los especialistas que realicen estas tareas (Fowler, 2014). Por ello, otro aspecto importante es mejorar la comunicación y colaboración entre los miembros del equipo.

4.5. EL CONDUCTO DE DESPLIEGUE

Como se ha visto anteriormente, el resultado de la integración continua es la entrada a una nueva etapa de pruebas y luego al resto del proceso de despliegue a producción. De este modo, se podría perder tiempo en las etapas finales haciendo que el producto software tarde en llegar a un ambiente de producción. Finalmente, el proceso de *feedback* entre estas etapas, al ser aisladas, no es inmediato. El conducto de despliegue o también llamada la tubería de despliegue es la solución propuesta por Humble y Farley (2010) como un enfoque holístico, de extremo a extremo, para la entrega continua.

La tubería de despliegue es un patrón de entrega continua que consiste en un proceso automatizado para abarcar todas las etapas desde la integración de código en el repositorio de control de versiones hasta el usuario final. Una manera gráfica de presentar la tubería de despliegue es presentada en la Figura 14.

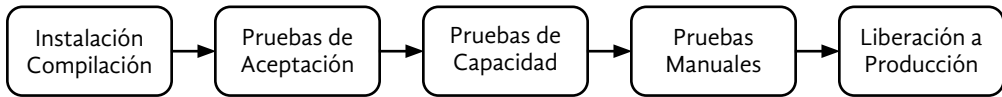


Figura 14. Anatomía de la tubería de despliegue (Humble y Farley, 2010).

La entrada a la tubería de despliegue se hace mediante la integración de código, donde el servidor de integración continua detecta los cambios introducidos en el repositorio de control de versiones. El código se construye y se ejecutan las pruebas unitarias para luego desplegar los cambios a un ambiente de pruebas y allí ejecutar las pruebas de aceptación. Posteriormente, los cambios se vuelven a desplegar en un ambiente con mayor similitud al ambiente de producción y se ejecutan las pruebas exploratorias manuales. Finalmente, el software es lanzado a producción.

Sin embargo, si se produjera un error o fallo en algunas de estas etapas, se genera *feedback* y se detiene el flujo de trabajo. El proceso en su conjunto se presenta en el siguiente diagrama de secuencia (ver Figura 15).

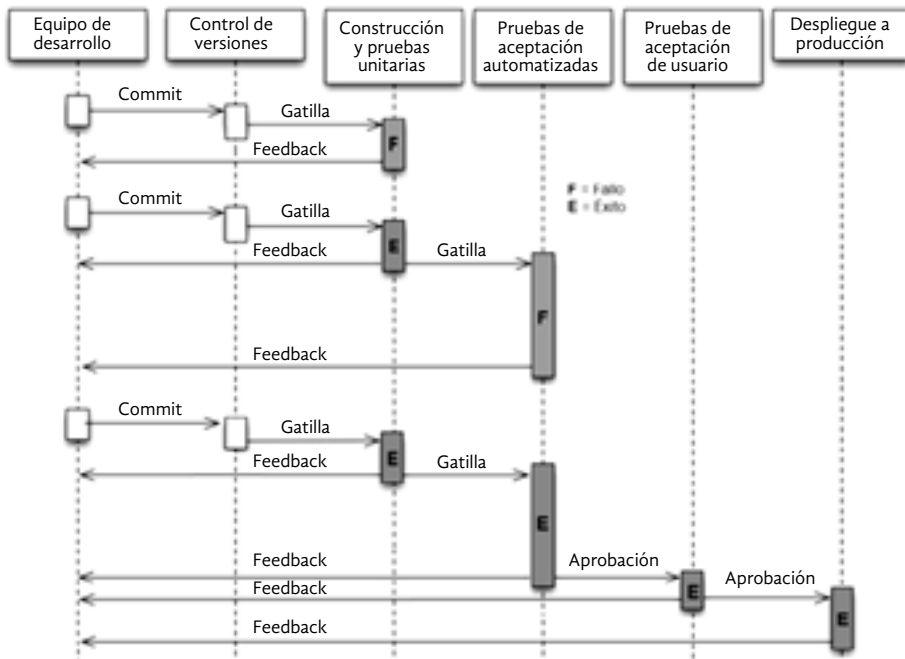


Figura 15. Cambios en el código moviéndose a través de la tubería de despliegue (Humble y Farley, 2010).

Con este patrón se logra un entorno automatizado de compilación y pruebas de código dividido en etapas. No obstante, la implementación de la tubería de despliegue depende de la necesidad de cada organización. Por ejemplo, Paddy Power¹, empresa del rubro de apuestas en línea, la ha implementado como se muestra en la Figura 16.

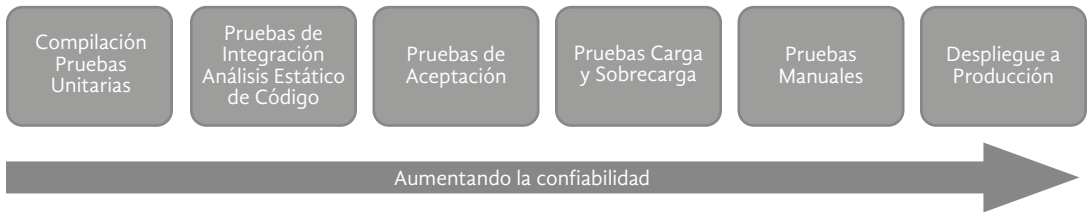


Figura 16. Implementación de la tubería de despliegue en la empresa Paddy Power (Chen, 2017).

La etapa de pruebas es la que más varía en la implementación de una tubería de despliegue. A continuación, se detallan las pruebas recomendables de acuerdo con diferentes autores:

- Pruebas unitarias
- Inspecciones del código estático
- Pruebas funcionales de interfaz gráfica de usuario (GUI) e interfaz de programación de aplicaciones (API)
- Pruebas no funcionales (capacidad, rendimiento, carga y sobrecarga)
- Pruebas exploratorias manuales
- Pruebas de aceptación de usuario.

En el siguiente capítulo se describirán estos tipos de prueba desde el punto de vista de la integración continua y las tuberías de despliegue.

El éxito de adoptar prácticas continuas en las organizaciones depende mucho de la implementación de la tubería de despliegue (Phillips *et al.*, 2014). Por lo tanto, la elección de las herramientas e infraestructuras adecuadas para componer tal proceso también ayuda a mitigar algunos de los desafíos en la adopción e implementación de prácticas, las tres prácticas continuas vistas en estos capítulos. En la Figura 17 se muestran herramientas utilizadas en la implementación de una tubería de despliegue, producto de una revisión sistemática de la literatura (Shahin, Babar y Zhu, 2017).

1. Véase Paddy Power en https://en.wikipedia.org/wiki/Paddy_Power

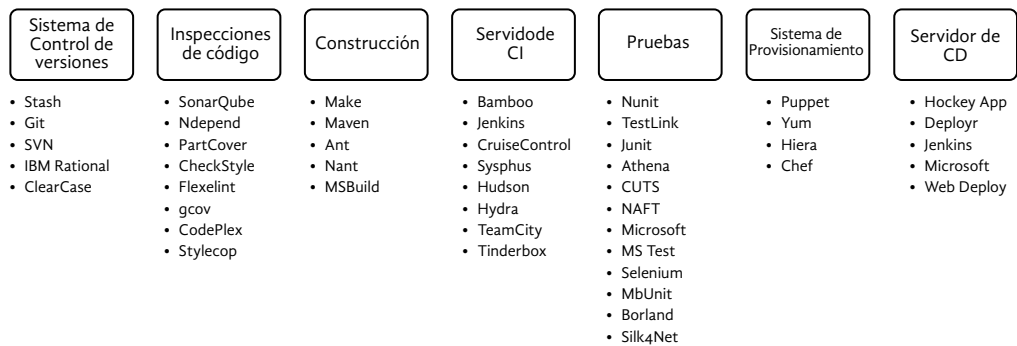


Figura 17. Una visión general de las herramientas utilizadas para implementar la tubería de despliegue.



Capítulo 5. Tipos de pruebas

La inclusión de las diferentes actividades de prueba implica conocer más en detalle sus características. En todos los casos, las pruebas de software buscarán evaluar la calidad del producto desarrollado y mejorar la identificación de los defectos y problemas. En particular, para Swebok, tiene cuatro características principales (Bourque y Fairley, 2014).

- Dinámica: las pruebas se realizan solo sobre el programa en ejecución o sus partes ya construidas.
- Finita: las pruebas exhaustivas llevan mucho tiempo y es por ello que los ciclos de pruebas deben ser reducidos buscando cumplir un objetivo particular.
- Adecuada: existen muchas técnicas de pruebas, por lo que se deben elegir las que satisfagan con eficacia y eficiencia el objetivo de la prueba. Interpretadas: el objetivo de las pruebas es comprobar el cumplimiento de los requisitos y también identificar los errores o los fallos.



El proceso de pruebas inicia en las etapas de planificación de proyecto y concluye en las de mantenimiento. Asimismo, con la llegada de las metodologías ágiles, la verificación de software incluye tipos y niveles de pruebas que no solo se realizan con el software en ejecución, sino también durante su integración y construcción (Crispin y Gregory, 2009; Duvall, Matyas y Glover, 2007; Gregory y Crispin, 2014).

En este marco, las pruebas deben cumplir ciertos requerimientos para ser implementadas en los entornos continuos, siendo uno de ellos la rapidez en sus tiempos de ejecución. Humble y Farley (2010) afirman que se deben automatizar todos los niveles de pruebas posibles, mientras que otros autores realizaron estudios demostrando que la automatización de pruebas es una pieza clave en la entrega continua (Shahin, Babar y Zhu, 2017).

Sin embargo, las pruebas exploratorias manuales asimismo son importantes en la entrega continua y no pueden ser automatizadas, ya que requieren la experiencia de un equipo especialista en pruebas manuales (Chen, 2017; Gregory y Crispin, 2014; Humble y Farley, 2010; Shahin, Babar y Zhu, 2017). De este modo, los autores de estas prácticas proponen niveles y tipos de pruebas como necesarias, las que se describen a continuación.

5.1. INSPECCIONES DE CÓDIGO

Las inspecciones de código, también conocidas como análisis estático del código fuente (Louridas, 2006), son una de las primeras etapas de verificación de calidad en los entornos de desarrollo continuo (Duvall, Matyas y Glover, 2007; Humble y Farley, 2010). La misma es utilizada para medir atributos de calidad en el código fuente, de acuerdo con buenas prácticas de codificación. Por lo general, de estas mediciones se obtienen valores que podrán asociarse al grado de mantenibilidad de este código fuente y se ejecutan justo después de la integración del código fuente al repositorio.

La inspección de código fuente es un tipo de análisis de software que se lleva a cabo sin que el producto software esté en ejecución o, incluso, sin que este pueda ser compilado y por esta razón se lo llama «estático» (Wichmann *et al.*, 1995). En este sentido, en enfoques tradicionales, la inspección de código no es considerada un tipo de prueba, pues se ejecuta antes de que concluya la etapa de desarrollo.

El análisis estático de código fuente se realiza únicamente de manera automática por varios motivos:

- Las inspecciones de código manuales son propensas a errores y no son repetibles con facilidad.
- Las revisiones de código entre pares pueden ser subjetivas.
- Son más rápidas que las manuales.
- Las herramientas de inspecciones de código permiten personalizarse y generar métricas, guardando los resultados de las diferentes ejecuciones también de manera automática.

La forma de medir los atributos de calidad en el código fuente recibe el nombre de «métrica». Algunas de las principales métricas y la evaluación de buenas prácticas obtenidas de diferentes herramientas que han demostrado estar relacionadas con la mantenibilidad son:

- Sentencias de código fuente no comentadas: esta métrica tiene sentido como descripción de qué tan grande puede ser un método, una subrutina, una función o una clase en el código fuente.
- Complejidad del código fuente: existen diferentes formas de medición de este atributo, pero todas intentan medir la cantidad de caminos independientes en un método o función.
- Métricas de diseño orientado a objetos: son un conjunto de métricas que buscan capturar las características de un buen código fuente desarrollado sobre la base del paradigma de orientación a objeto. Hacen hincapié en el tamaño, la complejidad, el acoplamiento y la cohesión de los métodos y clases.
- Código duplicado: en este caso, las herramientas analizan la similitud entre diferentes partes de un código fuente. Este es un ejemplo de cómo se puede medir una mala práctica, o sea, duplicar el código fuente en lugar de escribir un método genérico utilizado varias veces a lo largo del sistema. Desde el punto de vista de las pruebas

continuas, el resultado más importante es el descubrimiento de la duplicidad, más allá de cuantificar la cantidad de código duplicado existente.

- Código no utilizado: en este caso, el análisis, si se considera estático, es más limitado para hallar código que no será alcanzable en ningún momento durante la ejecución del código. Nuevamente, desde el punto de vista de las pruebas continuas, el resultado más importante es el descubrimiento del código fuente que no se ejecutará nunca.
- Validación de estilos: existen herramientas que pueden analizar los patrones de estilo de codificación y compararlos con patrones estándares. Así, por ejemplo, se valida la forma de nombrar las variables, los métodos o la presencia de comentarios en el código fuente. Esto favorece el entendimiento del código, una de las características de calidad relacionadas con la mantenibilidad.
- Dependencias cíclicas: en este caso, las unidades lógicas de cohesión en el código fuente que representan conjuntos de funcionalidades relacionadas deberían tener una relación jerárquica entre ellas y no una relación circular. Esto último disminuye la estabilidad en el código fuente.

5.2. PRUEBAS UNITARIAS

Las pruebas unitarias son verificaciones de pequeñas partes del código, por lo general funciones, métodos, subrutinas o partes de ellas. Según Swebok, las pruebas unitarias verifican el funcionamiento de trozos de software de forma aislada y se pueden probar por separado (Bourque y Fairley, 2014). Dependiendo del contexto, estas piezas de código también podrían ser subprogramas individuales o un componente más grande hecho de unidades estrechamente relacionadas. Es por ello que a veces a estas pruebas se las conoce como pruebas de componentes. Sin embargo, otros autores utilizan el término pruebas de componentes para referirse a las pruebas de integración (Duvall, Matyas y Glover, 2007). Por este motivo, en este texto, no se utilizará el término pruebas de componentes para evitar ambigüedad.

Normalmente, las pruebas unitarias se realizan con el acceso al código que se está probando y con el soporte de herramientas de depuración, involucrando principalmente a los programadores que escribieron el código fuente.

El estándar IEEE 1008-1987 define la etapa de pruebas unitarias de software como un proceso que incluye etapas de planificación, desarrollo, ejecución y medición de resultados. Asimismo, describe un proceso de pruebas compuesto por una jerarquía de fases y actividades, definiendo un conjunto mínimo de tareas para cada actividad (ver Figura 18).

| Fase de planificación de pruebas | Fase de obtención del lote de pruebas | Fase de medición de las pruebas unitarias |
|--|--|--|
| <ul style="list-style-type: none"> Planificación del enfoque, recursos y tiempos. Determinación de las características a ser probadas. Refinamiento del plan general. | <ul style="list-style-type: none"> Diseñar el lote de pruebas. Implementación del plan de pruebas y su diseño. | <ul style="list-style-type: none"> Ejecución de las pruebas. Control de la ejecución. Evaluación de los resultados. |

Figura 18. Actividades del proceso de pruebas unitarias.

No obstante, para que las pruebas unitarias sean parte de un entorno continuo, las mismas deben automatizarse. Es decir, se debe utilizar un trozo de código de un programa para probar un trozo de código de otro programa. Para automatizar las pruebas unitarias, existe en el mercado una gran variedad de tecnologías o librerías, dependiendo de la plataforma y lenguaje de programación del código a probar. En la Tabla 6 se listan las herramientas más conocidas para automatización de pruebas unitarias. En la primera columna se menciona el nombre de la herramienta; en la segunda columna, la tecnología o lenguaje de programación al que está asociada y, finalmente, en la tercera columna se indica el tipo de licencia.

Tabla 6. Herramientas más conocidas para automatización de pruebas unitarias

| HERRAMIENTA | TECNOLOGÍA | LICENCIA |
|-------------|------------|----------------|
| Atoum | PHP | Código abierto |
| C++ test | C/C++ | Comercial |
| CppUnit | C/C++ | LGPL |
| DDCput | C/C++ | Código abierto |
| Enhance PHP | PHP | Comercial |
| Go2xunit | Go | Código abierto |
| Jasmine | JavaScript | Código abierto |
| Jest | Java | Comercial |

| | | |
|----------------------------|------------|----------------|
| JMock | Java | Gratuito |
| Junit | Java | Código abierto |
| MbUnit | .NET | Gratuito |
| Mocha | JavaScript | Código abierto |
| Nunit | .NET | Código abierto |
| Parasoft C/C++ test | C/C++ | Comercial |
| PHPUnit | PHP | Código abierto |
| Pytest | Python | MIT |
| PyUnit | Python | Código abierto |
| QA Systems Cantata | C/C++ | Comercial |
| QUnit | JavaScript | Gratuito |
| SQLUnit | SQL | GLPv2 |
| utMySQL | SQL | GPL |
| Xunit.net | .NET | Código abierto |

Fuente: Rodríguez, Piattini y Ebert (2019).

A modo de ejemplo, se muestra una prueba unitaria escrita con la herramienta Junit para la tecnología Java.

```

public class PHDTest {

    @Test
    public void verifyAdditionTest()
    {
        Calculator calculator = new Calculator()
        int numberOne = 8;
        int numberTwo = 4;
        int expectedResult = 12;
        int actualResult = calculator.add(numberOne, numberTwo);
        Assert.assertTrue(actualResult, expectedResult);
    }
}

```

El caso de prueba presentado verifica que un objeto de tipo «Calculadora» sume dos números. Cada prueba es codificada en un método y, a su vez, las pruebas asociadas a un mismo caso de prueba se encuentran en una misma clase.

Algunas pruebas unitarias requieren mínimas dependencias de otras clases, pero no deben interactuar con entidades externas como sistemas de archivos o bases de datos, pues en este caso serían pruebas de integración. Cuando una prueba unitaria requiere de datos, configuraciones o pasos previos para ejecutar las verificaciones, se utiliza la tecnología de *mocks*. Los *mocks* son objetos simples que simulan servicios reales como, por ejemplo, una llamada a una base de datos.

El aspecto clave de una prueba unitaria es anular las dependencias externas, ya que si no podrían incrementar la cantidad de tiempo que tarda la prueba en etapas de configuración y ejecución. Por ello, esta fase debería ejecutarse rápida e inmediatamente después de la construcción del código fuente, con un tiempo máximo de ejecución de cinco minutos (Humble y Farley, 2010).

5.3. PRUEBAS DE INTEGRACIÓN

Las pruebas de integración verifican interacciones entre diferentes módulos funcionales de una aplicación o con sistemas externos como un sistema operativo, un sistema de archivos o la base de datos, entre otros.

Las pruebas de integración son un paso hacia la verificación de la adecuación funcional y la compatibilidad. La adecuación funcional se mide al realizar la integración entre diferentes módulos funcionales del mismo sistema. Por otro lado, la compatibilidad se mide al realizar la integración del software con otros sistemas, como sistemas de archivos, bases de datos, hardware o sistemas operativos.

Al igual que las pruebas unitarias, para los entornos continuos, las pruebas de integración también deben ser automatizadas. De acuerdo a Jöngren (2008), las pruebas de integración automatizadas son el resultado de combinar la definición de pruebas de

integración con las pruebas unitarias automatizadas. Es decir, es posible la utilización de librerías de pruebas unitarias para automatizar las pruebas de integración. Otros autores, como Humble y Farley (2010), proponen el uso de herramientas de automatización de pruebas de aceptación para el mismo propósito. A continuación, se presenta un ejemplo de prueba de integración automatizada, escrita en la herramienta DbUnit, para interactuar con una base de datos.

```
public class DefaultWordDAOImplTest extends DatabaseTestCase {

    public DefaultWordDAOImplTest(String name) {
        super(name);
    }

    protected IDataSet getDataSet() throws Exception {
        return new FlatXmlDataSet(new File("test/conf/wseed.xml"));
    }

    protected IDatabaseConnection getConnection() throws Exception {
        final Class driverClass =
            Class.forName("org.gjt.mm.mysql.Driver");
        final Connection jdbcConnection =
            DriverManager.getConnection(
                "jdbc:mysql://localhost/words",
                "words", "words");
        return new DatabaseConnection(jdbcConnection);
    }

    public void testFindVerifyDefinition() throws Exception{
        final WordDAOImpl dao = new WordDAOImpl();
        final IWord wrd = dao.findWord("pugnacious");
        for(Iterator iter =
            wrd.getDefinitions().iterator();
            iter.hasNext();){
            IDefinition def = (IDefinition)iter.next();
            TestCase.assertEquals(
                "def is not Combative in nature; belligerent.",
                "Combative in nature; belligerent.",
                def.getDefinition());
        }
    }
}
```

Finalmente, como las pruebas de integración también verifican las interfaces entre módulos funcionales del mismo sistema, los que a su vez están compuestos por pequeñas unidades de código (funciones, métodos, subrutinas, etcétera), las mismas deben ser ejecutadas luego de que las pruebas unitarias hayan finalizado (Duvall, Matyas y Glover, 2007; Humble y Farley, 2010; Jöngren, 2008).

5.4. PRUEBAS FUNCIONALES Y NO FUNCIONALES

En la literatura se encuentran diferentes definiciones para el tipo de pruebas que se describe a continuación: pruebas de sistema, pruebas de aceptación, pruebas funcionales, pruebas de aceptación de usuario, pruebas no funcionales y pruebas de extremo a extremo. Esto quizás se deba a la evolución del proceso de pruebas.

El Comité Internacional de Certificaciones de Pruebas de Software define en 2002 a las «pruebas de sistema» como la verificación del comportamiento y funcionalidades de todo el sistema según lo definido en el alcance del proyecto. Esta etapa puede incluir pruebas basadas en riesgos, especificación de requisitos, procesos empresariales, casos de uso y otras descripciones de alto nivel del comportamiento del sistema. Las pruebas de sistema deben incluir tanto verificaciones funcionales como no funcionales, utilizando técnicas como caja negra y caja blanca. Por otro lado, este mismo comité define a las «pruebas de aceptación» como una validación realizada por los usuarios finales o por el mismo cliente para corroborar que el sistema efectivamente cumple los criterios de aceptación y requerimientos planteados al inicio del proyecto.

Las «pruebas de sistema» conforman la última etapa de pruebas en el lado del equipo de desarrollo. En la mayoría de las organizaciones, esta etapa es realizada por un equipo especializado en pruebas o por una evaluación externa.

En los enfoques continuos se definen a las «pruebas de aceptación» y a las «pruebas funcionales» de forma análoga añadiendo que estas pruebas también pueden ser llevadas a cabo dentro del equipo de desarrollo, desde un punto de vista de usuario final (Duvall, Matyas y Glover, 2007). Finalmente, en 2010, con la introducción de las «entregas continuas», se utiliza el término «pruebas de aceptación» para referirse tanto a las pruebas funcionales como a las de sistema (Humble y Farley, 2010). Esto se debe principalmente a dos razones:

- En todas las definiciones anteriores se realizan verificaciones de requerimientos funcionales y no funcionales, variando únicamente la persona o grupo de personas que las llevan a cabo (desarrollador, especialista en pruebas, cliente o usuario final).
- Si una característica del sistema no funciona como fue requerida (prueba funcional y sistema), entonces el sistema no es aceptado (prueba de aceptación). Es responsabilidad del equipo de desarrollo verificar que el sistema reúna los requisitos para ser aceptado por el cliente o por el usuario final.

Finalmente, para diferenciar a estas pruebas de las realizadas por el usuario final, se llaman a estas últimas «pruebas de aceptación de usuario».

Como este libro se centra en el desarrollo continuo de software, se utiliza el término «pruebas de aceptación» para referirse a la última etapa de pruebas realizada por el equipo de desarrollo, abarcando tanto a las pruebas funcionales como a las pruebas no funcionales. Por otro lado, se utiliza el término «pruebas de aceptación de usuario» para referirse a las validaciones finales que realiza el cliente o usuario final.

5.4.1. Pruebas funcionales

Es un tipo de prueba de aceptación que, como las pruebas unitarias y de integración, se utiliza para medir la adecuación funcional del sistema desarrollado. La principal diferencia con estas dos últimas es que se lleva a cabo con el sistema en funcionamiento y es una de las últimas etapas de verificación del lado del equipo de desarrollo. Esto quiere decir que las pruebas funcionales son el mecanismo principal para verificar el cumplimiento de los requerimientos funcionales del sistema.

Dependiendo del sistema, las pruebas funcionales se realizan sobre una interfaz gráfica de usuario (o GUI por su sigla en inglés: *Graphical User Interface*) o sobre una interfaz de programación de aplicaciones (o API por sus siglas en inglés: *Application Programming interface*).

La GUI puede ser un sitio web al que se accede mediante un navegador, un programa de escritorio que se instala en la computadora del usuario o una aplicación ejecutada desde un dispositivo móvil. La API puede ser un servicio web, una interfaz de líneas de comandos o cualquier otro tipo de interfaz de sistema que no sea gráfica. Sin embargo, los autores de los enfoques continuos solo mencionan y ejemplifican la automatización de pruebas de sitios web mediante herramientas como Selenium (Duvall, Matyas y Glover, 2007; Humble y Farley, 2010). Esto se debe a que los usuarios reales interactúan con la aplicación solamente a través de la GUI del sistema. pero este enfoque no contempla a un sistema como usuario final de otro sistema.

En el siguiente algoritmo se muestra un ejemplo de una prueba funcional automatizada, utilizando la herramienta Selenium WebDriver. En el trabajo de Sabev y Grigorova (2017) se presentan en profundidad las herramientas que existen en la actualidad para automatización de pruebas funcionales de GUI.

```
public class FunctionalTest {

    private WebDriver driver;

    @BeforeTest
    public void setUp() {
        driver = new FirefoxDriver();
    }

    @Test
    public void verifyLogin() {
        driver.get("http://localhost:8080/login.html");
        driver.findElement(By.id("username"))
            .sendKeys("user1234");
        driver.findElement(By.id("password"))
            .sendKeys("pass1234");
        driver.findElement(By.id("submit")).click();
        String expectedResult = "Welcome user1234";
        Assert.assertEquals(expectedResult, driver.getTitle());
    }

    @AfterTest
    public void tearDown() {
        driver.close();
    }
}
```

5.4.2. Pruebas no funcionales

Es un tipo de prueba de aceptación que se utiliza para verificar requerimientos no funcionales teniendo como principal fuente las características de la norma ISO 25010 vista en el capítulo 2 de este libro. Estas características son: eficacia en el desempeño, usabilidad, seguridad, fiabilidad y portabilidad. Adicionalmente, la mantenibilidad también es un requerimiento no funcional, pero es medida mediante las inspecciones de código vistas anteriormente.

Los requerimientos no funcionales ayudan al éxito del sistema. Sin embargo, entre los conceptos de integración o despliegue continuo los autores solo se centran en las pruebas de capacidad (Humble y Farley, 2010). Según Chen (2017), si bien las pruebas unitarias y las pruebas de aceptación se han estudiado ampliamente en el despliegue continuo, las pruebas de requerimientos no funcionales han recibido una atención considerablemente menor. En este sentido, la capacidad de un sistema abarca dos características principales (Nygard, 2018):

- Rendimiento (también conocido como *performance*): mide el tiempo transcurrido en el que un sistema procesa una transacción, en condiciones determinadas de trabajo.
- Transferencia de datos (también conocido como *throughput*): mide el número de transacciones que un sistema puede procesar en un intervalo de tiempo.

La medición de la capacidad de un sistema incluye asimismo determinar las características de la aplicación y, para las verificaciones, pueden utilizarse los siguientes tipos de pruebas (Humble y Farley, 2010):

- Pruebas de escalabilidad: verifican cómo varía el tiempo de respuesta de un número N de peticiones al sistema, al agregar más servidores, servicios o hilos.
- Pruebas de «longevidad»: también llamadas pruebas de picos o *spike testing*, consiste en la ejecución del sistema durante un periodo largo de tiempo para verificar si el rendimiento varía en algún momento. El objetivo es detectar pérdidas de memoria o problemas de estabilidad.
- Pruebas de transferencia de datos: se utiliza para detectar cuántas transacciones, mensajes o peticiones a un sitio web, el sistema soporta en un intervalo de tiempo determinado.
- Pruebas de carga: verifica cómo varía el tiempo de respuesta del sistema cuando el uso de la aplicación aumenta. Dentro de este tipo de pruebas, se encuentran las pruebas de sobrecarga o estrés, que buscan hacer fallar todo el sistema mediante la utilización de una gran cantidad de carga. El objetivo de esta última es determinar la robustez de la aplicación en los momentos de carga extrema.

Existe un gran número de herramientas para realizar pruebas de capacidad. La Tabla 7 lista las herramientas más utilizadas del mercado. En la segunda columna se indica el tipo de tecnología objetivo de la herramienta y en la tercera, el tipo de licencia de la herramienta.

Tabla 7. Herramientas más conocidas para automatización de pruebas de capacidad

| HERRAMIENTA | TECNOLOGÍA | LICENCIA |
|-----------------------------|-------------------|-----------------|
| Apache Jmeter | Web | Código abierto |
| Google Page Speed Insights | Web | Gratuito |
| GTMetrix | Web | Gratuito |
| Jetbrains doTrace | .NET | Comercial |
| LoadUI NG Pro | Web/API | Comercial |
| Microfocus LoadRunner | Web | Comercial |
| Open STA | Web/Windows | Código abierto |
| Rational Performance Tester | Web/Server | Comercial |
| SmartMeter.io | Web | Comercial |
| WebLoad | Web/Mobile | Comercial |
| Gatling | Web | Código abierto |
| The Grinder | Web | Código abierto |

Fuente: Maila-Maila, Intriago-Pazmiño y Ibarra-Fiallo (2019) y Rodríguez, Piattini y Ebert (2019).

Por otro lado, la Figura 19 muestra el resultado de la monitorización del tiempo de respuesta bajo diferentes parámetros de pruebas de capacidad, utilizando la herramienta Load Impact.

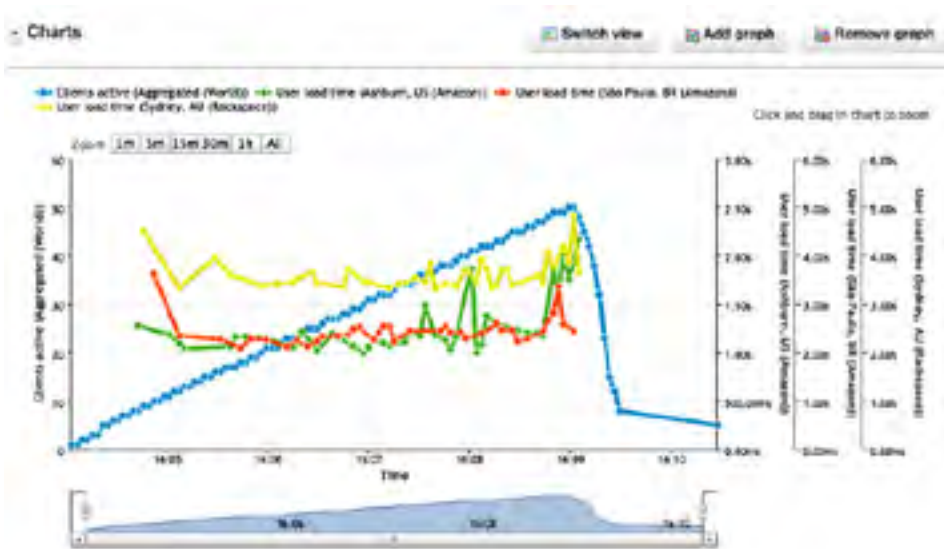


Figura 19. Monitorización del tiempo de respuesta utilizando determinados escenarios de pruebas de capacidad.

Sin embargo, para incorporar las pruebas de capacidad en los entornos continuos, no basta con utilizar herramientas de automatización, también debe convertirse en una etapa adicional de la tubería de despliegue, como se detalla en la Figura 20. Esta etapa no solo contiene verificaciones de características de capacidad, como el rendimiento y las transacciones de datos, sino también la configuración del ambiente donde se realizarán las pruebas, el despliegue de los archivos ejecutables, pequeñas pruebas de humo¹ y finalmente la ejecución de las pruebas de capacidad.

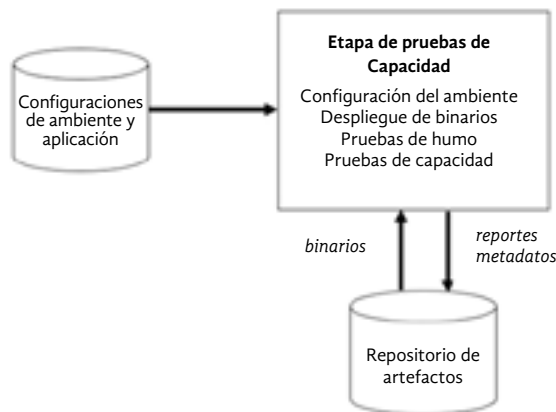


Figura 20. Etapa de pruebas de capacidad en la tubería de despliegue (Humble y Farley, 2010).

1. Las pruebas de humo (*smoke testing*) son una verificación rápida que se realiza para asegurarse de que la funcionalidad básica de la aplicación se encuentre estable y responda al comportamiento esperado. Para ello, se utiliza un grupo seleccionado de pruebas, del conjunto universal de pruebas funcionales.

Pruebas de aceptación del usuario

Existen tareas que no se pueden automatizar como, por ejemplo, la ejecución de pruebas basadas en la experiencia y la capacidad de análisis de un especialista en pruebas o las demostraciones de software funcional a los responsables del proyecto. En el despliegue continuo, este grupo de tareas pertenece a la etapa de pruebas de aceptación de usuario, que abarcan pruebas exploratorias, pruebas de usabilidad y demostraciones (Crispin y Gregory, 2009; Gregory y Crispin, 2014; Humble y Farley, 2010). Esta etapa debe realizarse continuamente e incluirse en la tubería de despliegue.

En los entornos continuos, el rol del especialista en pruebas en este proceso no debería disparar una validación total del sistema² en cada iteración, sino solamente el cumplimiento de los criterios de aceptación de los requerimientos que han sido desarrollados e incluidos en la integración que se está realizando en ese momento. Además, existen tareas que no pueden automatizarse como las pruebas exploratorias, las pruebas de usabilidad y las pruebas de aceptación de usuario.

Las pruebas exploratorias son un tipo de pruebas manuales. El especialista en pruebas navega a través de la aplicación, utilizando su experiencia y capacidad de análisis para detectar nuevos casos de prueba que pueden ser automatizados luego (Whittaker, 2009). Es un proceso creativo de aprendizaje, cuyo objetivo no es solo encontrar defectos, sino también aumentar la cobertura de las pruebas automatizadas mediante la detección de nuevos casos de pruebas (Humble y Farley, 2010; Whittaker, 2009).

Las pruebas manuales realizadas por equipos de pruebas especializadas, por los dueños de producto o por el mismo cliente para validar que el software funcione de acuerdo a los requerimientos, se denominan pruebas de aceptación de usuario. Si el equipo de desarrollo se encuentra entregando software a producción siguiendo un contrato, las pruebas de aceptación de usuario son una etapa requerida para la aprobación final del software (Crispin y Gregory, 2008). Es una manera de asegurar que «los usuarios finales pueden realizar sus tareas» (Gregory y Crispin, 2014).

Por otro lado, las pruebas de usabilidad deben incorporarse en los entornos continuos para descubrir qué tan fácil es para los usuarios finales cumplir sus objetivos con el software que se está desarrollando (Humble y Farley, 2010). Según la norma ISO 9241 (2011: 5), la usabilidad se define como «el grado en el que un producto puede ser utilizado por usuarios específicos para conseguir objetivos específicos con efectividad, eficiencia y satisfacción en un determinado contexto de uso». Existen herramientas que permiten automatizar algunos aspectos de las pruebas de usabilidad, pero ninguna de ellas puede automatizar el proceso total.

Otra manera de realizar validaciones desde el punto de vista del usuario es mediante las presentaciones del sistema a los clientes o dueños del producto. Estas presentaciones se conocen como demostraciones o showcases. Los equipos ágiles realizan este tipo de presentaciones al final de cada iteración para demostrar la nueva funcionalidad o características del sistema que serán entregadas (Cohn, 2009). La salida de esta reunión es la

2. Las pruebas de regresión consisten en la ejecución total de todos los casos de pruebas que verifican que se cumplan todos los requerimientos funcionales y no funcionales desde el inicio del proyecto.

aceptación de la nueva funcionalidad, sugerencias para mejorarla, el rechazo de la misma o el surgimiento de nuevas ideas que serán planificadas y convertidas en nuevos desarrollos en el futuro. Las demostraciones se pueden considerar como «el primer latido» de todo proyecto, por ser la primera vez que se obtiene la realimentación del cliente, usuario final o dueños del producto.

En general, la etapa de pruebas manuales verifica que el sistema funciona con los últimos cambios de código que representan los nuevos requerimientos funcionales y no funcionales que han sido desarrollados, detectando defectos que no hayan sido detectados por las pruebas automatizadas y validando que aporte valor para los usuarios finales.



Capítulo 6. Definición y fases de las pruebas continuas de software

En los anteriores capítulos se dio una aproximación a las pruebas continuas de software desde dos enfoques. Por un lado, se estudió la evolución de la integración, el despliegue y la entrega continua de software como un conjunto de técnicas cuya fuente principal es el desarrollo ágil de aplicaciones y, por otro lado, se enumeraron las diferentes actividades de prueba y su inclusión en el paradigma de la continuidad en el desarrollo software.

Pero esto trae consigo una serie de desafíos, problemas y soluciones que todavía se están estudiando. Los siguientes capítulos de este libro se ocupan de estos conceptos. En particular, este capítulo presenta un estudio detallado sobre la definición de una prueba continua de software y los pasos para el desarrollo de pruebas continuas de software de acuerdo a la literatura científica.



6.1. IMPORTANCIA DE LAS PRUEBAS CONTINUAS DE SOFTWARE

Un software que no funciona correctamente puede dar lugar a muchos problemas, incluyendo la pérdida de dinero, tiempo, imagen corporativa, daños personales o incluso la muerte. En este sentido, Kuhn, Wallace y Gallo (2004) analizaron 329 reportes de errores producidos en etapas de desarrollo e integración de un sistema distribuido de la Nasa. Uno de estos errores tiene que ver con la catástrofe del cohete Ariane 5 (Lions *et al.*, 1996), que fue producida por un defecto de software consistente en una conversión numérica errónea.

Estos ejemplos demuestran por qué son tan necesarias las pruebas de software en cualquier proceso de desarrollo. Tradicionalmente, las pruebas eran consideradas una tarea que debía realizarse al final del ciclo de desarrollo de software, justo antes de que el producto fuera entregado a los usuarios, como se muestra en la Figura 21.



Figura 21. Etapa de pruebas en los modelos tradicionales de desarrollo de software.

Con el correr de los años, se comprobó que la detección de errores al final del proceso era muy costosa en comparación con la detección de errores en etapas tempranas. En la Figura 22 se muestra nuevamente cómo el costo de solucionar defectos aumenta con respecto a las fases de desarrollo de una manera no lineal, sino más bien geométrica.

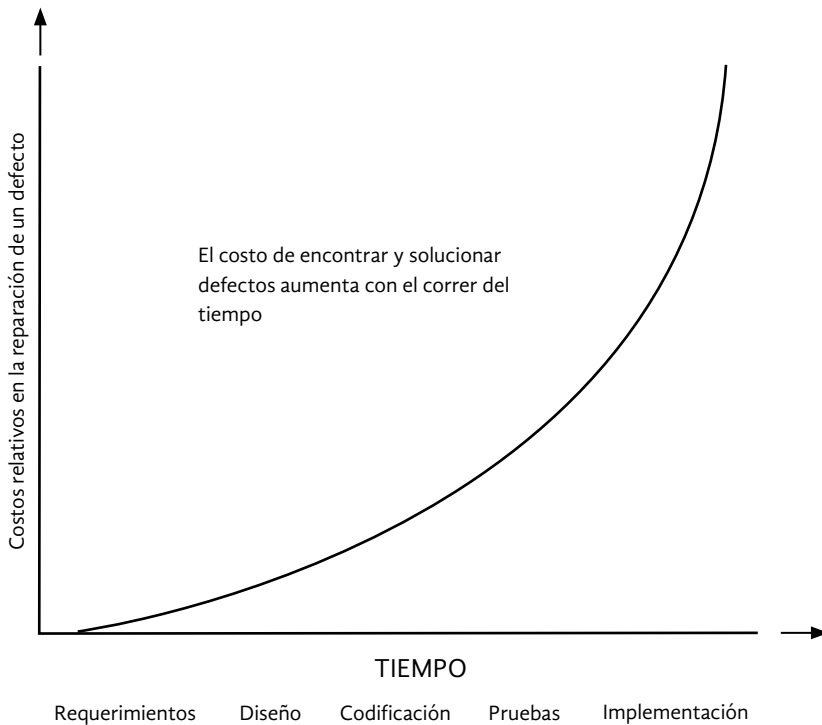


Figura 22. Costo de solucionar defectos en las diferentes etapas del desarrollo (Jain y Joshi, 2016).

Además de los costos elevados asociados a encontrar y solucionar defectos al final del proceso de desarrollo de software, Mike Cohn (2009) presenta una lista de razones por las cuales el modelo tradicional de pruebas no funciona:

- Es difícil mejorar la calidad de un producto que ya existe
- Los errores son desapercibidos
- El estado del proyecto es difícil de medir
- Las oportunidades de obtener *feedback* se pierden
- Las pruebas tienden a ser interrumpidas, reducidas o abandonadas, debido a ser la última actividad del proceso, y se tienen que cumplir los tiempos de entrega.

Por estas razones, la mayoría de los equipos migraron a enfoques de desarrollo iterativos y ágiles. Por ejemplo, los equipos que trabajan con metodologías ágiles hacen del proceso de pruebas un componente que acompaña al ciclo de vida del desarrollo de software, donde se trabaja en mejorar la calidad de los procesos y el producto de manera continua, desde el inicio del proyecto. De acuerdo a Jain y Joshi (2016), las pruebas deberían ser vistas como un esfuerzo colaborativo en todas las etapas, en lugar de ser una tarea que se realiza solo al final del proceso de desarrollo. Este modelo de pruebas actual se conoce como pruebas ágiles (Crispin y Gregory, 2008). En la Figura 23 se muestra un ejemplo de un proceso iterativo, que incluye a las pruebas en cada iteración.

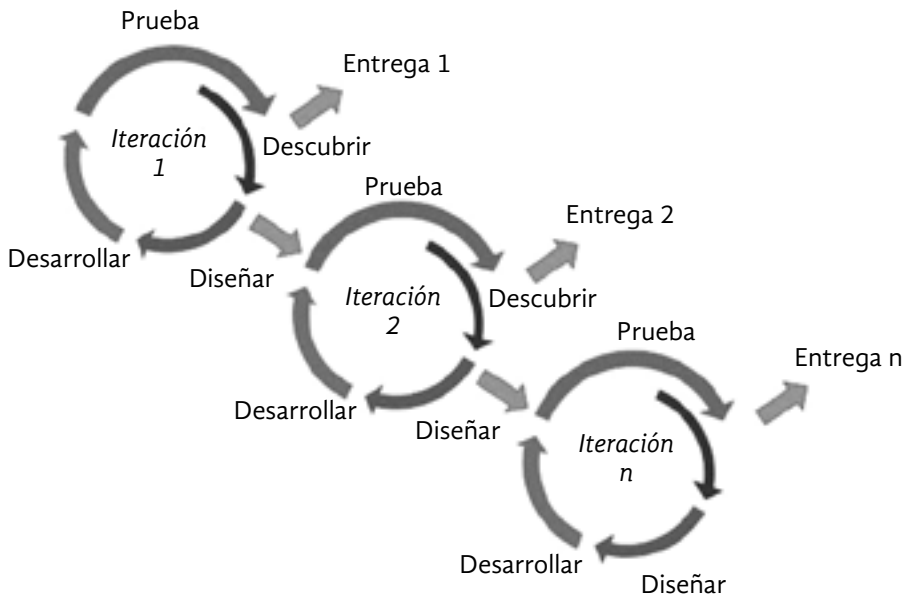


Figura 23. Etapas en un proceso de desarrollo de software iterativo.

No obstante, que las pruebas acompañen todo el proceso de desarrollo de software no es suficiente, pues en los procesos de desarrollo continuo estas deben ser automatizadas.

Recordando lo descrito en capítulos anteriores, en la integración continua se recomienda automatizar las pruebas unitarias, de integración y de rendimiento, e incluirlas en los scripts de construcción (Duvall, Matyas y Glover, 2007). Estos scripts luego tienen que ser ejecutados de manera local por los desarrolladores y posteriormente en el servidor de integración continua, al introducirse los cambios en el repositorio de control de versiones. En el despliegue continuo se buscan automatizar más tipos y niveles de pruebas (Humble y Farley, 2010) e introducirlos en el la tubería de despliegue.

Las pruebas automatizadas forman parte de uno de los componentes principales del despliegue continuo, ya que su objetivo es realizar despliegues de productos fiables. Sin embargo, la velocidad requerida para lanzar el producto al mercado con frecuencia en este tipo de enfoques hace que las empresas pasen por alto muchos detalles a la hora de realizar pruebas.

El reto principal consiste en encontrar el balance entre tiempo y exhaustividad en las pruebas, recomendado por Duvall (2007). Por un lado, si las pruebas son de extensa duración, el proceso se vuelve tedioso y genera bloqueos a los desarrolladores, mientras esperan por *feedback*. Por otro lado, si las pruebas no son lo suficientemente exhaustivas, se pueden introducir defectos en el software. Sin embargo, los resultados de las investigaciones que se presentan a continuación muestran que aún no hay una solución a este desafío y que es evidente la carencia de modelos y herramientas para asegurar la calidad de software en los procesos de desarrollo continuo.

Si bien la literatura contiene instrucciones y buenas prácticas de cómo adoptar el despliegue continuo, su implementación ha sido un desafío en la práctica (Laukkanen, Itkonen y Lassenius, 2017). Algunas organizaciones no han podido adoptar el despliegue continuo completamente y otras han encontrado gran cantidad de obstáculos (Ståhl y Bosch, 2014). La mayor parte de los problemas reportados están relacionados principalmente con las etapas de pruebas e integración.

Es por ello que, para algunos autores, las «pruebas continuas» son el elemento faltante en el proceso de desarrollo continuo que incluyen las prácticas mencionadas.

6.2. LOS PROBLEMAS QUE TIENEN LAS PRUEBAS EN EL DESPLIEGUE CONTINUO DE SOFTWARE

Como ya fue mencionado antes, diferentes autores han reportados problemas en la implementación del despliegue continuo, especialmente en las etapas de integración y pruebas. En la Tabla 8 se listan los problemas relacionados con la calidad de software que han sido reportados en revisiones de la literatura, casos de estudio y artículos empíricos, cuyo objetivo era el estudio y la implementación del despliegue continuo.

Tabla 8. Problemas relacionados a la etapa de pruebas en el despliegue continuo de software

| PROBLEMA | REFERENCIAS |
|--|--|
| Pruebas que consumen mucho tiempo de ejecución | Neely y Stolt (2013) Chen (2017) Brooks (2008) |
| Pruebas no deterministas o <i>flaky tests</i> | Neely y Stolt (2013) (Laukkanen, Itkonen y Lassenius (2016) (Cannizzo, Clutton y Ramesh (2008) Leppänen <i>et al.</i> (2015) Debbiche y Dienér (2015) |
| Dificultades en pruebas automatizadas de interfaz gráfica de usuario | (Alégroth, Feldt y Ryrholm (2015) Börjesson y Feldt (2012) Pradhan (2012) Suwala (2015) (Mascheroni, Cogliolo y Irrazábal (2017) Sabaren <i>et al.</i> (2018) |
| Resultados de ejecución de pruebas ambiguos | Ståhl y Bosch (2014) Cannizzo, Clutton y Ramesh (2008) |
| Dificultades en las pruebas de contenido dinámico web | Suwala (2015) |
| Dificultades en las pruebas en Big Data | Garg, Singla y Jangra (2016) |
| Dificultades en las pruebas de datos | Muşlu, Brun y Meliou (2013) |
| Dificultades en la implementación de pruebas en dispositivos móviles | Muccini, Di Francesco y Esposito (2012) |
| Dificultades en pruebas automatizadas no funcionales | Chen (2017) Leppänen <i>et al.</i> (2015) |

En la investigación basada en evidencias, el principal método utilizado es la revisión sistemática de la literatura (RSL). Una RSL proporciona una manera de identificar, evaluar e interpretar toda la investigación disponible sobre una cuestión de investigación, área o fenómeno de interés (Kitchenham, 2004). Las RSL son una revisión metodológicamente rigurosa de los resultados de investigación, cuyo objetivo final es servir a los investigadores para proporcionar soluciones de Ingeniería del Software apropiadas en un contexto específico.

En este marco, el Grupo de Investigación en Calidad de Software (Gics) de la Universidad Nacional del Nordeste realizó una RSL para buscar las soluciones de pruebas continuas que pueden ser utilizadas en el despliegue continuo (Mascheroni, 2018b). Como corolario de ello, también se ha investigado el significado que tienen las pruebas continuas para la industria y las diferentes etapas o niveles de pruebas que lo componen. A continuación, se presentan los resultados de este estudio en particular, respondiendo a la pregunta de investigación que da título al apartado.

6.3. ¿EXISTE UNA DEFINICIÓN VÁLIDA Y ACEPTADA PARA LAS PRUEBAS CONTINUAS?

El término fue mencionado por primera vez por Edward Smith (2000), como parte del proceso de Desarrollo Dirigido por Pruebas o TDD (*Test Driven Development*, por sus siglas en inglés), considerando las pruebas unitarias. Consistía en un proceso de pruebas que tenía que ser aplicado durante las etapas de desarrollo y de pruebas constantemente, como una regresión automática que se ejecutaba las 24 horas del día. Sin embargo, los resultados obtenidos de los trabajos seleccionados muestran que este concepto ha ido evolucionando durante los últimos años.

Los autores Saff y Ernst (2003: 3) introdujeron el concepto de prueba continua como «un medio para reducir el tiempo perdido en la ejecución de pruebas unitarias». Utilizaron la integración en tiempo real con el entorno de desarrollo para ejecutar asincrónicamente las pruebas que se aplican a la última versión del código, obteniendo eficiencia y seguridad al combinar las pruebas asincrónicas con las pruebas sincrónicas. Más tarde, los mismos autores propusieron una herramienta que complementaba el entorno de trabajo de los desarrolladores de software y hacía posible las pruebas de regresión en segundo plano, mientras el desarrollador escribía el código fuente (Saff y Ernst, 2004). Estas pruebas consistían en pruebas unitarias y pruebas de integración. Llamaron a este proceso «pruebas continuas», ya que proporcionaban *feedback* rápido a los desarrolladores con respecto a los errores que introducían inadvertidamente durante la escritura del código. Esta definición de prueba continua es la base de otras definiciones de prueba continua y también es utilizada por otros autores en la actualidad. Por ejemplo, Eyl, Reichmann y Müller-Glaser (2016: 2) usan el mismo término para referirse a «un proceso que proporciona *feedback* rápido sobre la calidad del código al ejecutar automáticamente pruebas de regresión en segundo plano mientras el desarrollador está cambiando el código fuente».

Burgin y Debnath (2010: 5) perfeccionan la definición y recomiendan «ejecutar casos de prueba todo el tiempo durante el proceso de desarrollo para garantizar la calidad del sistema software que se construye». Los autores presentaron un nuevo concepto, el de «pruebas continuas totales de por vida», que incluyen no solo la etapa de pruebas unitarias, sino también las siguientes: pruebas de especificación, pruebas de diseño, pruebas de codificación, pruebas de validación, pruebas funcionales, pruebas no funcionales, pruebas de despliegue, pruebas de operación, pruebas de soporte y pruebas de mantenimiento.

En 2013, la empresa Google sugirió «ejecutar cualquier tipo de prueba lo antes posible y detectar cualquier tipo de problema relacionado con un cambio realizado por un desarrollador» (Penix, 2012: 3).

Ya entre 2015 y 2016 aparecieron nuevos enfoques de pruebas continuas que incorporaron la experiencia del trabajo en entornos de alta productividad. Erder y Pureur (2015: 120) aconsejaron «usar la automatización para mejorar significativamente la velocidad de las pruebas al adoptar un enfoque de desplazamiento a la izquierda, que integra las fases de control de calidad y desarrollo». Este enfoque incluye flujos de trabajo automatizados de pruebas que se pueden combinar con métricas para proporcionar una imagen clara de la calidad del software que se entrega. La finalidad es establecer *feedback* sobre

la calidad del software que están desarrollando. También permite realizar pruebas antes y con una mayor cobertura al eliminar los cuellos de botella de pruebas, como el acceso a entornos de prueba compartidos o esperar a que la interfaz de usuario se estabilice.

Para Moe *et al.* (2015), las pruebas continuas implican ejecutar las tareas de prueba de inmediato al integrar los cambios de código con el sistema y actualizar los casos de pruebas para ejecutar las de regresión en cualquier momento. Para Virmani (2015), por otro lado, el concepto de prueba continua es la automatización de cada caso de prueba. De esta manera, los procesos de entrega de software podrían ejecutar la totalidad de las pruebas en cada compilación de software sin ninguna intervención del usuario, avanzando así hacia el último objetivo: poder entregar una versión de software con calidad y rápidamente.

Los resultados muestran que el concepto de prueba continua ha evolucionado incorporándose a todo el ciclo de vida del desarrollo (ver la Figura 24). Al principio, solo se aplicaba a la ejecución de pruebas unitarias de forma continua y en segundo plano, en la computadora del desarrollador mientras este seguía trabajando. La inclusión de diferentes etapas de pruebas durante los años en las definiciones de las pruebas continuas está relacionada con la aparición de herramientas que permiten a los equipos automatizar diferentes tipos de casos de prueba. Sin embargo, la mayoría de estos artículos basaron sus enfoques de prueba continua en la definición de Saff y Ernst (2003), y algunos de ellos utilizan la definición propuesta por Smith (2000). Por lo tanto, se puede concluir que la prueba continua es el proceso más eficiente para ejecutar cualquier tipo de caso de prueba automatizado lo más rápido posible para proporcionar feedback rápido al desarrollador y detectar problemas críticos antes que el producto software se encuentre en producción.

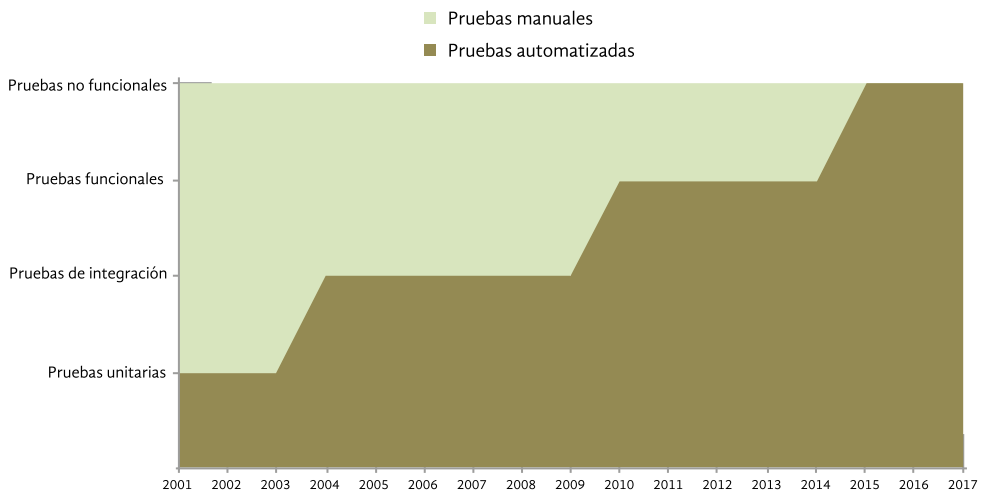


Figura 24. Evolución de las pruebas automatizadas durante los años (Mascheroni y Irrazábal, 2018).

6.4. NIVELES DE PRUEBA EN PROYECTOS DE DESARROLLO DE SOFTWARE CONTINUO

En general, son cuatro los niveles o etapas de pruebas más aceptados (Van Veenendaal y Graham, 2008): las pruebas unitarias, las pruebas de integración, las pruebas de sistema y las pruebas de aceptación que, a su vez, se pueden dividir en subniveles. En el capítulo anterior se analizó brevemente cada nivel y su enfoque desde el punto de vista del desarrollo ágil de aplicaciones.

Estos niveles no son siempre respetados en los procedimientos de integración y despliegue continuo. Teniendo en cuenta el estudio de la literatura científica actual, se pueden encontrar en total nueve etapas diferenciadas en las pruebas continuas de software que se describirán a continuación.

La primera etapa de pruebas en el despliegue continuo es la *revisión por pares*. Fue propuesta como una etapa de prueba o etapa de aseguramiento de calidad por Gomedede, Da Silva y De Barros (2015). Es una etapa manual, pero puede ser compatible con el uso de herramientas que asistan a los desarrolladores. Del mismo modo, Rossi *et al.* (2016) utilizan la *revisión de código* como un requisito antes de que cualquier cambio pueda enviarse a la rama principal del sistema de control de versiones.

La segunda etapa de pruebas en el despliegue continuo es la *construcción y las pruebas unitarias*. Gotimer y Stiehm (2016) han extendido esta etapa mediante el uso de pruebas de mutación automatizadas. La prueba de mutación es un proceso mediante el cual el código existente se modifica de diferentes maneras específicas (por ejemplo, invirtiendo una prueba condicional de igual a no igual o cambiando un valor verdadero a falso) y luego las pruebas unitarias se ejecutan nuevamente. Si el código modificado o la mutación no hacen que una prueba falle, entonces la prueba funciona. Asimismo, algunos autores también agregan a este nivel una etapa de *cobertura de código fuente* que se ejecuta al mismo tiempo que las pruebas unitarias.

La tercera etapa de pruebas es el *análisis estático del código fuente*. Esta metodología de trabajo fue implementada en las principales publicaciones de referencia (Gomedede, Da Silva y De Barros, 2015; Rathod y Surve, 2015). Es otra etapa automatizada que examina el código sin ejecutarlo, detectando problemas de estilos de codificación, alta complejidad, bloques de código duplicados, prácticas de codificación confusas o falta de documentación. El análisis estático permite que las revisiones manuales de código se concentren en diseños importantes y problemas de implementación, en lugar de aplicar estándares de codificación estilísticos, como el uso de espacios al inicio de un método o el formato de un comentario en el código fuente.

La cuarta etapa reportada son las *pruebas de integración*. En esta etapa, los módulos de software individuales se combinan y se prueban como un grupo.

Algunos autores llaman a la quinta etapa *pruebas de instalación o pruebas de despliegue*. Tiene el objetivo de verificar si la instalación del sistema o el despliegue de software en un entorno específico se realizó correctamente. Es una etapa de verificación muy corta, y las herramientas involucradas son las mismas que se utilizan para las pruebas unitarias o de integración.

La sexta etapa de pruebas en el despliegue continuo está constituida por las *pruebas funcionales*. El objetivo de esta etapa es verificar que las funcionalidades del sistema trabajen conforme lo esperado. Es una de las etapas más descritas entre los autores; sin embargo, algunos de los estudios (Gmeiner, Ramler y Haslinger, 2015; Rossi *et al.*, 2016) afirman que no es posible automatizar todos los casos de pruebas funcionales del proyecto. Por lo tanto, también utilizan pruebas manuales funcionales. Otros autores denominan a esta etapa como *pruebas de conformidad*, *verificación de características*, *pruebas funcionales de sistema* y *pruebas de aceptación funcionales*. En este sentido, asimismo es importante agregar pruebas negativas a esta etapa. Las pruebas negativas aseguran que el sistema pueda manejar entradas no válidas o comportamientos inesperados del usuario. Además, se han propuesto otras subetapas de pruebas como parte de la etapa de prueba funcionales, como las pruebas de instantáneas o *snapshot testing* (Rossi *et al.*, 2016). El propósito de estas últimas es generar capturas de pantalla de la aplicación que luego se comparan, píxel por píxel, con versiones de instantáneas anteriores.

La séptima etapa son las *pruebas de seguridad*. Se utilizan para verificar que se cumplan los requisitos de seguridad como confidencialidad, integridad, autenticación, autorización, disponibilidad y no repudio.

La octava etapa de pruebas documentada es la constituida por *las pruebas de rendimiento*, *pruebas de carga* y *pruebas de estrés o sobrecarga*. Esta etapa se implementa para determinar el comportamiento de un sistema en condiciones normales y anticipadas de carga máxima. Una subetapa adicional en este nivel son las pruebas de capacidad, dirigidas a verificar si la aplicación puede gestionar la cantidad de tráfico y conexiones esperables.

Finalmente, la novena etapa de pruebas en el despliegue continuo es la de *pruebas manuales exploratorias*. Las pruebas manuales se vuelven más importantes, ya que las pruebas automatizadas cubrirán los aspectos genéricos, dejando los problemas más alternativos o menos probables de ocurrir sin descubrir.

Todas estas etapas de pruebas se han implementado para diferentes tipos de plataformas: aplicaciones web, aplicaciones móviles, computación en la nube, servicios web o aplicaciones de big data. Las etapas de pruebas implementadas en los estudios se muestran en la Figura 25. Las pruebas unitarias, las funcionales y las de capacidad son las más utilizadas en entornos de desarrollo continuo de software.

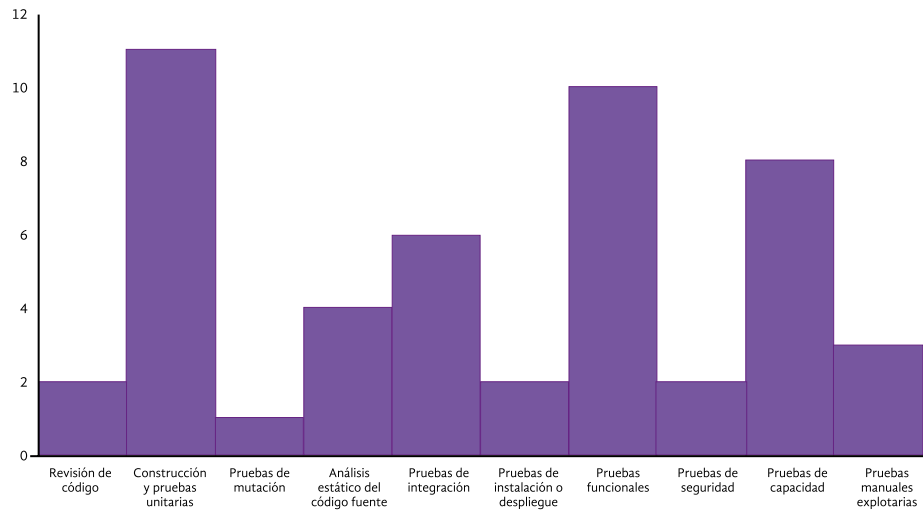


Figura 25. Etapas de pruebas en entornos continuos implementas por los estudios de la RSL.



Capítulo 7. Problemas en las pruebas continuas

El desarrollo de pruebas continuas tiene una gran cantidad de problemas y desafíos. Muchos de ellos, vinculados al logro de la automatización, la selección y la eficiencia de los tiempos de ejecución de las pruebas.

7.1. PROBLEMAS DETECTADOS EN LA LITERATURA ACADÉMICA

A continuación, se presenta un análisis de los problemas detectados en el desarrollo de las pruebas continuas de software que han sido reportados por diferentes autores en la literatura académica.

En una primera clasificación, estos problemas principales pueden resumirse en:

- **Pruebas que consumen mucho tiempo de ejecución.** Existen muchos tipos de pruebas que pueden implementarse en el despliegue continuo. Sin embargo, ejecutar una gran cantidad de pruebas es una tarea que lleva mucho tiempo. Además, los cambios son introducidos con más frecuencia y, por eso, es necesaria la ejecución de regresiones lo más rápido posible.
- **Pruebas no deterministas o *flaky tests*.** Es aquella que podría generar un resultado positivo o negativo por la misma versión del software. Son pruebas que producen los llamados «falsos positivos» y por su inestabilidad son más conocidas con el término inglés *flaky tests*. Una de las características más importantes del despliegue continuo es la confiabilidad. Por ello, las pruebas que fallan aleatoriamente no son confiables.
- **Automatización de pruebas de interfaz gráfica de usuario.** Antes eran ejecutadas de manera manual, pero con la aparición de herramientas como *Selenium WebDriver*, fue posible comenzar a automatizarlas. Sin embargo, la interfaz de usuario es la parte de la aplicación que cambia con más frecuencia y realizar pruebas sobre la misma puede ocasionar *flaky tests*.

- **Resultados ambiguos en la ejecución de pruebas.** En entornos de construcción continua del software, los desarrolladores deben ser notificados de cualquier defecto introducido o detectado a través de las pruebas. Cuando los resultados de la ejecución no son claros, es decir, no detallan el error, su causa, el lugar donde ocurrió, capturas de pantalla o cualquier otro tipo de información que permita la reproducción del error, el resultado se considera ambiguo.
- **Automatización de pruebas web con contenido dinámico.** Las aplicaciones web más modernas utilizan tecnologías como *ajax*, *react* o *angular* que generan contenido dinámico. Incorporar pruebas automatizadas de este tipo de tecnologías es complejo: una página web pudo alcanzar el estado esperado final, pero el contenido dinámico aún se encuentra cargando.
- **Pruebas de datos.** Son muy importantes para diferentes tipos de sistemas y constituyen, de alguna manera, el alimento de los procesos digitalizados por un desarrollo software. Si bien las pruebas de software han recibido gran atención en diferentes niveles, las pruebas de datos han sido poco tenidas en cuenta. Existen ejemplos, como las pruebas unitarias en base de datos o la normativa ISO/IEC 25012 sobre calidad en los datos, pero no es sencillo encontrar implementaciones para las pruebas de datos en un entorno de despliegue continuo.
- **Pruebas en big data.** A diferencia de las pruebas anteriores, son las pruebas relacionadas con el uso de grandes conjuntos de datos que no se pueden procesar utilizando técnicas tradicionales. Realizar verificaciones sobre este conjunto de datos es un nuevo reto que involucra varias técnicas y herramientas que aún están madurando y, por lo tanto, es difícil incorporarlas en el despliegue continuo.
- **Pruebas en dispositivos móviles.** Esta nueva tecnología genera nuevos paradigmas para el proceso de pruebas en sí, los niveles y tipos de pruebas a considerar, los diferentes tipos de dispositivos y los costos de las pruebas automáticas. Todos estos factores dificultan la implementación de pruebas en dispositivos móviles en el despliegue continuo.
- **Pruebas automatizadas de requerimientos no funcionales.** Si bien las pruebas unitarias, de integración y funcionales han sido estudiadas ampliamente en la literatura e implementadas por muchas organizaciones en la práctica, las pruebas no funcionales han sido poco consideradas. Las características de calidad como el rendimiento, la fiabilidad o la interoperabilidad son difíciles de medir y existen herramientas o estudios parciales como para ser automatizados en un proceso de despliegue continuo.
- **Pruebas de servicios web.** Aunque existen muchas herramientas para probar este tipo de arquitecturas, es difícil integrarlas en el despliegue continuo. Al igual que con los dispositivos móviles o los datos la tecnología es parcial y, por lo tanto, dificulta su uso en entornos automatizados.
- **Pruebas de aplicaciones compuestas por servicios en la nube.** Suman el desafío de la garantía en la calidad del servicio ofrecido.
- **TaaS.** Las pruebas como un servicio (también conocidas como TaaS por sus siglas en inglés: *test as a service*) es un modelo en el que las pruebas son ejecutadas por un proveedor de servicios en lugar de un equipo de pruebas perteneciente a la misma organización que desarrolla el software. Realizar TaaS en despliegue continuo resulta

una tarea muy compleja, puesto que los componentes y procesos del proveedor de servicios deben adaptarse a las características de la organización contratante.

Los problemas mencionados representan dificultades para la implementación correcta del proceso de pruebas en el despliegue continuo. Además, los mismos se encuentran relacionados entre sí, de tal manera que la ocurrencia de uno produce directamente otro o lo puede producir. Estas relaciones se muestran en la Figura 26. Para ahorrar espacio y mejorar la lectura, se incluye la sigla CD como sinónimo de despliegue continuo.

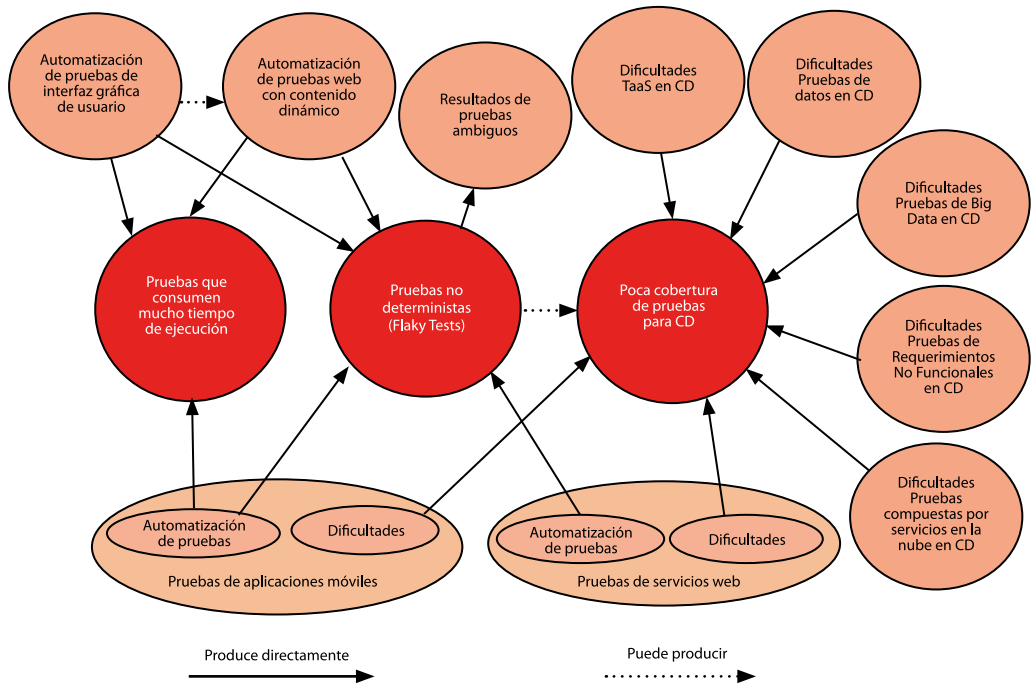


Figura 26. Relación entre los problemas de pruebas en despliegue continuo (Mascheroni, 2018).

En la Figura 26 puede apreciarse que los problemas más importantes son las pruebas que consumen mucho tiempo de ejecución y las pruebas automatizadas no deterministas. Además, basándonos en los problemas relacionados con dificultades en la implementación de ciertos tipos de pruebas, se ha agregado un problema más: poca cobertura de pruebas en despliegue continuo debido a la carencia de herramientas, metodologías, modelos o conjunto de buenas prácticas para pruebas automatizadas.

7.2. PROBLEMAS DETECTADOS POR LA INDUSTRIA

A lo largo de 2019, el Grupo de Investigación en Calidad de Software de la Universidad Nacional del Nordeste realizó una encuesta en la industria del software, cuyo foco fueron los proyectos y equipos que trabajan bajo metodologías ágiles e implementan prácticas de integración o despliegue continuos (Mascheroni *et al.*, 2019). Se obtuvieron 255 respuestas válidas pertenecientes a empresas ubicadas principalmente en los Estados Unidos, Canadá, Inglaterra, Alemania, Suiza, Argentina, Brasil, Chile, Irlanda y China. Los proyectos son de desarrollo de aplicaciones web de aerolíneas, sitios de comercio electrónico, sitios de agencias de viajes, motores de búsqueda, compañías de software de relaciones públicas, compañías de seguros, bancos o portales de noticias, entre otros.

A partir de esa encuesta, se identificaron dos nuevos problemas que no se habían detectado antes en la literatura académica actual. Por un lado, la falta de procedimientos, patrones y buenas prácticas para pruebas automatizadas en desarrollo continuo y, por otro lado, la presencia de ambientes inestables de trabajo en el marco de la integración o el despliegue continuo.

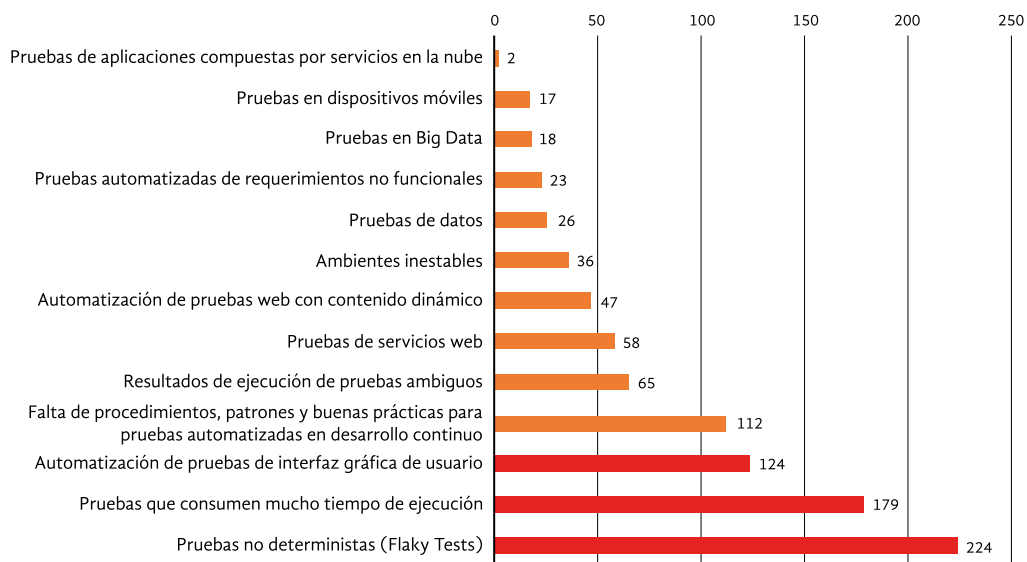


Figura 27. Problemas relacionados a las pruebas en la industria.

Como puede verse en la Figura 27, las pruebas no deterministas son el problema más reportado. Según los encuestados, hay varias causas de pruebas no deterministas, entre las que se incluyen fallos por datos de prueba faltantes o dañados, componentes de interfaz de usuario faltantes, tiempos de espera agotados debido a problemas del entorno, elementos dinámicos que tardan demasiado en cargar e inconsistencias en el código de las pruebas. La mayoría de los encuestados declararon que las pruebas no deterministas son la razón principal para tener pruebas automatizadas en un flujo de despliegue continuo. Con respecto a las pruebas automatizadas de interfaz de usuario, el problema más difícil informado es el mantenimiento de los scripts de pruebas debido a cambios en los localizadores de los elementos de la interfaz del usuario¹, así como su tiempo de ejecución. Del mismo modo, ejecutar pruebas automatizadas en un sitio web que tiene elementos dinámicos es un desafío para los equipos de desarrollo; por ejemplo, una prueba puede fallar porque la página ya ha cargado por completo y el elemento aún no se muestra, debido a llamadas asíncronas.

Asimismo, las pruebas asociadas con grandes datos o big data presentan una alta severidad. En todos los casos se ha dado mucha importancia a la falta de procedimientos estándar y, especialmente, a lo nuevo de la tecnología a ser probada.

Finalmente, algunos desarrolladores declararon que no existe una guía con buenas prácticas y procesos para lograr una buena cobertura de pruebas automatizadas, desde las pruebas unitarias hasta las pruebas funcionales y de capacidad. Diferentes equipos han mencionado que hay una falta de documentación formal sobre las estrategias de gestión de datos de pruebas y la gestión de las herramientas de automatización de pruebas.

7.3. RESUMEN DE LOS PROBLEMAS

En la Tabla 9 se presenta un resumen de los problemas encontrados, tanto en la literatura académica como en la industria, y su severidad de acuerdo a la cantidad de artículos científicos que lo indicaban o a la cantidad de respuestas dadas en la industria. Como puede verse, hay algunos problemas detectados en alguno de los dos ámbitos de análisis que no son evidentes en el otro.

1. Los localizadores de elementos de la interfaz de usuario son sentencias de código en las pruebas que permiten interactuar con componentes visibles en la interfaz, como por ejemplo un botón o un formulario.

Tabla 9. Relación entre los problemas encontrados en la literatura y en la industria y su severidad

| PROBLEMA | LITERATURA | INDUSTRIA | SEVERIDAD |
|--|-------------------|------------------|------------------|
| Pruebas no deterministas | Presente | Presente | Muy alta |
| Pruebas que consumen mucho tiempo de ejecución | Presente | Presente | Alta |
| Pruebas automatizadas de interfaz gráfica de usuario | Presente | Presente | Alta |
| Pruebas en Big Data | Presente | Presente | Alta |
| Pruebas en dispositivos móviles | Presente | Presente | Media |
| Falta de procedimientos, patrones y buenas prácticas para pruebas automatizadas en desarrollo continuo | Faltante | Presente | Media |
| Resultados de ejecución de pruebas ambiguos | Presente | Presente | Baja |
| Pruebas automatizadas de web con contenido dinámico | Presente | Presente | Baja |
| Pruebas de servicios web | Presente | Presente | Baja |
| Pruebas de datos | Presente | Presente | Muy baja |
| Pruebas automatizadas de requerimientos no funcionales | Presente | Presente | Muy baja |
| Pruebas de aplicaciones compuestas por servicios en la nube | Presente | Presente | Muy baja |
| Pruebas como un servicio | Presente | Faltante | Muy baja |
| Ambientes inestables | Faltante | Presente | Muy baja |



Capítulo 8. Soluciones en las pruebas de desarrollo continuo

Diferentes tipos de soluciones han sido propuestas para resolver los problemas relacionados a las pruebas en los proyectos de desarrollo continuo, presentados en este libro en la Tabla 9 del capítulo 7. A continuación, se analizan esas soluciones para cada problema con mayor detalle.

8.1. PRUEBAS NO DETERMINISTAS



Una característica importante de una prueba automatizada es su determinismo. Esto significa que una prueba siempre debe tener el mismo resultado cuando el código probado no cambia. Las pruebas automatizadas de este tipo ralentizan el progreso, no son confiables, esconden errores reales y aumentan el costo de mantenimiento. La estrategia general para abordar estos problemas es la priorización de pruebas y las técnicas de selección de pruebas (Elbaum, Rothermel y Penix, 2014). La idea subyacente es encontrar las pruebas que fallan e ir descartándolas o volver a priorizarlas para disminuir el tiempo de mantenimiento.

Martin Eyl (2016) propone un marco de trabajo que ejecuta solo pruebas relacionadas con una nueva característica o una funcionalidad modificada. De esta manera, el número de pruebas no deterministas se reduce significativamente. Del mismo modo, Busjaeger y Xie (2016) proponen seleccionar y priorizar las pruebas utilizando diferentes técnicas. Por un lado, la cobertura del código fuente que ha sido modificado; por otro lado, la similitud textual entre las pruebas relacionadas con el cambio y, finalmente, el historial reciente de las fallas. Cuando este enfoque encuentra fallas que no están relacionadas con el código nuevo o modificado en términos de cobertura o similitud de texto, significa que la prueba ejecutada es del tipo no determinista.

Las pruebas automatizadas también pueden ser analizadas para detectar si son deterministas o no (Laukkanen, Itkonen y Lassenius, 2017). Por ejemplo, Luo *et al.* (2014) proponen un mecanismo para la clasificación y el análisis de pruebas no deterministas estudiando sus causas comunes. El objetivo de los autores es identificar enfoques que puedan revelar un comportamiento inestable y describir estrategias comunes utilizadas

por los desarrolladores para corregir esta inestabilidad. Sin embargo, las «pruebas de pruebas» introducen esfuerzo y tiempo.

Otra estrategia común es volver a ejecutar las pruebas. Google, por ejemplo, tiene un sistema que recopila todas las pruebas que fallan durante el día volviéndolas a ejecutar por la noche (Penix, 2012). También usa un sistema de notificación que enumera el historial de ejecución de la prueba que ha fallado, facilitando el análisis. Si el nivel de inestabilidad de una prueba es demasiado alto, el sistema de monitoreo pone en cuarentena automáticamente la prueba y comunica el error a los desarrolladores.

Finalmente, Eloussi (2015) evalúa y propone enfoques para determinar si una falla o error al ejecutar una prueba se debe a un falso positivo. En esos casos recomienda:

- Posponer la reejecución de la prueba hasta el final de la ejecución del lote completo de pruebas. En este momento hay más información disponible y las repeticiones se pueden evitar por completo.
- Volver a ejecutar las pruebas en un entorno diferente.
- Revisar la cobertura de las pruebas con el último cambio. Podrá afirmarse que una prueba es no determinista si ha sido ejecutada con éxito en la revisión anterior y falla en una nueva, de tal manera que este fallo no depende de los últimos cambios.

De acuerdo con la literatura actual y la experiencia en la industria, las soluciones para las pruebas no deterministas tienen ventajas y desventajas (ver Tabla 10), y todavía no existe una solución amplia aceptada para esta problemática.

Tabla 10. Ventajas y desventajas de las soluciones para las pruebas no deterministas

| SOLUCIÓN | VENTAJAS | DESVENTAJAS |
|--|---|---|
| Priorización y selección de pruebas. | 1. Reduce el número de pruebas no deterministas en la ejecución del lote de pruebas. | 1. Las pruebas no deterministas siguen existiendo. 2. Las pruebas no deterministas no son identificadas. |
| Ejecución de las pruebas solo para verificar la funcionalidad afectada por el código modificado. | 1. Las pruebas no deterministas son fáciles de identificar e ignorar. | 1. Las pruebas no deterministas siguen existiendo. |
| Probar las pruebas. | 1. Las pruebas no deterministas pueden ser identificadas e ignoradas. 2. Es posible determinar la causa de la inestabilidad de las pruebas. 3. Las pruebas no deterministas pueden ser eliminadas o solucionadas. | 1. Costo y tiempo. |
| Volver a ejecutar las pruebas si fallan. | 1. Reduce la cantidad de fallos producidos por pruebas no deterministas. | 1. Tiempo. 2. Las pruebas no deterministas siguen existiendo. |
| Volver a ejecutar las pruebas si fallan, pero solo al final de la ejecución de todo el lote. | 1. Reduce la cantidad de fallos producidos por pruebas no deterministas. 2. Es posible determinar la causa de la inestabilidad de las pruebas. | 1. Tiempo. 2. Las pruebas no deterministas siguen existiendo. |

8.2. PRUEBAS QUE CONSUMEN MUCHO TIEMPO DE EJECUCIÓN

En cualquier entorno de desarrollo continuo de software, los cambios se introducen con alta frecuencia en el repositorio, por lo que es necesario ejecutar regresiones lo más rápido posible. Sin embargo, la ejecución de un lote de casos de prueba grande llevaría horas o días, incluso estando automatizados. A continuación, se analizan las diferentes soluciones agrupándolas por nivel de prueba.

8.2.1. Pruebas unitarias

Algunos autores, como por ejemplo Alshraideh (2008) y Tsai *et al.* (2011), han propuesto el uso de técnicas de generación automática de casos de prueba para enfrentar este problema. Al tener un sistema completamente automatizado de generación de casos de prueba unitarios, es posible reducir considerablemente el costo de las pruebas de software y también facilita la escritura de los casos. Por otro lado, Campos *et al.* (2015) proponen

un sistema automático de generación de casos de prueba para lenguajes orientados a objetos utilizando pruebas basadas en la búsqueda y un mecanismo llamado generación de pruebas continuas. Para este propósito, se presenta una herramienta llamada EvoSuite. Sin embargo, los autores describen que no es aplicable para clases internas y tipos genéricos.

Además, Tsai *et al.* (2011) presentan una técnica de priorización de pruebas unitarias, donde los casos de prueba se pueden clasificar. No obstante, el desarrollador debe hacer la priorización manualmente.

Madeyski y Kawalerowicz (2013) proponen un mecanismo que consiste en ejecutar pruebas unitarias en segundo plano, mientras el desarrollador está codificando. David Staff (2004) presenta un complemento del entorno de desarrollo Eclipse para proyectos Java, que compila automáticamente el código fuente al ser guardado e indica errores de compilación en el editor de texto del entorno de desarrollo.

Finalmente, Eyl, Reichmann y Müller-Glaser (2016) mejoran este enfoque al agregar una estrategia orientada a la selección de pruebas. Cada módulo tiene su propio complemento y no es necesario ejecutar todas las pruebas unitarias, sino solo las relacionadas con el módulo afectado. Mediante el uso de convenciones de nomenclatura, el marco de trabajo propuesto puede encontrar el complemento de prueba para un módulo específico y las herramientas de cobertura de código pueden detectar clases modificadas. Por lo tanto, este enfoque impacta en los tiempos de ejecución de las pruebas unitarias, reduciéndolos, y no impacta en la etapa de escritura del caso de prueba.

En la Tabla 11 se muestra un resumen de las soluciones propuestas para las pruebas unitarias que requieren mucho tiempo de ejecución. En la primera columna se incluye el título de la solución; en la segunda columna, las referencias donde se implementa y en la tercera columna, el grado de cobertura de la solución que podrá ser parcial o total.

Tabla 11. Soluciones para pruebas unitarias que requieren mucho tiempo de ejecución

| SOLUCIÓN | ARTÍCULOS QUE IMPLEMENTARON LA SOLUCIÓN | GRADO |
|---|--|--------------|
| Generación de casos de prueba. | Alshraideh (2008) Campos <i>et al.</i> (2015) | Parcial |
| Priorización de casos de prueba. | Tsai <i>et al.</i> (2011) | Parcial |
| Ejecución de pruebas unitarias en segundo plano mientras se desarrolla el código. | Madeyski y Kawalerowicz (2013) | Parcial |
| Ejecución de grupos de pruebas unitarias seleccionadas, en segundo plano, mientras se desarrolla el código. | Eyl, Reichmann y Müller-Glaser (2016) | Total |

8.2.2. Pruebas funcionales

En la literatura se proponen técnicas de segmentación y agrupación de casos de prueba para disminuir los tiempos de ejecución de prueba funcionales elevados (Laukkanen, Itkonen y Lassenius, 2017; Sinisalo, 2016; Tsai *et al.*, 2011). Las pruebas se agrupan en diferentes lotes según la funcionalidad y la velocidad, de tal manera que las pruebas más críticas se pueden ejecutar primero y los desarrolladores obtienen retroalimentación rápida de ellas. Las pruebas no críticas y más lentas se ejecutan más tarde y solo si las primeras han pasado. Por lo tanto, la segmentación de prueba resuelve parcialmente el problema de pruebas funcionales que consumen mucho tiempo.

Otra alternativa para este problema es el uso de la paralelización en las pruebas automatizadas disminuyendo la cantidad de tiempo invertido. La ejecución de las pruebas se distribuye a través de diferentes computadoras, servidores o, en general, hilos de ejecución. Engblom (2015) propone el uso de la virtualización como una alternativa para tener una sola computadora y distribuir la ejecución de la prueba a través de máquinas virtuales.

También es posible reducir la cantidad de pruebas a ejecutar seleccionando solo las que están relacionadas con el código fuente modificado (Eyl, Reichmann y Müller-Glaser, 2016). Uno de los mecanismos que han implementado para este marco es la estrategia de selección de pruebas orientada a los requisitos.

Otra técnica de selección es la basada en el análisis de correlaciones entre fallas de casos de prueba y los cambios en el código fuente (Knauss *et al.*, 2015). Sin embargo, ninguno de los dos enfoques ha resuelto el problema de los cambios que tienen un efecto en todo el sistema.

Otros autores proponen técnicas de priorización automática de casos de prueba para enfrentar el problema de los tiempos de ejecución elevados. Una de las estrategias es la priorización basada en datos históricos para determinar un orden óptimo de pruebas de regresión en las ejecuciones de pruebas posteriores. Finalmente, Marijan y Liaen (2017) presentan una herramienta que ejecuta las pruebas que toman menos tiempo que antes.

En la literatura se implementan también otras alternativas, en Tilley y Floss (2014) se propone un enfoque llamado «pruebas continuas de larga duración» que utiliza métodos de inteligencia artificial. El enfoque consiste en ejecutar casos de prueba todo el tiempo en un servidor de compilación y detectar problemas mediante técnicas de inteligencia artificial.

Distintos autores proponen el uso de diferentes navegadores como una alternativa para aumentar la velocidad en la ejecución de pruebas de interfaz web. Rotar los navegadores entre ejecuciones consecutivas puede lograr gradualmente la misma cobertura que ejecutar las pruebas en cada navegador para cada ejecución (Sinisalo, 2016).

Por otro lado, Engblom (2015) afirma que ejecutar pruebas automatizadas en paralelo es una de las mejores soluciones para pruebas que requieren mucho tiempo, pero esto suma la necesidad de recursos de hardware. Una de las principales ventajas de la estrategia de pruebas como servicio o TaaS sobre las pruebas tradicionales es su modelo escalable a través de la nube: utiliza potencia informática, espacio en disco y memoria según los requisitos actuales, pero tiene la capacidad de aumentar la demanda muy rápidamente.

En la Tabla 12 se muestra un resumen de las soluciones propuestas para las pruebas funcionales que requieren mucho tiempo.

Tabla 12. Soluciones para pruebas funcionales que requieren mucho tiempo de ejecución

| SOLUCIÓN | ARTÍCULOS QUE IMPLEMENTARON LA SOLUCIÓN | GRADO |
|---|---|--------------|
| Agrupamiento o selección de casos de pruebas. | Tsai <i>et al.</i> (2011) Sinisalo (2016) Laukkanen, Itkonen y Lassenius (2017) | Parcial |
| Paralelización de pruebas. | Tsai <i>et al.</i> (2011) Rossi <i>et al.</i> (2016) Sinisalo (2016) Laukkanen, Itkonen y Lassenius (2017) | Total |
| Selección automática de casos de pruebas. | Eyl, Reichmann y Müller-Glaser (2016) Knauss <i>et al.</i> (2015) | Parcial |
| Priorización automática de casos de pruebas. | Elbaum, Rothermel y Penix (2014) Marijan y Liaaen (2017) | Parcial |
| Selección y priorización automática de casos de pruebas. | Busjaeger y Xie (2016) | Parcial |
| Ejecución de pruebas continuamente en un servidor. | Burgin y Debnath (2010) | Parcial |
| Pruebas como un servicio (TaaS). | Tilley y Floss (2014) | Total |
| Selección y priorización automática de casos de pruebas, con ejecución de los mismos en paralelo utilizando TaaS. | Penix (2012) | Total |
| Optimización de casos de pruebas. | Marijan y Liaaen (2017) | Parcial |
| Rotación de navegadores. | Sinisalo (2016) | Parcial |
| Uso de APIs REST. | Sinisalo (2016) | Parcial |

8.2.3. Pruebas automatizadas de interfaz gráfica

Las pruebas de alto nivel como, por ejemplo, las pruebas de aceptación de interfaz gráfica de usuario, se realizan principalmente con prácticas manuales que a menudo son costosas, a la par que tediosas, lo cual puede generar descuidos y errores. Es en este sentido que, una vez más, la automatización de pruebas se ha propuesto como una alternativa para resolver estos problemas. Sin embargo, la interfaz de usuario cambia con muy alta frecuencia y puede conducir a pruebas automatizadas que produzcan falsos positivos, es decir, las descritas en la sección 8.1 de este capítulo. Por lo tanto, se propusieron varias soluciones para enfrentar estos desafíos.

En una tubería de despliegue continuo, donde las pruebas automatizadas se ejecutan muchas veces al día, la estabilidad de las pruebas es un factor clave para lograr un rendimiento aceptado. Una posible solución es el desarrollo de interfaces adicionales para acceder a la aplicación que se está probando mediante una API. Los pasos que se ejecutan en la etapa de precondiciones se pueden realizar utilizando API y solo la característica particular a probar se realiza a través de la GUI. Esta solución reduce la cantidad de pasos de pruebas innecesarios en la GUI (Gmeiner, Ramler y Haslinger, 2015). Otros autores proponen el mismo enfoque: «los pasos de configuración se pueden ejecutar mediante el uso de las API REST (si es posible) para hacerlos más confiables y así reducir fallas innecesarias» (Sinisalo, 2016: 60).

Por otro lado, Smith (2001) presenta un enfoque llamado «pruebas de interfaz gráfica conducidas por elementos visuales» o VGT (*Visual GUI Testing*, por su sigla en inglés). Es una técnica de pruebas que utiliza el reconocimiento de imágenes para interactuar y afirmar la exactitud de un sistema bajo prueba a través del mapa de bits de la pantalla que se muestra al usuario en el monitor de la computadora. El uso del reconocimiento de imágenes permite que la técnica se use en cualquier sistema basado en interfaz de usuario, independientemente de su implementación o plataforma. La principal ventaja es que los cambios en el código de la interfaz de usuario no harán que la prueba falle. Sin embargo, el principal desafío es el reconocimiento de las imágenes de acuerdo con el grado de dinamismo de la interfaz.

8.2.4. Pruebas ambiguas

Los resultados de las pruebas deben comunicarse adecuadamente a los desarrolladores, indicando si han sido o no exitosas. En caso de un fallo, debe quedar en claro con la mayor precisión posible qué lo ha provocado. Cuando el resultado de una prueba no cumple con estos requisitos, entonces es un resultado ambiguo.

Una opción es utilizar el análisis de archivos de volcado de memoria (*dump files*) por fallos y archivos de registro para extraer resúmenes o detalles de fallas consistentes.

Otro enfoque lo presentan Eyl, Reichmann y Müller-Glaser (2016), con una solución que mejora la calidad de la retroalimentación con los resultados de la prueba al:

- ejecutar pruebas con la compilación del sistema en un entorno similar al entorno de producción;
- proporcionar solo resultados de pruebas para el código modificado del desarrollador;
- proporcionar información sobre si una prueba ha fallado debido al último cambio del desarrollador o debido a un cambio anterior.

La información enviada al usuario de las pruebas incluye:

- Datos sobre el conjunto de cambios introducidos que activaron la ejecución de las pruebas: registro de cambios, momento de la introducción de cambios, archivos introducidos, etc.

- Una descripción general de todas las pruebas ejecutadas con sus resultados (éxito o fallo). También se proporcionan datos sobre la frecuencia de fallo de la prueba.
- Direcciones a un sitio web construido de forma dinámica que enmarca y narra los resultados.
- Un informe de cobertura de código de todas las pruebas ejecutadas, para que el desarrollador pueda verificar si las pruebas realmente han ejecutado el código fuente modificado.

Igualmente es recomendable mejorar los mensajes de las pruebas cuando fallan y el nombre de las clases o métodos de prueba para mejorar la legibilidad de los resultados (Sinisalo, 2016). A veces, las pruebas fallan debido a una excepción lanzada desde objetos de página¹ (patrón de PageObject), y los mensajes de excepción de estos objetos de página van desde mensajes personalizados hasta las excepciones detalladas de las herramientas que automatizan la interfaz de usuario. Estos resultados a menudo revelan la causa raíz como, por ejemplo, cierto elemento HTML que no se encontró en la página. Sin embargo, la persona que realiza el análisis de las pruebas que fallan puede no ser capaz de reconocer el elemento por el selector de elementos web que se menciona en el error. Por lo tanto, agregar mensajes personalizados a las excepciones puede servir mejor a la claridad si son lo suficientemente precisos.

Finalmente, la ambigüedad de los resultados se puede mejorar mediante el uso de informes, teniendo en cuenta las siguientes características.

- La causa de la falla se describe de manera precisa.
- Solo se muestra el estado de las pruebas relacionadas con el cambio del desarrollador.
- No se muestran los resultados de pruebas no deterministas.
- Se presentan mensajes personalizados en lugar de excepciones o volcados de la pila de memoria.
- Se proporcionan direcciones a páginas web y capturas de pantalla donde las pruebas han fallado.



1. PageObject es un patrón de diseño que se utiliza en pruebas automatizadas para evitar código duplicado y mejorar el mantenimiento de las mismas.

Capítulo 9. Uso de prácticas del desarrollo continuo de software en la industria

En este capítulo se analizan las respuestas del estudio llevado adelante por el Grupo de Investigación en Calidad de Software de la Universidad Nacional del Nordeste en 2019, para determinar cómo se implementan las mejores prácticas del despliegue y las pruebas continuas en la industria del software.

La encuesta fue realizada, como se describió en el capítulo 8, a empresas ubicadas principalmente en los Estados Unidos, Canadá, Inglaterra, Alemania, Suiza, Argentina, Brasil, Chile, Irlanda y China. En total, se obtuvieron 255 respuestas válidas, y los proyectos fueron de desarrollo de aplicaciones web de aerolíneas, sitios de comercio electrónico, sitios de agencias de viajes, motores de búsqueda, compañías de software de relaciones públicas, compañías de seguros, bancos y portales de noticias.



9.1. BUENAS PRÁCTICAS EN EL DESARROLLO DE LAS PRUEBAS

La primera parte de la encuesta se dedicó a buenas prácticas por el desarrollador de forma local, donde se consultó si los miembros del equipo están construyendo y probando el código correctamente antes de integrarlo en la rama principal del repositorio de control de versiones, así como la frecuencia de estas integraciones. Los resultados se presentan a continuación.

Como puede verse en la Figura 28, el 6% de los equipos encuestados manifestó que ocasionalmente compilan el código antes de integrarlo a la rama principal en el repositorio de control de versiones, ya que los cambios pequeños se integran directamente. Todos afirmaron que solo compilan o construyen el código fuente cuando los cambios en el mismo son muy grandes.



Figura 28. Construcción del código antes de integrarlo a la rama principal.

El caso de las pruebas unitarias es bastante similar: el 12% de los equipos declaró que solo ejecutan pruebas unitarias localmente –o en una rama separada– cuando los cambios en el código son grandes. De igual manera, el 15% de los equipos no ejecuta las pruebas unitarias (ver Figura 29).

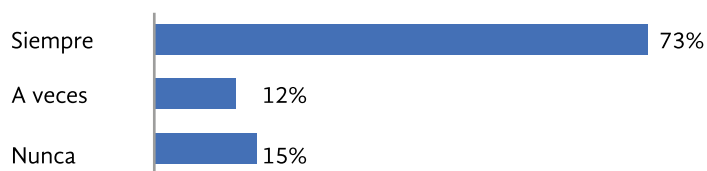


Figura 29. Ejecución de pruebas unitarias antes de integrar el código a la rama principal.

Respecto de las pruebas de aceptación, existe un valor significativo de un 58% que las ejecuta solamente una vez, hacia el final del proyecto (ver Figura 30). El problema principal es el tiempo que lleva, especialmente para realizar las pruebas de regresión asociadas con los cambios de funcionalidad.

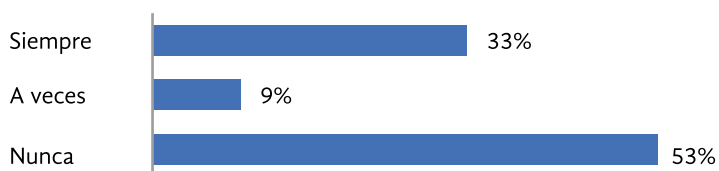


Figura 30. Ejecución de pruebas automatizadas antes de integrarlas a la rama principal.

Según los equipos preguntados, ejecutar pruebas automatizadas lleva mucho tiempo y deciden hacerlo solo cuando se trata de un cambio de código muy importante. Otros proyectos ejecutan pruebas automatizadas específicas, relacionadas con la funcionalidad que se está modificando, pero depende del tiempo disponible y si esas pruebas automatizadas relacionadas pueden determinarse.

En la Figura 31 puede verse la frecuencia de integración del código fuente en los equipos de trabajo. Teniendo en cuenta que una integración continua debería ser por lo menos diaria, solamente el 47% de los proyectos lo cumple. Y de manera estricta solamente un 25% integra con mayor frecuencia.

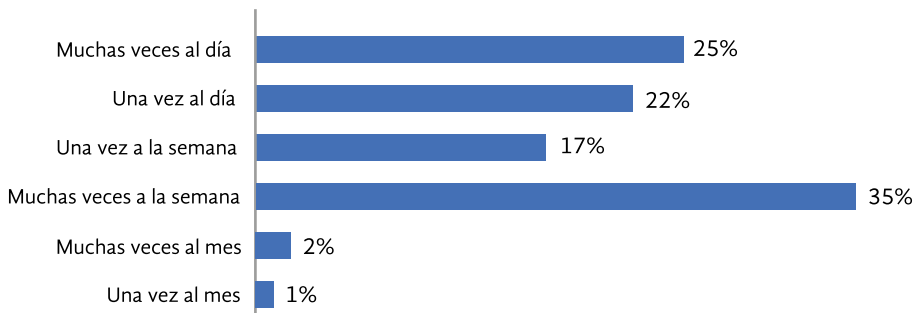


Figura 31. Frecuencia de las integraciones a la rama principal.

Como se mencionó anteriormente, todos los proyectos utilizan una o más prácticas de desarrollo continuo. Por lo tanto, el código sigue un curso compuesto de diferentes etapas que implican construir, probar y desplegar el software. Lo que pudimos encontrar en la encuesta se muestra en la Tabla 13, con las diferentes etapas utilizadas por los proyectos en sus flujos y la manera en que se activan.

Con respecto a los modos de activación para realizar la integración, los equipos de trabajo manifestaron tener por lo menos dos flujos de trabajo: uno para la integración y construcción del código base con pruebas unitarias y otro utilizado para las pruebas de aceptación automatizadas. Esto denota la poca frecuencia de ejecuciones en las pruebas de aceptación respecto del total de integraciones visto en la Figura 31. Además, hay algunos proyectos que activan el flujo de integración automáticamente con el solo hecho de crear solicitudes de integración de código o *pull requests*. Estos proyectos representan las respuestas pertenecientes a la categoría «otro». La Tabla 13 muestra los detalles de los modos de activación de flujo de integración de código.

Tabla 13. Modo de activación para flujos de integración por proyecto

| <u>MODO DE ACTIVACIÓN PARA FLUJOS DE CONSTRUCCIÓN Y PRUEBAS UNITARIAS</u> | <u>MODO DE ACTIVACIÓN PARA FLUJOS DE PRUEBAS DE ACEPTACIÓN AUTOMATIZADAS</u> | <u>RESPUESTAS</u> |
|---|--|-------------------|
| Automatizado | Automatizado | 67% |
| Programado | Programado | 43% |
| Manual | Manual | 75% |
| Automatizado | Programado | 5% |
| Automatizado | Manual | 24% |
| Programado | Manual | 32% |
| Otro | Otro | 9% |

Las etapas más utilizadas por los equipos de trabajo son la construcción y las pruebas unitarias. En segundo término, las pruebas funcionales que pueden automatizarse y el despliegue. Las inspecciones del código fuente y las pruebas manuales no son tan

frecuentes, al igual que la implementación de notificaciones. En la Figura 32 se presentan los resultados en la secuencia natural que puede darse en una tubería de despliegue continuo de software.

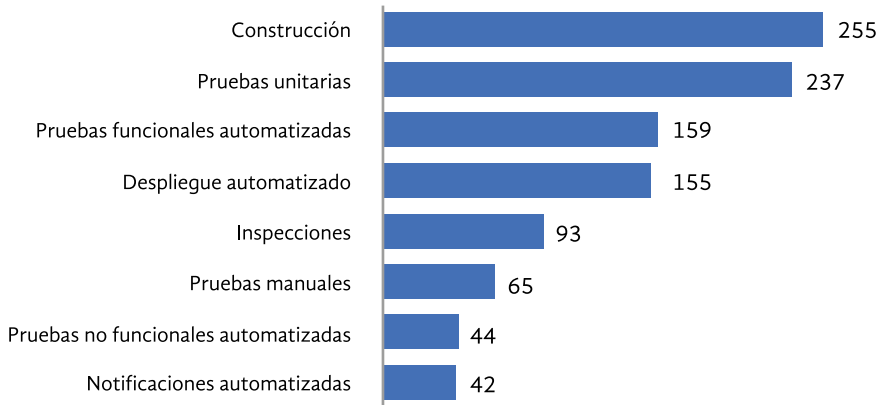


Figura 32. Etapas identificadas en la integración de los proyectos.

Como se ha mencionado en otros capítulos, la integración continua es uno de los componentes clave de cualquier práctica continua de desarrollo de software. En el libro *Integración Continua, de Duvall, Matyas y Glover (2007)*, se menciona un conjunto de buenas prácticas para implementar la integración continua de manera adecuada. A continuación, se indica qué tanto estos equipos de trabajo utilizan las buenas prácticas de integración continua de Duvall (ver la Figura 33).

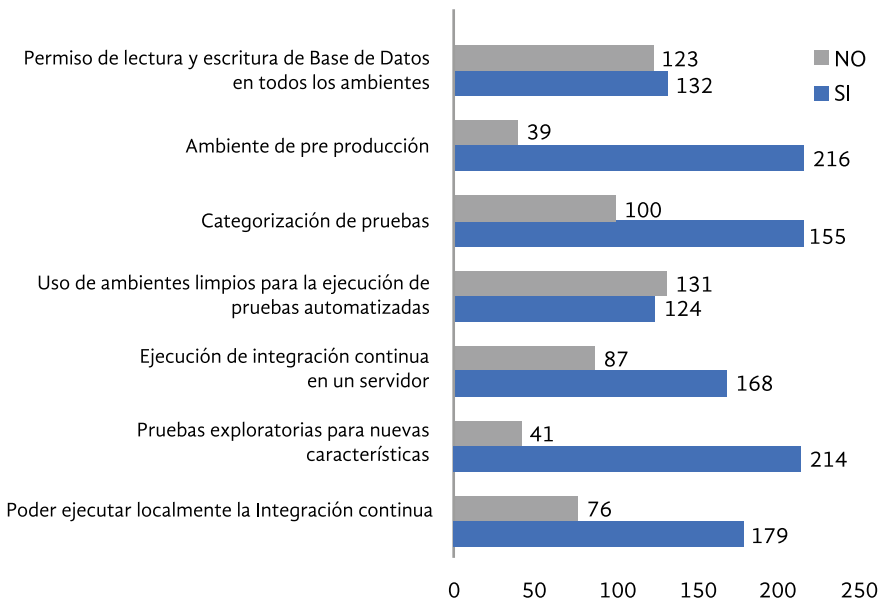


Figura 33. Uso de las buenas prácticas de integración continua.

Como primera buena práctica, todos los desarrolladores deben llevar a cabo compilaciones y construcciones privadas en sus estaciones de trabajo locales antes de enviar su código al repositorio de control de versiones. Eso asegurará que sus cambios no causarán errores en el flujo de integración. Para hacerlo, deberían ejecutar localmente el mismo script que ejecutan en el servidor de integración continua. Respecto de los resultados de la encuesta, un 70% de los equipos tiene la capacidad mencionada anteriormente. Además, se debe utilizar una máquina de integración separada para ejecutar las etapas de este flujo de integración; esto es realizado por un 65% de los equipos.

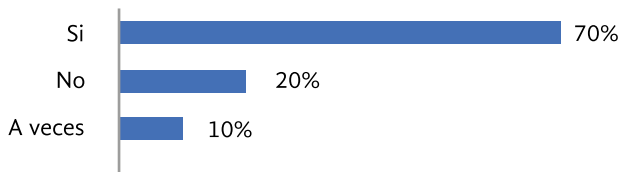
Con respecto a las buenas prácticas de control de calidad, el 84% de los proyectos está ejecutando pruebas exploratorias para nuevas características y el 61% categoriza las pruebas. Asimismo, es importante contar con un entorno similar a producción para las pruebas, de modo que los riesgos relacionados con la introducción de errores en producción sean bajos. La mayoría de los proyectos, más de un 90%, declaró que tiene un entorno similar a producción.

Según Humble y Farley (2010), para implementar el despliegue continuo, la colaboración entre los diferentes miembros del equipo es una de las claves del éxito. Los resultados de la encuesta mostraron que casi la mitad de los proyectos tienen desarrolladores o especialistas en pruebas sin permisos de lectura y escritura para las bases de datos (48%).

Respecto de la práctica de manejo de fallas en los flujos de integración continua, Martin Fowler afirma que deben repararse de inmediato y que la estrategia más rápida es quitar los últimos cambios y llevando al sistema de vuelta a la última versión estable (Fowler y Foemmel, 2006). Los resultados de la encuesta, vistos en la Figura 34, muestran que el 70% de los proyectos comparte esta filosofía, donde la tarea de mayor prioridad es arreglar un error en el flujo del servidor de integración continua. Al mismo tiempo, otro 10% manifiesta que lo soluciona de inmediato, dependiendo de la situación. Del mismo modo, en cuanto a las pruebas automatizadas, un 60% indica que se solucionan los fallos que se presentan en esa etapa. Finalmente, el 55% de los equipos utiliza el mecanismo de vuelta atrás o *roll back*.

Todos expresaron que, si se trata de un problema crítico, revertirán los cambios en el código, de lo contrario, lo ignorarán. La mayoría de ellos declaró que simplemente lo reportan como un error o problema en sus herramientas de seguimiento de incidentes, para que otro equipo o persona pueda solucionarlo más tarde.

Cuando hay una falla: se vuelve a la última versión estable



Se solucionan los fallos en el momento que ocurren



No se deja continuar la integración y se vuelve atrás

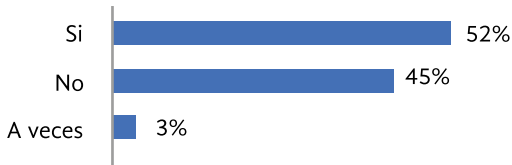


Figura 34. Prácticas de manejo de fallas en los flujos de integración continua.

Las siguientes preguntas fueron relativas a las prácticas de despliegue continuo automático utilizadas. En este sentido, solo el 6% realiza el despliegue automático, mientras que el 15% lo hace presionando un botón, le sigue el 20% que ejecuta un conjunto de scripts manuales y el 8% final que lo hace completamente manual.

El subgrupo del 51% restante se divide entre un 21% que declaró que la implementación es realizada por un equipo diferente y un 79% con miembros experimentados que son los únicos capaces de realizar esta tarea (ver Figura 35).

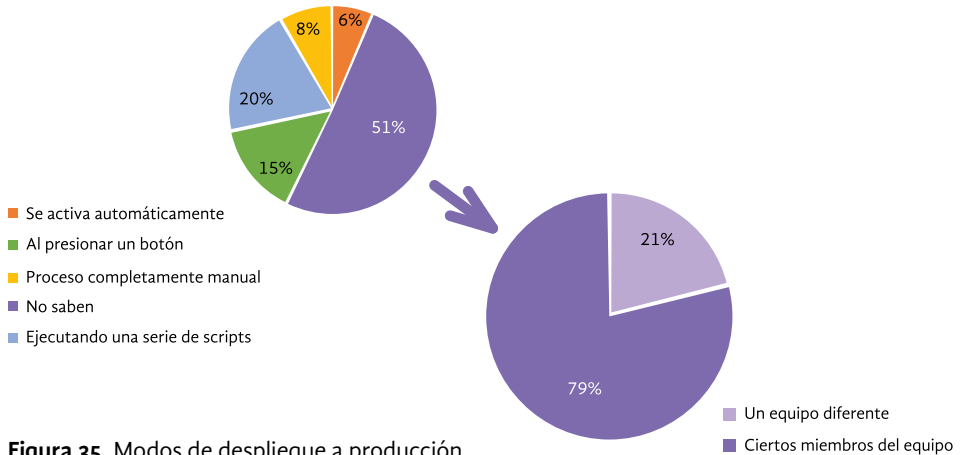


Figura 35. Modos de despliegue a producción.

Respecto de la gestión de los activos del software, Martin Fowler afirma que es necesario colocar en el sistema de control de versiones todo lo necesario para construir el producto software con un solo comando (Fowler y Foemmel, 2006). Esto significa que todos los activos de software deben ser cohesivos, verificables funcionalmente y ser accedidos desde un repositorio de control de versiones, para que la construcción del software se pueda llevar a cabo a través de un solo comando. La Figura 36 muestra los activos que están almacenados en el repositorio de control de versiones de cada proyecto. El código principal de la aplicación, los scripts de pruebas automatizadas y los archivos de configuración son los activos que más se almacenan en los repositorios de control de versiones. El 53% de los equipos almacena en el control de versiones todos sus activos, 39% hace hincapié en el código fuente, scripts de base de datos y código ejecutable y, finalmente, el 8% no mantiene los activos etiquetados en el control de versiones.

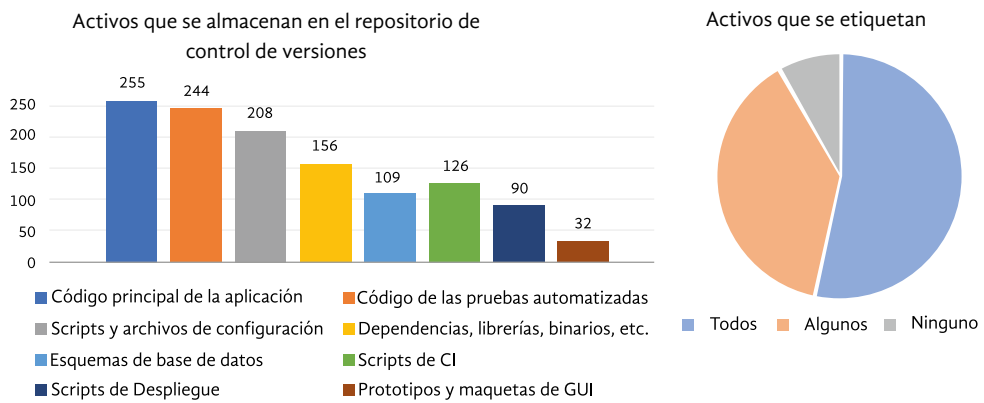


Figura 36. Gestión de los activos en un sistema de control de versiones.

Respecto de las etapas de prueba durante la integración continua, en la Figura 37 se muestran los niveles de pruebas que fueron automatizados por los equipos de trabajo y el número de proyectos que crea pruebas automatizadas de sus defectos para evitar que vuelvan a aparecer en el futuro.

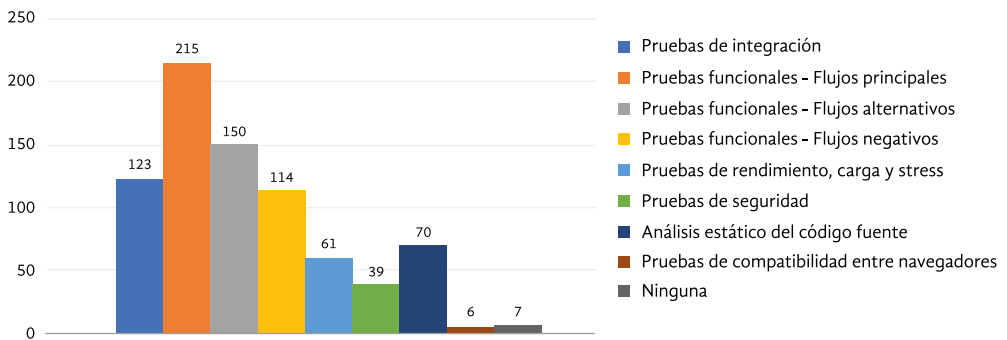


Figura 37. Niveles de prueba automatizados.

Aunque la mayoría de los niveles y tipos de prueba están automatizados, hay muchos equipos que no integran esos scripts de prueba automatizados en su flujo de despliegue. La Tabla 14 muestra cuántos proyectos han automatizado un cierto tipo de nivel de prueba, pero sin incluirlo en la tubería.

Tabla 14. Niveles de pruebas automatizados por proyecto

| NIVELES | PORCENTAJE DE PROYECTOS |
|---|-------------------------|
| Pruebas de integración | 19 |
| Pruebas funcionales-Flujos principales | 17 |
| Pruebas funcionales-Flujos alternativos | 12 |
| Pruebas funcionales-Flujos negativos | 13 |
| Pruebas de rendimiento, carga y stress | 17 |
| Pruebas de seguridad | 14 |
| Análisis estático del código fuente | 9 |

Para descubrir por qué los proyectos no incluyen las pruebas automatizadas mencionadas en la Tabla 14, se les pidió a los equipos que describieran las razones. Las principales afirmaciones fueron las siguientes: falta de conocimiento, falta de experiencia, falta de tiempo, falta de herramientas y ambientes inestables. Las mismas se muestran en la Figura 38.

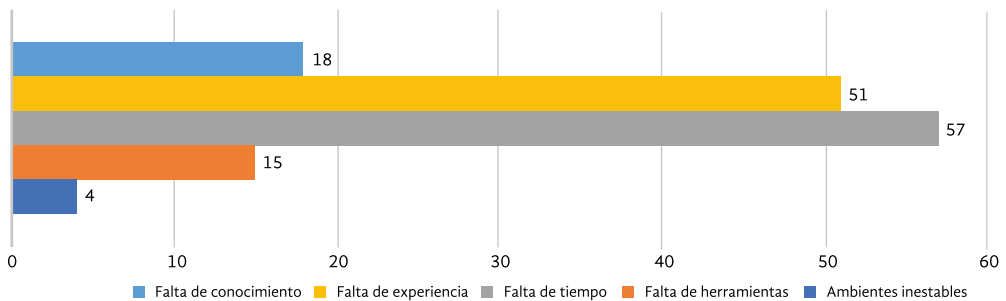


Figura 38. Causas por las que no se incorporan las pruebas automatizadas al flujo de despliegue.

La mayoría de los equipos de trabajo justificó no agregar las pruebas automatizadas debido a la falta de tiempo y experiencia.

Con respecto a la automatización de casos de pruebas para defectos encontrados, se puede ver una división entre los que no lo tienen en cuenta, un 53%, y los que siempre o muchas veces lo hacen (ver Figura 39).

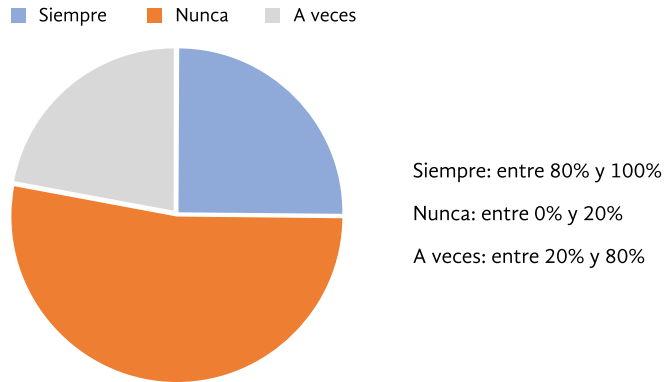


Figura 39. Automatización de pruebas para defectos encontrados.

De los equipos que en ocasiones han automatizado los casos de pruebas de los defectos encontrados, todos declararon que solo desarrollan scripts para errores de alta gravedad.

Según Duvall (2007), realizar inspecciones es otra buena práctica que funciona bien para los equipos que ejecutan integración continua en un proyecto de desarrollo de software. Es importante automatizar esta práctica incorporando, cuando sea posible, herramientas de análisis estático de código fuente. Al realizar las inspecciones, existen muchos atributos del código fuente que pueden medirse como, por ejemplo, el código duplicado, la cobertura de las pruebas unitarias o el acoplamiento. En la Figura 40 se presentan los atributos que miden los proyectos. Aquí puede verse que las principales métricas tienen que ver con aspectos del tamaño y la complejidad del código fuente o con características de estilo.

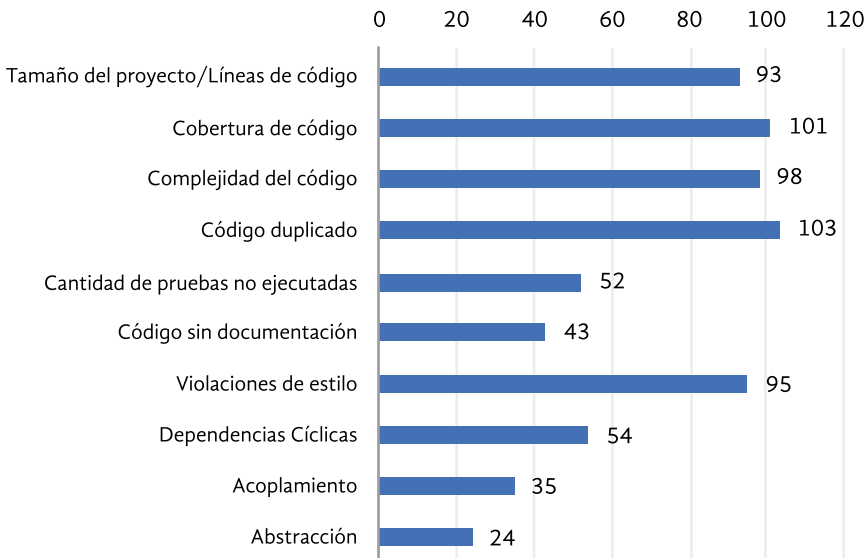


Figura 40. Principales atributos del código fuente que se miden en las inspecciones.

La generación de datos de prueba es uno de los métodos más recomendados en la gestión de datos de pruebas para implementar un proceso de pruebas en el despliegue continuo de software (Humble y Farley, 2010). La Figura 41 muestra los diferentes mecanismos de gestión de datos de prueba implementados por los equipos de desarrollo entrevistados.

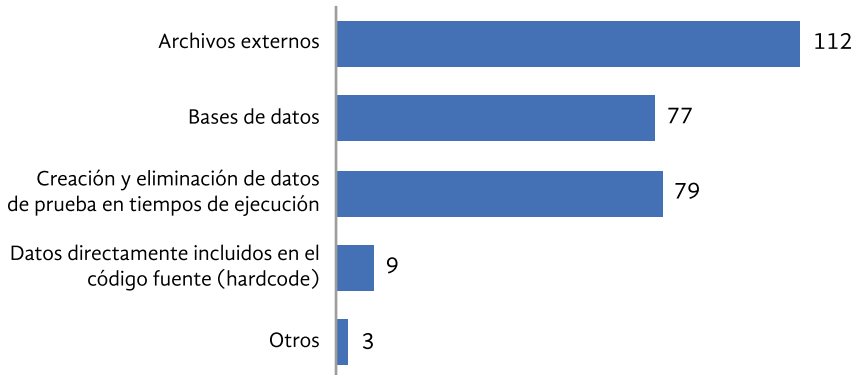


Figura 41. Gestión de los datos de pruebas.

En este sentido, un 40% de los equipos manifestó que usan fuentes externas como hojas de cálculo, archivos en texto plano o estructurado (xml, json, etcétera). Aproximadamente, un tercio ha implementado un mecanismo de generación de datos de prueba que los prepara, utiliza y elimina en cada ejecución. Otro tercio, en cambio, almacena los datos permanentemente en una base de datos. Existe también un pequeño porcentaje que ha introducido los datos de prueba directamente en el código fuente y a esta práctica se la conoce como *hardcoding*¹.

Finalmente, otros tres proyectos utilizan un mecanismo híbrido de trabajo: se ejecutan las pruebas que buscarán los datos en un almacenamiento persistente; si no están en la base de datos, los crea y los almacena. Una vez que finaliza la ejecución, los datos de prueba no se eliminan.

9.2. COLABORACIÓN EN EL EQUIPO DE TRABAJO

La colaboración entre los miembros de equipo es uno de los pilares para lograr prácticas de desarrollo continuo, lo que se contrapone a la forma de trabajo tradicional (Cohn, 2009).

Por un lado, los especialistas en pruebas (también llamados *testers*) deben trabajar con por lo menos otros tres roles en el proyecto: con el analista para definir los criterios de aceptación de los requisitos del cliente, con los desarrolladores para ayudarlos a

1. *Hardcoding* consiste en una «mala práctica» en la que se introducen datos directamente en el código fuente del programa, en lugar de obtenerlos de una fuente externa o de parámetros recibidos.

comprender estos requisitos desde el principio y con los encargados de automatizar las pruebas para escribir las pruebas de aceptación automatizadas.

Por otro lado, los desarrolladores deberían ayudar a los *testers* a probar la aplicación cuando sea necesario. Las pruebas automatizadas también deben agregarse como parte de la «definición de hecho»² y deben desarrollarse antes de que finalice la iteración, para que puedan incluirse en los lotes de regresión que se ejecutarán cuando se incluyan nuevas características. En la Figura 42 se muestra la colaboración entre estos roles y los proyectos que han incluido pruebas automatizadas como requisito obligatorio.

Así, en el 55% de los equipos hay desarrolladores que ayudan a los *testers* a probar la aplicación cuando es necesario y un 25% que, por el contrario, no sigue esta práctica de colaboración. También existe un 12% de equipos en los que sus desarrolladores ayudan a los *testers* solo cuando la función a probar requiere habilidades de codificación.

Además, el 55% de los equipos tiene especialistas en pruebas que ayudan a los desarrolladores a comprender los requisitos, un 27% que no lo hacen y un 10% en donde los equipos tienen *testers* que ayudan a los desarrolladores solo cuando tienen tiempo libre. Finalmente, el 8% de los equipos manifestó que no tienen *testers* en sus equipos. Con respecto a los proyectos que incluyen las pruebas automatizadas como un elemento en la definición de hecho, un 29% está implementando esta práctica y lográndolo en cada iteración. Asimismo, hay otro 20% que incluye las pruebas automatizadas en la definición de hecho, pero la mayoría de las veces no lo pueden cumplir. Finalmente, un 51% no implementa esta práctica.

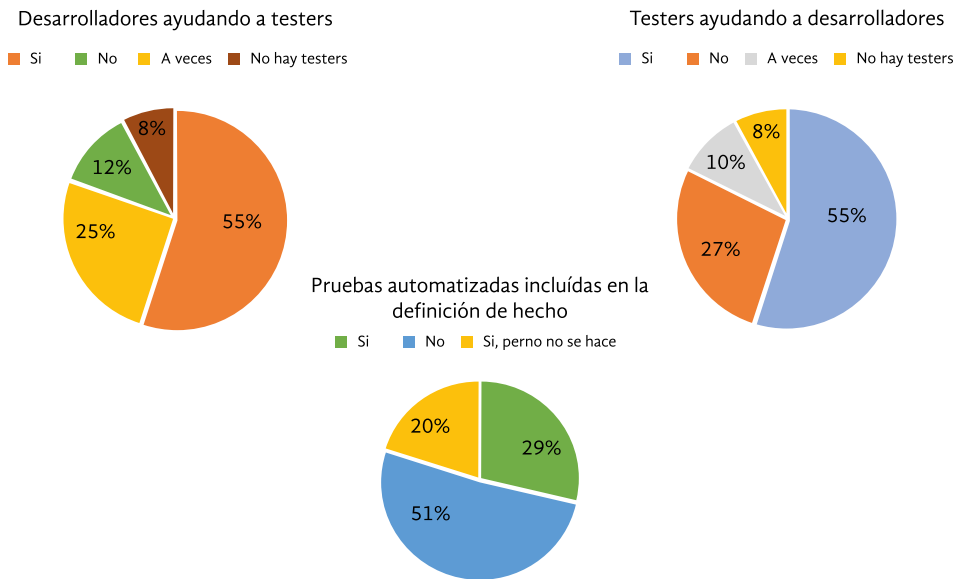


Figura 42. Colaboración entre los miembros del equipo.

2. La «definición de hecho» es un acuerdo común que debe existir entre los miembros de un equipo ágil para considerar una pieza de trabajo como terminada.

Capítulo 10. Automatización de las pruebas de software

En el capítulo 5 se presentó la clasificación de los niveles de prueba típicos en un ambiente de desarrollo con pruebas continuas de software. A continuación, se darán ejemplos para llevar a la práctica estos niveles en un entorno de desarrollo con tecnología Java. Las explicaciones no son una guía paso a paso, sino, más bien, una referencia. Tampoco agotan la temática en cuestión, que es cambiante y evoluciona a mayor velocidad que la escritura de un libro. En su lugar, al final del capítulo se enumera una serie extensa de materiales digitales libres e introductorios a cada tecnología ejemplificada. Le sugerimos al lector que avance incorporando los conocimientos prácticos en su ámbito profesional o personal con pasos iterativos e incrementales, utilizando la hoja de ruta trazada a continuación.



10.1. PRUEBAS UNITARIAS

En este nivel inicial, la etapa de verificación local consiste solamente en la escritura de pruebas unitarias para su posterior ejecución junto con la construcción y la compilación del código antes de integrarlo con el repositorio de control de versiones, ver más detalles en el Anexo, en «Control de versiones con Git». En el artículo de Rodríguez, Piattini y Ebert (2019) se listan muchas de las opciones que existen en la actualidad para implementar pruebas unitarias. La ejecución de este tipo de pruebas puede realizarse utilizando un comando por consola, como también mediante una herramienta en el entorno de desarrollo integrado del programador. Las pruebas son ejecutadas de forma secuencial, una después de la otra, y en caso de que una falle, no se interrumpe el resto de la ejecución.

En el capítulo anterior se enumeraron las buenas prácticas en cuanto a la frecuencia de ejecución de las pruebas unitarias. Existen dos momentos principales en los que deberían ejecutarse:

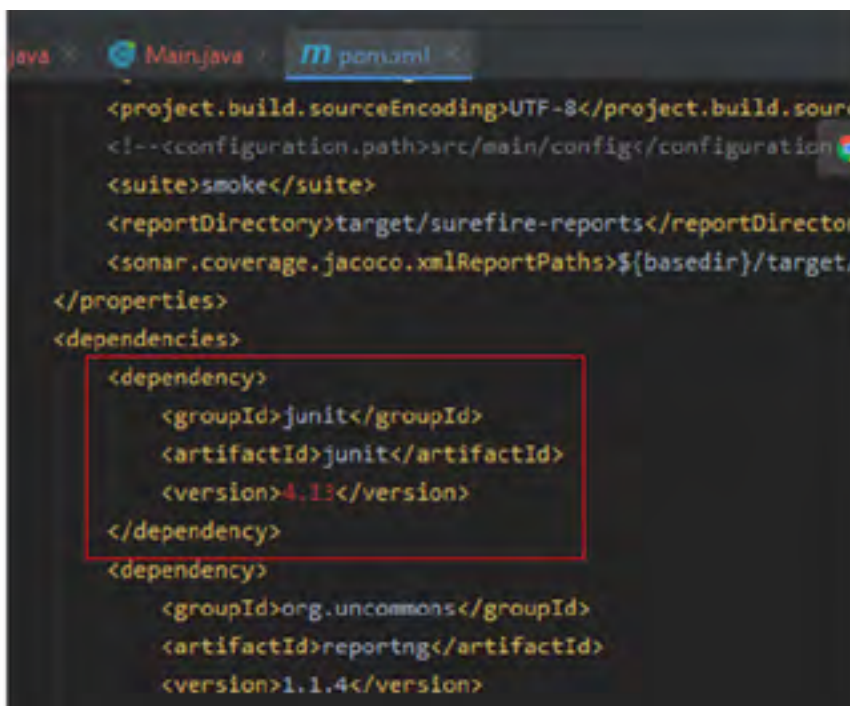
- Al momento de pasar los cambios del desarrollo desde la copia local del programador hacia la rama principal de desarrollo.
- Al momento de integrar los desarrollos del equipo, por ejemplo, al final del día.

Esta frecuencia dependerá de la cantidad y el tipo de pruebas. Al inicio, es normal ejecutarlas junto con cada compilación del código fuente.

10.1.1. Implementación de pruebas unitarias automatizadas

Antes de comenzar con la escritura y la ejecución de pruebas unitarias, se debe instalar una librería que ayude al programador. Para el caso de nuestros ejemplos, utilizaremos la librería Junit, la más conocida y aceptada para la tecnología Java.

Si se utiliza un gestor de proyecto, basta con añadir la dependencia de JUnit en el archivo de dependencias. Un ejemplo con el gestor Maven (más detalles en «Gestión de la construcción con Maven» del Anexo) se muestra en la Figura 43.



```
java x Main.java pom.xml
<project.build.sourceEncoding>UTF-8</project.build.source
<!--<configuration.path>src/main/config</configuration
<suite>smoke</suite>
<reportDirectory>target/surefire-reports</reportDirector
<sonar.coverage.jacoco.xmlReportPaths>${basedir}/target/
</properties>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13</version>
  </dependency>
  <dependency>
    <groupId>org.uncommons</groupId>
    <artifactId>reportng</artifactId>
    <version>1.1.4</version>
```

Figura 43. Automatización de pruebas para defectos encontrados.

Una vez instalada la librería de pruebas, ya podemos escribir las pruebas unitarias (ver sección Guía para la escritura de pruebas unitarias). La Figura 44 muestra un ejemplo de una clase *counter* con dos métodos *countNumbers* y *hasNumbers*. El primer método cuenta los números que hay en un parámetro de entrada de tipo *string* y el segundo método verifica si un *string* de entrada contiene números.

```

public class Counter {

    /* To count the numbers in the input */
    public static int countNumbers(String input) {
        int count = 0;
        for (char letter : input.toCharArray()) {
            if (Character.isDigit(letter))
                count++;
        }
        return count;
    }

    /* To check whether the input has number*/
    public static boolean hasNumber(String input) {
        return input.matches(regex: ".*\\d.*");
    }
}

```

Figura 44. Clase *counter* de ejemplo para verificar mediante pruebas unitarias.

Antes de crear las pruebas unitarias para la clase *counter*, es necesario comprender sentencias de JUnit. Se utiliza la anotación `@Test` para identificar un método como un caso de prueba unitaria, y métodos especiales denominados aserciones para realizar las verificaciones. Estas aserciones verifican que se cumplan condiciones esperadas como, por ejemplo, que un número sea mayor que otro.

Para las pruebas unitarias de la clase *counter*, como para cualquier otra clase a verificar, se crea una clase de prueba con el mismo nombre seguido de la palabra *test*: el nombre final para nuestro ejemplo será *CounterTest*. Luego, en ella, se tendrán tantos métodos de pruebas con la anotación `@Test` como escenarios se quieran verificar. Un ejemplo se muestra en la Figura 45.

```

import org.junit.Assert;
import org.junit.Test;

public class CounterTest {

    @Test
    public void countNumbersTest() {
        int expectedCount = 3;
        int actualCount = Counter.countNumbers( input: "Hi 123");
        Assert.assertEquals(expectedCount, actualCount);
    }

    @Test
    public void hasNumberTest() {
        boolean expectedValue = false;
        boolean actualValue = Counter.hasNumber( input: "Hi there!");
        Assert.assertEquals(expectedValue, actualValue);
    }
}

```

Figura 45. Clase de pruebas unitarias *CounterTest*.

10.1.2. Agrupamiento de pruebas unitarias

El agrupamiento de pruebas unitarias consiste en estructurarlas según el módulo y la funcionalidad que están verificando con el fin de ejecutarlas parcialmente o con diferentes prioridades. El objetivo principal es, entonces, tener diferentes estrategias de frecuencia por agrupamiento. Por ejemplo, si el código está orientado a objetos, cumpliendo con el patrón de *bajo acoplamiento* y *alta cohesión*, estará estructurado en clases y luego en paquetes de acuerdo con un determinado criterio. Para las pruebas, se debe seguir la misma estructura, de tal manera que cada clase y paquete del código base tendrá su correspondiente clase y paquete de prueba. Dependiendo de la tecnología de prueba unitaria, es posible categorizar las pruebas con palabras reservadas (ver «Estrategias de agrupamiento de pruebas unitarias» en el Anexo). Se muestra un ejemplo en la Figura 46.

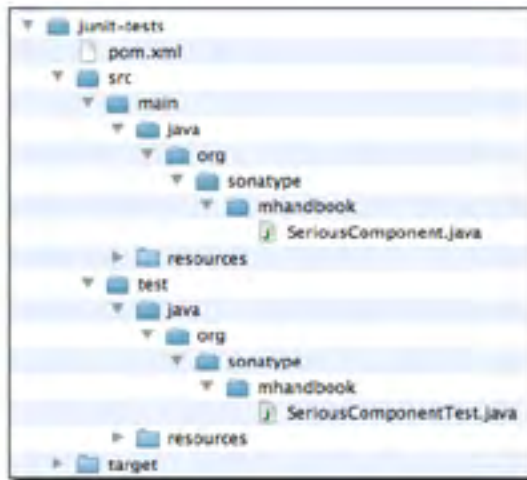


Figura 46. Pruebas unitarias agrupadas según el código base.

10.1.3. Cobertura de pruebas unitarias

La cobertura de pruebas unitarias es una medida utilizada para describir el grado en que el código fuente –es decir, la funcionalidad– es comprobado por las pruebas unitarias. Un programa con una alta cobertura de pruebas tiene una menor probabilidad de contener errores de software no detectados en comparación con un programa con baja cobertura de pruebas. Además, proporciona información crítica para mostrar a los equipos dónde enfocar las pruebas. Otra de las ventajas más importantes que presenta el uso de una herramienta para medir la cobertura de pruebas unitarias es detectar qué parte del código ha sido modificada y, por lo tanto, se puede reducir la cantidad de pruebas a ejecutar. En la «Cobertura de pruebas unitarias» del Anexo se enlaza más información respecto de las estrategias de cobertura.

Dependiendo del lenguaje de programación utilizado, tanto para la aplicación como para las pruebas unitarias existe una gran cantidad de herramientas que pueden ser utilizadas para medir la cobertura del código. La mayoría de estas herramientas se integran al proyecto y su ejecución depende de la ejecución de las pruebas unitarias.

Una vez que las pruebas unitarias hayan finalizado su ejecución, la herramienta analiza el código y genera el reporte de cobertura de todo el proyecto, como así también de cada paquete y clase. Algunas herramientas también muestran en detalle qué partes del código están cubiertas y qué partes no. Un ejemplo de reporte de cobertura se muestra en la Figura 47.



Figura 47. Ejemplo de reporte de cobertura utilizando la herramienta JaCoCo.

El nivel de cobertura se mide, por lo general, como el porcentaje de líneas de código fuente del sistema bajo prueba a partir del código de la prueba unitaria. Si se define como objetivo un porcentaje determinado de cobertura, los desarrolladores intentarán alcanzarlo. El problema es que los números de alta cobertura son fáciles de alcanzar con pruebas de baja calidad. Los valores mayores a un 80% de cobertura son considerados buenos, en tanto también la inspección de la calidad de las pruebas dé buenos resultados.

En este sentido, es necesario establecer el umbral de cobertura o el nivel de cobertura deseado, para evitar desarrolladores que no escriban pruebas unitarias por cada cambio que introducen al repositorio de control de versiones.

10.2. PRUEBAS FUNCIONALES

Pertenece a esta etapa cualquier tipo de prueba que se realiza para verificar el funcionamiento de la aplicación. Es importante tener un registro de los diferentes escenarios de pruebas que existen. Dependiendo del tiempo del que se disponga en el proyecto y la cantidad de recursos dedicados al proceso de pruebas, este registro puede ser simplemente un listado de todos los escenarios que se ejecutan para verificar las funcionalidades del sistema, como también un conjunto de casos de pruebas bien documentados.

En la actualidad, existen muchas herramientas de automatización de acuerdo con la necesidad y la tecnología de cada proyecto¹.

La librería de pruebas funcionales debe ser parte del proyecto donde se encuentra el código base de la aplicación. En lo que respecta al lenguaje de programación, algunos autores recomiendan utilizar el mismo lenguaje que se utiliza en el *backend* para las pruebas de API y el que se utiliza en *frontend* para las pruebas de GUI. Sin embargo, esto no siempre es posible en los equipos de especialistas en pruebas automatizadas que trabajan de forma separada los desarrolladores.

1. Ver lista detallada en Joe Colantonio (2022). 58 best automation testing tools: the ultimate list guide. EE.UU.: Test Guild. Disponible en <https://testguild.com/automation-testing-tools/>

Es importante señalar que, en los entornos continuos, la etapa de pruebas funcionales abarca únicamente las regresiones, donde solo se prueban funcionalidades previamente desarrolladas. La verificación de la funcionalidad que se está desarrollando, cuyos cambios son introducidos en el flujo de integración continua, se realiza mediante pruebas de aceptación.

No solo deben automatizarse las pruebas funcionales de GUI (sitios web, aplicaciones móviles o de escritorio), sino también las de *backend*, como ser llamadas a bases de datos, servicios web y otras APIs.

10.2.1. Librerías de automatización de pruebas funcionales

Una librería de automatización de pruebas funcionales es un conjunto de herramientas, funciones y prácticas que proporcionan el marco de trabajo para desarrollar pruebas de software funcionales automatizadas. Integran las bibliotecas de funciones, fuentes o llamadas a datos de pruebas, modelos de objetos y diversos módulos reutilizables. Los componentes actúan como pequeños bloques de construcción que deben ser ensamblados para representar un proceso de negocio.

En pruebas continuas, para construir un *framework*, se debe utilizar una arquitectura que lo divida en tres capas:

- La capa de más bajo nivel debe comprender cómo interactuar con la aplicación (sea *frontend* o *backend*) para realizar acciones y devolver resultados.
- En la capa intermedia, el código no interactúa con elementos de una GUI, llamadas de servicios o a otras partes del *backend*. En lugar de eso, invoca a las funciones brindadas por la capa de más bajo nivel.
- La capa de más alto nivel está expresada en un lenguaje no técnico para usuarios en general (*stakeholders*, dueños de producto, analistas, entre otros).

En la capa de más bajo nivel se implementa una herramienta de interacción con la aplicación. Por ejemplo, si la aplicación es web, la herramienta más utilizada es Selenium WebDriver, que brinda una serie de funciones para manipular un navegador web como si fuera un usuario real: navegar a un sitio, presionar un botón o completar un formulario.

En la Figura 48 se muestra un ejemplo de un caso de prueba funcional automatizado que utiliza Selenium. El objeto WebDriver es el que genera un nuevo navegador en el que se ejecutarán las pruebas, mientras que los objetos WebElement simulan los componentes de la GUI. Estos últimos soportan acciones para interactuar con los componentes reales del navegador, ver más detalles en «Ejemplo de un procedimiento para el uso de Selenium».

```

import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

public class SeleniumExample {

    private WebDriver driver;

    @BeforeTest
    public void setUp() {
        driver = new ChromeDriver();
    }

    @Test
    public void simpleTest() {
        driver.get("http://google.com");
        WebElement searchBox = driver.findElement(By.id("q"));
        searchBox.sendKeys("Continuous Testing");
        WebElement searchbutton = driver.findElement(By.id("submit"));
        searchbutton.click();
        String titlePage = driver.getTitle();
        Assert.assertTrue(titlePage.contains("Continuous Testing"),
            "message: " + "No se muestran resultados de Continuous Testing");
    }
}

```

Figura 48. Ejemplo de prueba funcional con herramienta Selenium WebDriver.

Capa de más bajo nivel

Como se mencionó antes, esta capa es la encargada de realizar las interacciones entre el navegador y sus componentes. Los tests no pertenecen a esta capa, sino que la invocan y utilizan los métodos que provee.

Un patrón de diseño utilizado para la implementación de esta instancia es el modelo objeto-página, más conocido en inglés como *PageObject* (más detalles en «Información del modelo página-objeto» del Anexo). Este patrón se ha popularizado en la automatización de pruebas para mejorar el mantenimiento de las pruebas y reducir la duplicación de código. Un *PageObject* es una clase orientada a objetos que sirve como interfaz para una página web o pantalla de escritorio de la aplicación que se está probando. Las pruebas utilizan los métodos de esta clase siempre que necesiten interactuar con la GUI, en lugar de contener el código que realiza dicha interacción. El beneficio es que, si la GUI cambia, el código de las pruebas no necesita ser cambiado, solo el código dentro del *PageObject*. Posteriormente, todos los cambios para admitir esa nueva GUI se encuentran en un solo lugar. En la Figura 49 se muestra una clase de ejemplo en un entorno de desarrollo.

```

[File] [View] [Window] [Help]  org.automation [D:\code\pages\webdriver\org.automation\org.automation\src\main\java\org\automation\.../AdvisorySearchPage.java
  go  publish  pages  AdvisorySearchPage
  SeleniumExample.java  CalendarPage.java  HomePage.java  ProductPage.java  AdvisorySearchPage.java  Maven
  33
  34  public class AdvisorySearchPage extends BasePage<WebDriver> {
  35
  36      private String cssSelectorFormContainer = "[@id='main-container']/div[1]/div[1]/div[1]";
  37      private By cssSelector = By.cssSelector("div.containerWrapper.containerWrapperTop");
  38      private By dateFrom = By.xpath("//*[@id='from']");
  39      private By dateTo = By.xpath("//*[@id='to']");
  40      private By dateFromTableResults = By.xpath("//*[@id='results']/div");
  41      private By updateResultsButton = By.xpath("//*[@id='updateResults']");
  42      private By searchInput = By.xpath("//*[@id='numberText']");
  43      private By categoryCheckboxes = By.xpath("//*[@id='categoryBlock']/div");
  44
  45      public AdvisorySearchPage(WebDriver driver, Logger log) { super(driver, log); }
  46
  47      public boolean isSearchTableFound() {
  48          WebElement processInputWidget = this.findElement(driver);
  49          return driver.findElement(By.cssSelector("#tableBody.containerTable")).isDisplayed();
  50      }
  51
  52      public String getFirstNotice() {
  53          WebElement processInputWidget = this.findElement(driver);
  54          String firstNoticeText = find(By.xpath("//*[@id='noticeText']")).getText(); // TODO
  55          return firstNoticeText;
  56      }
  57
  58      public void clickOnNotice(String firstNoticeText) {
  59          WebElement noticeLink = driver.findElement(By.linkText(firstNoticeText));
  60          WebElement webFormLink = driver.findElement(By.xpath("//a"));
  61      }
  62
  63      AdvisorySearchPage - gefindKorcel
  4:57 CRP UTP-4 4 spaces C:\code\automation\src\main
  
```

Figura 49. Ejemplo de implementación de una prueba funcional con el modelo objeto-página.

Capa intermedia

En la capa intermedia se escriben los casos de pruebas funcionales. Una clase de casos de prueba funcionales contiene método y un conjunto de instrucciones con anotaciones `@Test` al inicio de cada caso de prueba, la instanciación de las clases objeto-página a utilizar junto con la invocación de sus métodos y la verificación de un resultado esperado.

Para esta capa, se puede utilizar la misma librería de prueba instalada para pruebas unitarias o se puede utilizar otra más específica para pruebas funcionales, como es el caso de TestNG (ver «Pruebas funcionales con TestNG» del Anexo). Lo importante, más allá de la herramienta, es la estrategia de la prueba, que buscará desarrollar un conjunto de pasos que simulen el uso del sistema.

En la Figura 50 se muestra un ejemplo de caso de prueba funcional. Este invoca una clase del tipo *PageObject* llamada *HomePage* y sus métodos.

```

import org.testng.Assert;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

public class HomePageTest extends BaseTest {

    HomePage homePage;

    @BeforeTest
    public void setUp() {
        driver.get("http://continuous-testing.demopy.com");
        homePage = new HomePage();
    }

    @Test
    public void verifyFormSuccess() {
        homePage.openForm();
        homePage.fillForm( email: "uno@dos.tres", message: "Mensaje a enviar");
        String actualResult = homePage.getPopUpMessage();
        Assert.assertTrue(actualResult.contains("Mensaje Enviado"),
            message: "El mensaje no ha sido enviado");
    }
}

```

Figura 50. Ejemplo de caso de prueba funcional automatizado.

Capa de alto nivel

La capa de más alto nivel para las pruebas funcionales automatizadas es la capa de criterios de aceptación. Los analistas definen los criterios de aceptación para las historias de usuario, y estos deben cumplirse para que la historia de usuario sea reconocida como hecha. En 2006 se referencia por primera vez a un lenguaje específico de dominio para escribir criterios de aceptación, que toma la siguiente forma:

- *Dado* un contexto inicial,
- *Cuando* un evento se produce,
- *Entonces* se generan resultados.

En este sentido, existen diferentes herramientas, como Cucumber, JBehave, Conordion, Twist o FitNesse, con las que es posible incorporar criterios de aceptación directamente en las pruebas y vincularlos a la implementación subyacente (ver «Pruebas de aceptación» en el Anexo).

Sin embargo, también se puede adoptar el enfoque de codificar los criterios de aceptación en los nombres de las pruebas utilizando la librería de pruebas. Luego, se pueden ejecutar las pruebas directamente en lugar de incluir una capa de criterios de aceptación.

10.2.2. Agrupamiento de pruebas funcionales

La categorización de pruebas se define como el proceso de agrupar o segmentar las pruebas según características en común. Para las pruebas funcionales, se utilizan dos grupos: funcionalidades y prioridad. En primer lugar, si las pruebas están agrupadas por funcionalidad, se tiene la posibilidad de ejecutar solo determinadas pruebas para cambios pequeños, en lugar del lote completo. Por otro lado, si las pruebas se encuentran categorizadas según su prioridad (pruebas de humo, regresión, entre otras), se pueden ejecutar las más críticas al principio, permitiendo la detección de fallos críticos lo antes posible. Una ventaja adicional es que la paralelización es más fácil de implementar teniendo pruebas segmentadas.

Para realizar el agrupamiento primero se deben recorrer las pruebas funcionales automatizadas y marcarlas como pertenecientes a una funcionalidad y a un nivel de criticidad. Para ello, la mayoría de las librerías de pruebas tiene una característica de agrupamiento que utiliza anotaciones o etiquetas (ver la Figura 51).

```
@Test(groups = "smoke")
public void verifyFormSuccess() {
    homePage.openForm();
    homePage.fillForm( email: "uno@dos.tres", message: "Mensaje + enviar");
    String actualResult = homePage.getPopUpMessage();
    Assert.assertTrue(actualResult.contains("Mensaje Enviado"),
        message: "El mensaje no ha sido enviado");
}
```

Figura 51. Anotación de tipo «groups» de TestNG para agrupar casos de pruebas.

Finalmente, se deben crear archivos de configuración de lotes de pruebas o «suites» (ver la Figura 52). Un archivo de lotes de pruebas permite ejecutar casos de pruebas automatizados pertenecientes a uno o a varios grupos específicos. También permite excluir casos de pruebas de otros grupos. El objetivo es crear suites por cada funcionalidad y por cada nivel de criticidad.

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="smoke suite" verbose="1" parallel="classes" thread-count="4">
<test name="regression">
  <groups>
    <run>
      <exclude name="deprecated"/>
      <exclude name="disabled"/>
      <exclude name="not-beta"/>
      <exclude name="only-preview"/>
      <exclude name="only-prod"/>
      <exclude name="full"/>
      <exclude name="regression"/>
      <include name="smoke"/>
    </run>
  </groups>
  <packages>
    <package name="com.qa.tests.publish.*"/>
    <package name="com.qa.tests.integration.*"/>
  </packages>
</test>
</suite>
```

Figura 52. Archivo de lote de pruebas creado con TestNG.



Capítulo 11. Servidores para la construcción de pruebas continuas de software

En el capítulo anterior se dieron ejemplos de cómo automatizar algunos de los niveles de prueba necesarios para tener pruebas continuas de software. En este capítulo, de manera análoga, se explicarán los pasos para la construcción de un servidor de integración continua que incluya la automatización de las pruebas.

Como se describe en el capítulo 3 de este libro, en una estrategia de integración continua de desarrollo, a cada grupo de cambios introducidos en el código fuente, le sigue una fase de despliegue y prueba. El conjunto de las etapas que participan en este proceso se denomina conducto de despliegue.

Las tres tecnologías necesarias para construir un ecosistema de integración continua con conducto de despliegue son:

- Un servidor de integración continua. Algunos ejemplos son: Jenkins, Bamboo, Cruise-Control o TeamCity.
- Un gestor para el control de versiones del código fuente, como por ejemplo GIT, SVN, Mercurial, Perforce o ClearCase.
- Un script de construcción de código y ejecución de pruebas.

Además, en entornos de desarrollo continuo de software se requieren herramientas adicionales. Una de las técnicas más usadas para disminuir los tiempos de ejecución de pruebas es la paralelización (Mascheroni e Irrazábal, 2018b) y, para ello, es necesario construir una infraestructura de nodos conectados entre sí en los que serán distribuidas para su ejecución. La herramienta más utilizada es Selenium Grid. En la siguiente sección se describen los pasos para su instalación y configuración.

11.1. SELENIUM GRID

Selenium Grid permite ejecutar lotes de pruebas en paralelo, en una arquitectura con diferentes máquinas conectadas entre sí, a partir de una arquitectura cliente-servidor.

Normalmente, las pruebas de interfaz de usuario automatizadas son desarrolladas utilizando herramientas como Selenium WebDriver y son ejecutadas a través de librerías de pruebas, como Junit o TestNg. Una de las principales ventajas de Selenium Grid es su compatibilidad con la mayoría de las herramientas de pruebas automatizadas (ver más en «Instalación de una estrategia de pruebas en paralelo» en el Anexo).

Esta herramienta funciona como un servidor proxy inteligente, donde las instrucciones de las pruebas automatizadas son dirigidas a instancias de navegadores web. El nodo concentrador o servidor recibe el nombre de «hub» y los nodos cliente, donde se encuentran los navegadores web que ejecutarán las pruebas, son los nodos «grid».

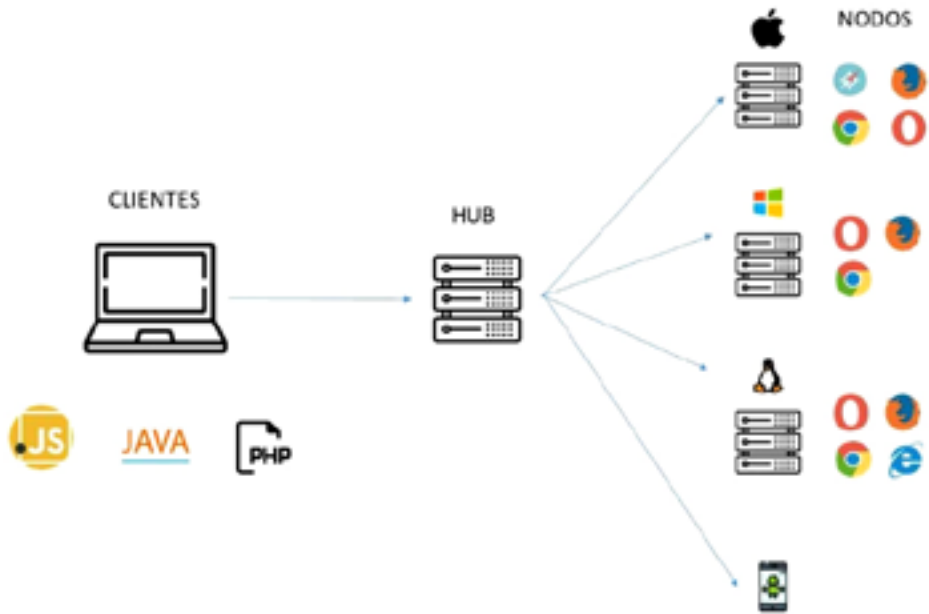


Figura 53. Infraestructura de pruebas en paralelo.

Antes de comenzar con la instalación de Selenium Grid, se deben determinar cuántas computadoras van a ser utilizadas para distribuir las pruebas. Pueden ser computadoras físicas o máquinas virtuales, el único requisito es que se encuentren conectadas a la misma red de área local.

El servidor de Selenium Grid es gratuito y fácil de obtener en línea. Una vez descargado el servidor, el primer paso consiste en determinar cuál de los nodos será el «hub» principal. El archivo se debe copiar al nodo y ejecutar un comando por consola para su ejecución.

Hecho esto, se comienza a ejecutar el servidor de Selenium y en la misma consola se mostrará la dirección URL donde los nodos cliente tendrán que registrarse.

Una vez enlazados los nodos cliente al servidor, la infraestructura está lista para ejecutar las pruebas en paralelo, distribuidas a través de los diferentes navegadores y sistemas operativos de los nodos.

11.2. CONDUCTO DE INTEGRACIÓN CONTINUA

Tal y como se describió en el capítulo 4, un conducto de despliegue continuo es un conjunto de etapas de construcción, pruebas de todo tipo y despliegue del software, desde el momento en el que los cambios son introducidos al repositorio de control de versiones hasta que se convierte en una versión lista para ser liberada a producción.

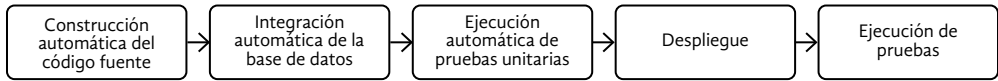


Figura 54. Flujo de pasos de la integración continua.

Dependiendo del tipo de enfoque utilizado (tradicional, despliegue continuo o entrega continua), el conducto tendrá más o menos etapas de pruebas y de despliegues. En el subtítulo 4.3 del libro se han detallado los pasos del proceso de ejecución del conducto de despliegue.

El primer paso práctico para la generación de un conducto de prueba continua es la utilización de un servidor de integración continua. A continuación, se describe la instalación y la configuración de un conducto de despliegue paso a paso.

11.2.1. Instalación del servidor de integración continua

Una de las herramientas de código abierto más utilizada para implementar un conducto de integración continua es Jenkins, que permite la automatización y la programación de tareas para diferentes propósitos como, por ejemplo, ejecución y reporte de pruebas.

Como primera recomendación, los diferentes autores proponen seleccionar un hardware o máquina virtual de manera exclusiva como servidor de integración continua para así aprovechar los recursos, especialmente necesarios en pruebas exhaustivas y complejas. Por este motivo, el sistema operativo más utilizado para este fin es Linux.

Desde la página principal de Jenkins, se pueden crear nuevas tareas, gestionar los usuarios y los complementos, agregar o eliminar vistas para organizar las tareas y ver el estado de las ejecuciones de las pruebas (ver «Servidor de integración continua» del Anexo).



Figura 55. Ventana principal típica para la configuración de un servidor de integración continua.

11.2.2. Configuración del conducto de integración continua

Para la construcción de los conductos de integración continua, es necesario el uso de complementos en Jenkins, que servirá para el diseño del flujo de trabajo de las tareas. El complemento más utilizado para ello es el Pipeline¹, donde se incluyen las secuencias de comandos.

A través de varios pasos, los conductos de integración permiten ejecutar tareas simples y complejas, de acuerdo con los parámetros que se establezcan. Una vez creados, pueden orquestar el trabajo necesario para llevar versiones de la aplicación desde la primera etapa de introducción de cambios hasta su entrega o despliegue a producción.

11.2.3. Creación del conducto de integración continua en Jenkins

Una vez instalado el complemento de Jenkins Pipeline, para crear un conducto de integración continua simple desde la interfaz de Jenkins, es posible un nuevo flujo de trabajo de este tipo, como puede verse en la Figura 56.

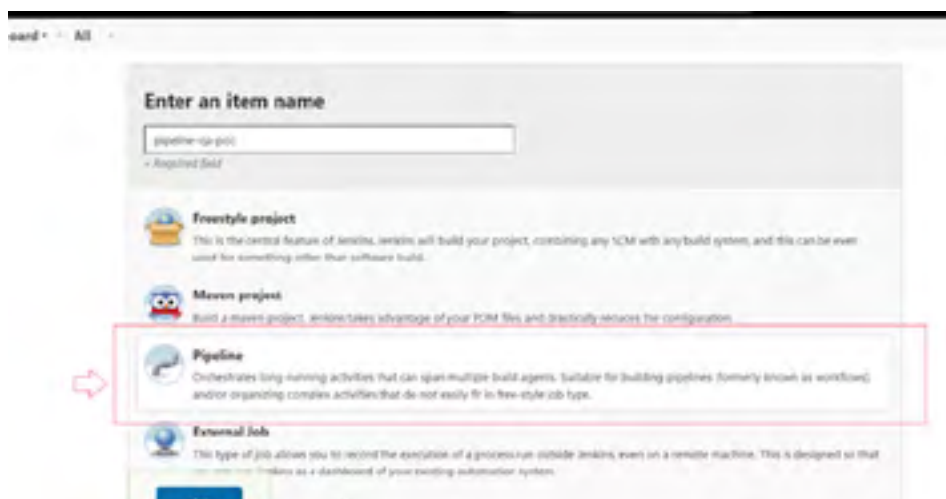


Figura 56. Nuevo flujo de trabajo con el complemento «Pipeline».

Se presenta debajo un ejemplo de un conducto de integración continua que utiliza tareas que ya existen en el servidor Jenkins:

- qa-ct-build compila el código fuente;
- qa-ct-functional-tests ejecuta pruebas funcionales utilizando Selenium WebDriver en el Grid;

1. Ver detalle en «Creating your first pipeline» en <https://www.jenkins.io/doc/pipeline/tour/hello-world/io>

- qa-ct-non-functional-tests ejecuta pruebas no funcionales de rendimiento y seguridad;
- qa-ct-unit-tests ejecuta pruebas unitarias.

Asimismo, se utilizará una tarea existente main-ct-deployment, que realiza el despliegue de la aplicación a un ambiente de prueba (ver Figura 57).

| | | | | | | |
|---|---|----------------------------|-------------------|-----|------|---|
| 🌐 | 🟡 | qa-ct-build | 1 min 12 sec - #1 | N/A | 9 ms | 🔄 |
| 🌐 | 🟡 | qa-ct-functional-tests | N/A | N/A | N/A | 🔄 |
| 🌐 | 🟡 | qa-ct-non-functional-tests | N/A | N/A | N/A | 🔄 |
| 🌐 | 🟡 | qa-ct-unit-tests | N/A | N/A | N/A | 🔄 |

Figura 57. Tareas de Jenkins que conformarán el flujo de trabajo de un «Pipeline».

Para configurar el conductor de integración continua, utilizando las tareas mencionadas, debemos dirigirnos al panel principal de Jenkins y crear un nuevo flujo de trabajo utilizando la opción «Pipeline». En la Figura 58 se muestra un ejemplo de cómo enlazar las diferentes tareas a partir de la secuencia de comandos.

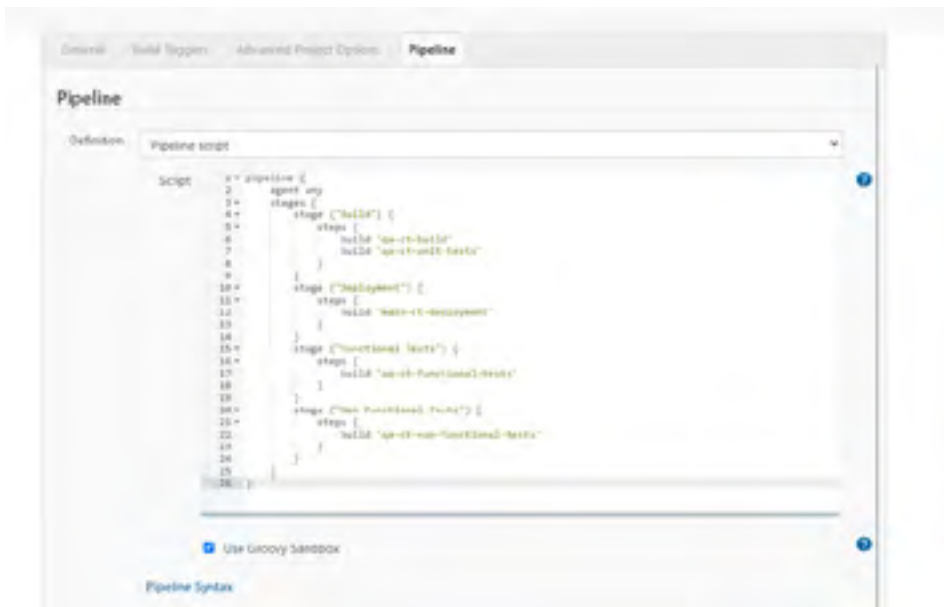


Figura 58. Flujo de trabajo de un conducto de despliegue.

Al ejecutar el conducto, la salida es la mostrada en la Figura 59. Esto indica que todas las tareas fueron ejecutadas correctamente como parte del conducto o del flujo de trabajo creado.

Pipeline qa-pipeline-poc



Figura 59. Reporte de un conducto de despliegue.



CONTROL DE VERSIONES CON GIT

Las aplicaciones para el control de versiones del software son fundamentales para la coordinación del trabajo en equipo. Su objetivo es llevar un registro de los cambios en el código fuente y asegurar que cada programador pueda trabajar de manera independiente, pero a la vez cooperativa.

Una de las guías consideradas oficiales de Git se encuentra en el siguiente enlace: Chacon, Scott y Straub, Ben (2014). *Pro Git*. Apress. Disponible en <https://bit.ly/3p871AZ>



GESTIÓN DE LA CONSTRUCCIÓN CON MAVEN

Maven es una herramienta de tecnología Java para la gestión y la construcción de proyectos software a partir de un archivo de configuración. Su objetivo es la gestión de los componentes internos y externos del proyecto junto con la planificación de los pasos de construcción, incluyendo tareas de inspección del código fuente y análisis del mismo.

En los siguientes enlaces encontrará un libro en línea gratuito sobre Maven: la referencia completa de la herramienta y una guía paso a paso con ejemplos.

O'Brien, Tim *et al.* (2010). *Maven: The complete reference*. EE.UU.: Sonatype. Disponible en <https://bit.ly/3CroBrw>

O'Brien, Tim *et al.* (2010). *Maven by example*. EE.UU.: Sonatype. Disponible en <https://bit.ly/3cfQQP6>

GUÍA PARA LA ESCRITURA DE PRUEBAS UNITARIAS

La escritura de pruebas unitarias requiere el conocimiento de las librerías utilizadas para ello, de sus principales funcionalidades y de las estrategias para lograr una correcta cobertura. En los siguientes enlaces se la describe por medio de ejemplos actuales:

Bechtold, Stefan *et al.* (Sin fecha). *Junit 5 User Guide*. Disponible en <https://bit.ly/2CEevRB>
De la Vega, Ramiro (sin fecha). «Pruebas unitarias en Java con Junit 5». En *Pharos.sh*. Disponible <https://bit.ly/3doBfNb>

ESTRATEGIAS DE AGRUPAMIENTO DE PRUEBAS UNITARIAS

La gestión de las pruebas unitarias, especialmente en cuanto a la cobertura y la ejecución eficiente de pruebas de regresión, implica el uso de estrategias de agrupamiento y priorización. En el siguiente enlace se describen ejemplos:

Kashyap, Vinod Kumar (sin fecha). *Junit group tests example*. Disponible en <https://bit.ly/3Sjbd7S>

COBERTURA DE PRUEBAS UNITARIAS

Una de las funcionales obtenidas del uso sistemático de librerías en pruebas unitarias es la medición automática del grado de cobertura. Para ello, es indispensable establecer el tipo y propósito de la cobertura, que podrá ser diferente por grupo de prueba o funcionalidad que se esté probando. En el primer enlace presentado, uno de los mayores especialistas en pruebas de software describe las buenas prácticas de cobertura. En el segundo enlace, se dan más ejemplos de cobertura:

Fowler, Martin (sin fecha). *Test coverage*. Disponible en <https://bit.ly/2s3SWqj>

Bodner, Jon (2019). *Improve Java Code Coverage and Quality with Unit Tests and JaCoCo*. Disponible en <https://bit.ly/3QEr8SK>

EJEMPLO DE UN PROCEDIMIENTO PARA EL USO DE SELENIUM

En el siguiente enlace se presenta un procedimiento para el uso de una de las herramientas más utilizadas para la automatización de pruebas funcionales.

Marco de desarrollo de la Junta de Andalucía (sin fecha). *Selenium y la automatización de las pruebas*. Disponible en <https://bit.ly/3zSjnxo>

INFORMACIÓN DEL MODELO PÁGINA-OBJETO

Las pruebas unitarias ayudan al desarrollo con patrones de diseño y a la mejora del código fuente. En el siguiente enlace se describe un patrón de diseño que ayuda a la construcción de pruebas unitarias.

Fowler, Martin (sin fecha). *Page object*. Disponible en <https://bit.ly/2uDq8WU>

PRUEBAS FUNCIONALES CON TESTNG

El paso de automatizar pruebas unitarias a pruebas de integración es simple, en tanto la tecnología utilizada es similar. Existen herramientas para el desarrollo de pruebas de integración que han evolucionado de las librerías de pruebas unitarias. Un ejemplo es TestNG y a continuación el enlace a un tutorial detallado:

Rajkumar (2020). *TestNG Tutorial – Complete Guide For Testers | Software Testing Material*. Disponible en <https://bit.ly/3QAGV5k>

PRUEBAS DE ACEPTACIÓN

Uno de los últimos pasos para probar la funcionalidad es hacerlo teniendo en cuenta qué tanto satisface los requerimientos del usuario. Esto ha llevado al uso de herramientas para la escritura de requerimientos que puedan lanzar pruebas de forma automática para comprobar la cobertura de las necesidades a partir de las funcionalidades. En los siguientes enlaces se describe esta estrategia.

Soto, Marcela (2019). *Automatización de Pruebas Funcionales: Serenity BDD+Screen Play+Java*. Disponible en <https://bit.ly/3pcyKQQ>

Blé, Carlos (2020). *Diseño ágil con TDD*. Disponible en <https://bit.ly/3SBrb3E>

INSTALACIÓN DE UNA ESTRATEGIA DE PRUEBAS EN PARALELO



La exhaustividad y velocidad de las pruebas en entornos continuos requieren de una estrategia de paralelización que, en otros entornos, no es tenida en cuenta. El siguiente enlace describe una herramienta de paralelización que forma parte de una familia de herramientas para el desarrollo de pruebas funcionales con altas prestaciones.

Selenium (2022). *The Selenium Browser Automation Project*. Disponible en <https://bit.ly/3pEViZ>

SERVIDOR DE INTEGRACIÓN CONTINUA

La instalación, la personalización, el diseño de tareas y los reportes para materializar un servidor de integración continua requiere de la lectura y el uso de la documentación específica de cada herramienta. Ha quedado atrás el tiempo donde los manuales eran inexistentes o con muy poca información y, como parte de las funcionalidades presentadas por estas aplicaciones, se encuentra su documentación. En el siguiente enlace, ejemplificados, todos los pasos para el uso correcto del servidor.

Jenkins (sin fecha). *User Handbook overview*. Disponible en <https://bit.ly/3vX2eWE>

Referencias bibliográficas

- AGARWAL, Puneet (2011). «Continuous SCRUM: agile management of SAAS products» [Actas]. *Proceedings of the 4th India Software Engineering Conference* (pp. 51-60). India: TCS Innovation Labs.
- ALÉGROTH, Emil, Feldt, Robert y Ryrholm, Lisa (2015). «Visual gui testing in practice: challenges, problems and limitations». *Empirical Software Engineering*, 20 (3), 694-744.
- ALSHRAIDEH, Mohammad (2008). «A complete automation of unit testing for javascript programs». *Journal of Computer Science*, 4 (12), 1012.
- BECK, Kent *et al.* (2001). *Manifesto for agile software development*. Disponible en <https://agilemanifesto.org/>.
- BEIZER, Boris y Vinter, Otto (1990). «Bug taxonomy and statistics». *Software Engineering Mentor*, 2630.
- BOOCH, Grady (1995). *Object solutions: managing the object-oriented project*. Addison Wesley Longman Publishing Co Inc.
- BÖRJESSON, Emil y Feldt, Robert (2012). «Automated system testing using visual gui testing tools: A comparative study in industry» [Actas]. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (pp. 350-359). IEEE.
- BOSCH, Jan (2000). *Design and use of software architectures: adopting and evolving a product-line approach*. EE.UU.: Pearson Education.
- BOURQUE, Pierre y Fairley, Richard E. (2014). *SWEBOK v3. 0: Guide to the software engineering body of knowledge* (pp. 1-335). IEEE Computer Society.
- BROOKS, Graham (2008). «Team pace keeping build times down» [Actas]. *Agile 2008 Conference* (pp. 294-297). IEEE.
- BURGIN, Mark y Debnath, Narayan (2010). «Intelligent testing systems» [Actas]. *2010 World Automation Congress* (pp. 1-6). IEEE.
- BUSJAEGER, Benjamin y Xie, Tao (2016). «Learning for test prioritization: an industrial case study» [Actas]. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 975-980). San Francisco.
- CAMPOS, José *et al.* (2015). «Continuous test generation on guava» [Actas]. *International Symposium on Search Based Software Engineering* (pp. 228-234). Suiza: Springer, Cham.
- CANNIZZO, Fabrizio, Clutton, Robbie y Ramesh, Raghav (2008). «Pushing the boundaries of testing and continuous integration» [Actas]. *Agile 2008 Conference* (pp. 501-505). IEEE.



- CHEN, Lianping (2017). «Continuous delivery: overcoming adoption challenges». *Journal of Systems and Software*, 128, 72-86.
- CHOW, Tsun y Cao, Dac-Buu (2008). «A survey study of critical success factors in agile software projects». *Journal of systems and software*, 81(6), 961-971.
- CLAPS, Gerry Gerard, Svensson, Richard Berntsson y Aurum, Aybüke (2015). «On the journey to continuous deployment: Technical and social challenges along the way». *Information and Software technology*, 57, 21-31.
- CMMI Production Team SEI (2010). *CMMI for Development, Version 1.3*. EE.UU.: Carnegie Mellon.
- COHN, Mike (2009). *Succeeding with Agile. Software Development using Scrum*. Reading, Massachusetts: Addison-Wesley.
- COUPAYE, Thierry y Estublier, Jacky (2000). «Foundations of enterprise software deployment» [Actas]. *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering* (pp. 65-73). IEEE.
- CRISPIN, Lisa y Gregory, Janet (2009). *Agile testing: A practical guide for testers and agile teams*. EE.UU.: Pearson Education.
- CUSUMANO, Michael A. y Selby, Richard W. (1998). *Microsoft secrets: how the world's most powerful software company creates technology, shapes markets, and manages people*. Nueva York: Simon and Schuster.
- DEBBICHE, Adam y Dienér, Mikael (2015). *Assessing challenges of continuous integration in the context of software requirements breakdown: a case study*. Tesis de Maestría.
- DUVALL, Paul M., Matyas, Steve y Glover, Andrew (2007). *Continuous integration: improving software quality and reducing risk*. EE.UU.: Pearson Education.
- ELBAUM, Sebastian, Rothermel, Gregg y Penix, John (2014). «Techniques for improving regression testing in continuous integration development environments» [Actas]. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 235-245).
- ELOUSSI, Lamyaa (2015). *Determining flaky tests from test failures*. Urbana, Illinois, EE.UU.: Universidad de Illinois.
- ENGBLOM, Jakob (2015). «Virtual to the (near) end: Using virtual platforms for continuous integration» [Actas]. *Proceedings of the 52nd Annual Design Automation Conference* (pp. 1-6).
- ERDER, Murat y Pureur, Pierre (2015). *Continuous architecture: sustainable architecture in an agile and cloud-centric world*. Massachusetts: Morgan Kaufmann.
- EYL, Martin, Reichmann, Clements y Müller-Glaser, Klaus (2016). «Fast feedback from automated tests executed with the product build» [Actas]. *International Conference on Software Quality* (pp. 199-210). Suiza: Springer, Cham.
- FEITELSON, Dror G., Frachtenberg, Eitan y Beck, Kent L. (2013). «Development and deployment at facebook». *IEEE Internet Computing*, 17(4), 8-17.
- FOWLER, Martin y Foemmel, Matthew (2006). «Continuous integration». *Thought-Works*, 122(14), 1-7. Disponible en <http://www.thoughtworks.com/ContinuousIntegration>
- FOWLER, Martin (2014). «Software Development in the 21st century». *Thought Works XCONF 2014*. Hamburg & Manchester.
- GARCÍA, Cecilia, Dávila, Abraham y Pessoa, Marcelo (2014). «Test process models: Systematic literature review» [Actas]. *International Conference on Software Process Improvement and Capability Determination* (pp. 84-93). Suiza: Springer, Cham.
- GARG, Naveen, Singla, Sanjay y Jangra, Suren (2016). «Challenges and techniques for testing of big data». *Procedia Computer Science*, 85, 940-948.
- GARVIN, David (1987). «Competing on the eight dimensions of quality». *Harvard Business Review*, 101-109.

- GMEINER, Johannes, Ramler, Rudolf y Haslinger, Julian (2015). «Automated testing in the continuous delivery pipeline: A case study of an online company» [Actas]. 2015 *IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 1-6). IEEE.
- GOMEDE, Everton, Da Silva, Rafael Thiago y De Barros, Rodolfo Miranda (2015). «A Practical Approach to Software Continuous Delivery Focused on Application Lifecycle Management». *Seke*, 320-325.
- GOTIMER, Gene y Stiehm, Thomas (2016). «Devops advantages for testing: Increasing quality through continuous delivery». *Cross Talk Magazine*, 13-18.
- GREGORY, Janet y Crispin, Lisa (2014). *More agile testing: learning journeys for the whole team*. Reading, Massachusetts: Addison-Wesley Professional.
- HASKINS, Bill, Stecklein, Jonette, Dick, Brandon, Moroney, Gregory, Lovell, Randy, Dabney, James y Mitasiunas, Antanas *et al.* (2004). «Software Process Improvement and Capability Determination». *Communications in Computer and Information Science*, 477, 1723-1737.
- HOFFMAN, Daniel y Weiss, David (2001). «Some software engineering principles». En Parnas, D. (ed.) *Software fundamentals: collected papers by David L. Parnas* (pp. 257-266). Reading, Massachusetts: Addison-Wesley Professional.
- HUIZINGA, Dorota y Kolawa, Adam (2007). *Automated defect prevention: best practices in software management*. Hoboken, Nueva Jersey: John Wiley & Sons.
- HUMBLE, Jez y Farley, David (2010). *Continuous delivery: reliable software releases through build, test, and deployment automation*. EE.UU.: Pearson Education.
- HUMPHREY, Watts S., Kitson, David H. y Kasse, Tim C. (1989). «The state of software engineering practice» [Actas]. *Proceedings of the 11th international conference on Software engineering* (pp. 277-285).
- IRRAZÁBAL, Emanuel y Garzás, Javier (2010). «Análisis de métricas básicas y herramientas de código libre para medir la mantenibilidad». *Reicis. Revista española de innovación, calidad e ingeniería del software*, 6 (3), 56-65.
- JAIN, Sarika y Joshi, Harshit (2016). «Impact of early testing on cost, reliability and release time» [Actas]. 2016 *5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)* (pp. 318-322). IEEE.
- JÖNGREN, Christian (2008). *Automated integration testing: An evaluation of cruisecontrol.net*. Estocolmo: Skolan för datavetenskap och kommunikation, Kungliga Tekniska högskolan.
- KITCHENHAM, Barbara y Pfleeger, Shari Lawrence (1996). «Software quality: the elusive target [special issues section]». *IEEE software*, 13 (1), 12-21.
- KITCHENHAM, Barbara (2004). «Procedures for performing systematic reviews». *Keele, UK, Keele University*, 33, 1-26.
- KNAUSS, Eric *et al.* (2015). «Supporting continuous integration by code-churn based test selection» [Actas]. 2015 *IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering* (pp. 19-25). IEEE.
- KUHN, D. Richard, Wallace, Dolores R. y Gallo, Albert M. (2004). «Software fault interactions and implications for software testing». *IEEE transactions on software engineering*, 30 (6), 418-421.
- LACOSTE, Francis J. (2009). «Killing the gatekeeper: Introducing a continuous integration system» [Actas]. 2009 *agile conference* (pp. 387-392). IEEE.
- LAUKKANEN, Eero, Itkonen, Juha, Lassenius, Casper (2017). «Problems, causes and solutions when adopting continuous delivery. A systematic literature review». *Information and Software Technology*, 82, 55-79.

- LEHMAN, Manny M. (1996). «Laws of software evolution revisited» [Actas]. *European Workshop on Software Process Technology* (pp. 108-124). Springer, Berlin: Heidelberg.
- LEPPÄNEN, Marko *et al.* (2015). «The highways and country roads to continuous deployment». *IEEE software*, 32 (2), 64-72.
- LIONS, Jacques-Louis *et al.* (1996). *Ariane 5 flight 501 failure report by the inquiry board*. París.
- LUO, Qingzhou *et al.* (2014). «An empirical analysis of flaky tests» [Actas]. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 643-653).
- LOURIDAS, Panagiotis (2006). «Static code analysis». *IEEE Software*, 23 (4), 58-61.
- MADEYSKI, Lech y Kawalerowicz, Marcin (2013). «Continuous Test-Driven Development-A Novel Agile Software Development Practice and Supporting Tool». *Enase*, 260-267.
- MAILA-MAILA, Fernando, Intriago-Pazmiño, Monserrate y Ibarra-Fiallo, Julio (2019). «Evaluation of open source software for testing performance of web applications» [Actas]. *World Conference on Information Systems and Technologies* (pp. 75-82). Suiza: Springer, Cham.
- MARIJAN, Dusica y Liaaen, Marius (2017). «Test prioritization with optimally balanced configuration coverage» [Actas]. *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)* (pp. 100-103). IEEE.
- MARTIN, Robert C. (2002). *Agile software development: principles, patterns, and practices*. Hoboken, Nueva Jersey: Prentice Hall.
- MASCHERONI, Maximiliano Agustín, Cogliolo, Mariela Katherina e Irrazábal, Emanuel (2017). «Automatic detection of Web Incompatibilities using Digital Image Processing». *Electronic Journal of SADIO (EJS)*, 16, 29-45.
- MASCHERONI, Maximiliano Agustín e Irrazábal, Emanuel (2018a). «Problemas que afectan a la calidad de software en entrega continua y pruebas continuas» [Actas]. *XXIV Congreso Argentino de Ciencias de la Computación*. La Plata.
- MASCHERONI, Maximiliano A. e Irrazábal, Emanuel (2018b). «Continuous testing and solutions for testing problems in continuous delivery: A systematic literature review». *Computación y Sistemas*, 22(3), 1009-1038.
- MASCHERONI, Maximiliano Agustín *et al.* (2019, octubre 14-18). «Rapid Releases and Testing Problems at the industry: A survey» [Actas]. *XXV Congreso Argentino de Ciencias de la Computación (Cacic)*. Río Cuarto, Córdoba: Universidad Nacional de Río Cuarto.
- MCCONNELL, Steve (1997). «Software's ten essentials». *IEEE Software*, 14(2), 144.
- MOE, Nils Brede *et al.* (2015). «Continuous software testing in a globally distributed project» [Actas]. *2015 IEEE 10th international conference on global software engineering* (pp. 130-134). IEEE.
- MUCCINI, Henry, Di Francesco, Antonio y Esposito, Patrizio (2012). «Software testing of mobile applications: Challenges and future research directions» [Actas]. *2012 7th International Workshop on Automation of Software Test (AST)* (pp. 29-35). IEEE.
- MUŞLU, Kivanç, Brun, Yuriy y Meliou, Alexandra (2013). «Data debugging with continuous testing» [Actas]. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (pp. 631-634).
- NEELY, Steve y Stolt, Steve (2013). «Continuous delivery? easy! just change everything (well, maybe it is not that easy)» [Actas]. *2013 Agile Conference* (pp. 121-128). IEEE.
- NYGARD, Michael T. (2018). *Release it!: design and deploy production-ready software*. Pragmatic Bookshelf.

- PENIX, John (2012). «Large-scale test automation in the cloud» [Actas]. *Proceedings of the 34th IEEE International Conference on Software Engineering (ICSE)* (pp. 1122).
- PFLEEGER, Shari Lawrence y Atlee, Joanne M. (1998). *Software engineering: theory and practice*. India: Pearson Education.
- PRADHAN, Ligaj (2011). *User Interface Test Automation and its Challenges in an Industrial Scenario*. Tesis doctoral. Suecia: Mälardalen University.
- PHILLIPS, A. et al. (2015). *The IT Manager's Guide to Continuous Delivery: Delivering business value in hours, not months*. Massachusetts: Xebia Labs.
- POPPENDIECK, Mary y Poppendieck, Tom (2003). *Lean software development: an agile toolkit*. Reading, Massachusetts: Addison-Wesley.
- RATHOD, Nikhil y Surve, Anil (2015). «Test orchestration a framework for continuous integration and continuous deployment» [Actas]. *2015 international conference on pervasive computing (ICPC)* (pp. 1-5). IEEE.
- RIES, Eric (2011). *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. Currency.
- RODRÍGUEZ, Moisés, Piattini, Mario y Ebert, Christof (2019). *Software verification and validation technologies and tools*. *IEEE Software*, 36(2), 13-24.
- ROBSON, Colin y McCartan, Kieran (2016). *Real world research*. Hoboken, Nueva Jersey: John Wiley & Sons.
- ROSSI, Chuck et al. (2016). «Continuous deployment of mobile software at facebook (showcase)» [Actas]. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 12-23).
- SABAREN, Leandro N. et al. (2018). «A systematic literature review in cross-browser testing». *Journal of Computer Science & Technology*, 18.
- SABEV, Peter y Grigorova, Katalina (2017). «A Comparative Study of GUI Automated Tools for Software Testing» [Actas]. *SOFT-ENG 2017, The Third International Conference on Advances and Trends in Software Engineering*. Vol. 3. Venecia, Italia.
- SADALAGE, Pramodkumar J. (2007). *Recipes for Continuous Database Integration*. Reading, Massachusetts: Addison-Wesley.
- SAFF, David y Ernst, Michael D. (2003). «Reducing wasted development time via continuous testing» [Actas]. *14th International Symposium on Software Reliability Engineering, ISSRE 2003* (pp. 281-292). IEEE.
- SAFF, David y Ernst, Michael D. (2004). «Continuous testing in Eclipse». *Electronic Notes in Theoretical Computer Science*, 107, 103-117.
- SHAHIN, Mojtaba, Babar, Muhammad Ali y Zhu, Liming (2017). «Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices». *IEEE Access*, 5, 3909-3943.
- SINISALO, Markus Tapani (2016). *Improving web user interface test automation in continuous integration*. Tesis de maestría. Finlandia: Universidad de Tampere.
- SMITH, Edward (2000). «Continuous testing» [Actas]. *17th International Conference on Testing Computer Software*.
- STÅHL, Daniel y Bosch, Jan (2014). «Automated software integration flows in industry: A multiple-case study» [Actas]. *Companion Proceedings of the 36th International Conference on Software Engineering* (pp. 54-63).
- SUWALA, Pawel (2015). *Challenges with modern web testing*. Estocolmo: Universidad de Linköping.

- TASSEY, Gregory (2003). «One Point of View: R&D Investment Trends: US Needs More High-Tech». *Research-Technology Management*, 46(2), 9-11.
- The Standish Group (2013). *The Curious Case of the CHAOS Report 2009*. Disponible en <https://bit.ly/3JPwmc8>
- TILLEY, Scott y Floss, Brianna (2014). «Hard Problems in Software Testing: Solutions Using Testing as a Service (TaaS)». *Synthesis Lectures on Software Engineering*, 2(1), 1-103.
- TSAI, Wei-Tek et al. (2011). «An approach for service composition and testing for cloud computing». *2011 Tenth International Symposium on Autonomous Decentralized Systems* (pp. 631-636). IEEE.
- VAN VEENENDAAL, Erik y Graham, Dorothy (2008). «Foundations of Software Testing: ISTQB Certification». *Cengage Learning Emea*, 30.
- VAFIAE, Nahid y Arvidsson, Mikael (2015). *Metrics to measure the impact of continuous integration—An empirical study*. Suecia: Universidad de Gotemburgo.
- VAN DER STORM, Tijds (2008). «Backtracking incremental continuous integration» [Actas]. *2008 12th European Conference on Software Maintenance and Reengineering* (pp. 233-242). IEEE.
- VIRMANI, Manish (2015). «Understanding DevOps & bridging the gap from continuous integration to continuous delivery» [Actas]. *Fifth international conference on the innovative computing technology (itech 2015)* (pp. 78-82). IEEE.
- WHITTAKER, James A. (2009). *Exploratory software testing: tips, tricks, tours, and techniques to guide test design*. EE.UU.: Pearson Education.
- WICHMANN, Brian A. et al. (1995). «Industrial perspective on static analysis». *Software Engineering Journal*, 10(2), 69-75.
- WIEGAND, J. et al. (2004). «Eclipse: A platform for integrating development tools». *IBM Systems Journal*, 43(2), 371-383.

Programas

- Ant (2021). *Ant (1.10.12)* [Script de construcción]. EE.UU.: The Apache Software Foundation. Disponible en <https://bit.ly/3dELcpY>
- Bamboo (2022). *Bamboo (8.2.5)* [Servidor de integración continua]. EE.UU.: Atlassian. Disponible en <https://bit.ly/3wpD3fM>
- CruiseControl (2022). *CruiseControl (2.8.4)* [Servidor de integración continua]. EE.UU.: ThoughtWorks. Disponible en <https://bit.ly/3CvIaPv>
- EvoSuite (2022). *EvoSuite (1.2.0)* [Automatic Test Suite Generation for Java]. EE.UU.: DGordon Fraser, Andrea Arcuri. Disponible en <https://bit.ly/3c9XiHy>
- Git (2007). *Git (2.37.2)* [Repositorio de control de versiones]. EE.UU.: Linus Torvalds. Disponible en <https://bit.ly/3dMZgOk>
- Jenkins (2022). *Jenkins (2.0)* [Servidor de integración continua]. EE.UU.: Oracle. Disponible en <https://bit.ly/3QIFNwY>
- Load Compact (0.39.0). *Load Compact (k6.10)* [Prueba de capacidad]. EE.UU.: Grafana Labs. Disponible en <https://bit.ly/3A1hsR>
- Maven (2022). *Maven (3.8.6)* [Script de construcción]. EE.UU.: The Apache Software Foundation. Disponible en <https://bit.ly/3AHELLY>
- MSBuild (2016). *MSBuild (12.0)* [Script de construcción]. EE.UU.: Microsoft. Disponible en <https://bit.ly/3TIQS8W>
- Nant (2012). *Nant (0.92)* [Script de construcción]. EE.UU.: Gerry Shaw. Disponible en <https://bit.ly/3PObqDU>
- Rake (2022). *Rake (13.0.6)* [Script de construcción]. EE.UU.: Jim Weirich. Disponible en <https://bit.ly/3KcJEQq>

Subversion (2000). *Subversion (1.14.2-1.10.8)*
[Repositorio de control de versiones]. EE.UU.:
The Apache Software Foundation. Disponible
en <https://bit.ly/3Agliv>

Travis CI (2022). Travis CI (13.4.1) [Servidor de
integración continua]. Alemania: Travis CI,
GmbH. Disponible en <https://bit.ly/3cfpV61>



Autores

Emanuel Irrazábal. Doctor en Informática (Universidad Rey Juan Carlos), magíster e ingeniero en Sistemas de Información. Es director del Grupo de Investigación en Calidad de Software de la Universidad Nacional del Nordeste, donde imparte clases en la asignatura Tópicos Avanzados de Ingeniería de Software de la Licenciatura en Sistemas de Información. Actualmente, dirige la Especialización en Tecnologías de la Información en la Universidad Nacional del Nordeste e imparte los cursos de posgrado de Pruebas de Software.

Agustín Mascheroni. Doctor en Ciencias Informática (Universidad Nacional de La Plata) y Licenciado en Sistemas de Información. Es miembro del Grupo de Investigación en Calidad de Software de la Universidad Nacional del Nordeste, donde realizó su tesis doctoral en el área de Pruebas Continuas de software. Cuenta con una trayectoria de 10 años trabajando en el área de calidad de software de diferentes empresas y fábricas de software. Actualmente, trabaja en la empresa Major Key Technologies, automatizando el aseguramiento de la calidad de los productos software.



**Rector**

Gerardo Omar Larroza

Vicerrector

José Leandro Basterra

**Secretaria General
de Ciencia y Técnica**

Laura Leiva

WWW.UNNE.EDU.AR

**Gerente**

Carlos Manuel Quiñonez

WWW.EUDENE.UNNE.EDU.AR

**Fundamentos de las pruebas continuas
de software** se compuso y diagramó
en Eudene Corrientes, Argentina,
en el mes de diciembre de 2022.



Relays 602 in 033 fault
 in Relay
 Relays changed
 Start: Cosine Taps (Sine check)
 Stop: Mod. Multi + Adder Test.
 Relay #70 Panel F
 (moth) in relay.

El 9 de septiembre de 1947 en la Universidad de Harvard tuvo lugar el primer defecto informático cuando una polilla ingresó entre las dos lengüetas de un relé. Este incidente quedó registrado (la responsable también). Desde ese momento el término *bug* es utilizado para referirse a los errores en el código fuente de un software, y se impusieron las rutinas de inspección y corrección: las aplicaciones software necesitaban ser probadas.