MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN INGENIERÍA DEL SOFTWARE Y SISTEMAS INFORMÁTICOS

E.T.S. DE INGENIERÍA INFORMÁTICA

TRABAJO FINAL DE MÁSTER (Código: 31105151)

AUTOMATIZACIÓN DE PROCESOS DE VERIFICACIÓN, VALIDACIÓN E INTEGRACIÓN SOFTWARE

AUTOR: David Ruiz Ansola

TUTOR: Ismael Abad Cardiel

CONVOCATORIA: junio 2019

TRABAJO FINAL DE MASTER

ITINERARIO:	Ingeniería del Software
CODIGO ASIGNATURA:	31105151
TIPOLOGIA:	Tipo B
TITULO:	AUTOMATIZACIÓN DE PROCESOS DE VERIFICACIÓN, VALIDACIÓN E INTEGRACIÓN DEL SOFTWARE
NOMBRE:	D. David Ruiz Ansola
DIRECTOR:	Ismael Abad Cardiel
CURSO ACADÉMICO:	2018/2019
CONVOCATORIA:	Junio

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 23/01/19

Quién suscribe:

Autor: DAVID RUIZ ANSOLA
DNI: 44788345T

Hace constar que es el autor del trabajo:

TRABAJO FINAL DE MASTER:

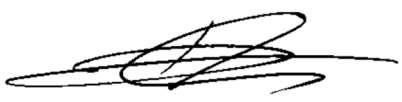
AUTOMATIZACIÓN DE PROCESOS DE VERIFICACIÓN, VALIDACIÓN E INTEGRACIÓN DEL SOFTWARE

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo.



Autorización de publicación y difusión del TFM para fines académicos

Autorización

Autorizo a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del Autor

RESUMEN

El proceso de verificación, validación e integración del software es esencial para el aseguramiento de la calidad en la ingeniería del software. Si bien, existen antecedentes con metodologías estructuradas por etapas que relacionan este proceso como una etapa final al desarrollo, cada vez se está implantando más una tendencia de evaluación temprana de la calidad del software, como un proceso continuo al desarrollo, haciendo que este sea más flexible y adaptándose a los cambios del proyecto, a la vez que asegura la implicación del usuario final en el desarrollo.

La práctica de la integración continua como metodología de desarrollo, tiene como objetivo la realización de entregables de manera continua, lo más frecuente posible, para obtener una evaluación temprana de la calidad. Esto da lugar a la reducción del riesgo del proyecto, al detectar fallos y errores a medida que se va desarrollando, de manera que el usuario final va percibiendo su evolución desde el inicio.

Estableciendo como base los principios propuestos en el manifiesto ágil y sus antecedentes, asi como el proceso unificado y la programación extrema, que dieron lugar a la definición de prácticas y metodologías conocidas como integración continua, este trabajo pretende realizar una evaluación de un modelo basado en dichas metodologías. Para ello, será necesario la definición de un prototipo de proyecto software, que sirva como ejemplo para aplicar un sistema de integración continua basado en dicho modelo y sobre este, proponer y aplicar una serie de mejoras en cuanto a calidad y eficacia, definidas en el alcance de este trabajo.

Por tanto, el objetivo final será el desarrollo y la implantación de técnicas de automatización que supongan una mejora sustancial al planteamiento inicial del proceso de verificación, validación e integración software mediante prácticas y metodologías basadas en la automatización de procesos para la integración continua.

PALABRAS CLAVE

Integración Continua (CI). Entrega Continua (CD). Despliegue Continuo (CD). Programación Extrema (XP). Proceso Unificado (UP). Desarrollo dirigido por pruebas (TDD). Aseguramiento de la calidad (QA). Última versión estable (LSV). Test Unitario (UT). Test Funcional (FT). Regresión de *testing*. *Scope* de *testing*. Proceso de *Nursing*. Proceso de *Release*. *Commit*. *Revert*. *Jobs. Builds. Pipeliene*.

ÍNDICE

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

Este trabajo pretende abarcar los contenidos y procesos relacionados con el desarrollo software, y en particular, aquellos procedimientos, metodologías y prácticas que supongan una mejora sustancial en cuanto al aseguramiento de la calidad. Es necesario por tanto comprender los principales modelos de desarrollo software que se han implementado desde los inicios de la programación, así como los principales procedimientos y estándares de gestión de mejora de procesos, desarrollados principalmente para los proyectos software.

Como punto de partida para comprender mejor el objetivo principal de este trabajo y el contexto en el que se desarrolla, es necesario una introducción de los modelos de desarrollo software precedentes al modelo en el que se quiere orientar el trabajo; el modelo iterativo e incremental, concretamente en las metodologías y prácticas de la integración continua.

Estableciendo como base la programación ágil, y como marco teórico la integración continua, la motivación de este trabajo se fundamenta en abarcar aquellos aspectos teóricos de estas metodologías que supongan una mejora en el proceso de desarrollo, y especialmente en la calidad del software, para proponer una serie de mejoras mediante la implantación de técnicas de automatización de procesos de verificación, validación e integración del software.

Se define el alcance de este trabajo con la implantación de aquellas mejoras propuestas al modelo planteado basado en la integración continua, que se fundamentan en los procesos automáticos de integración del software, mediante previa verificación y validación en función de unos criterios establecidos. Las mejoras propuestas por este trabajo tienen un alcance definido, dejando abierta una línea de investigación para posibles trabajos futuros.

En esta sección introductoria se mencionará, por tanto, el contexto en el que se desarrolla este trabajo, su motivación y objetivos, y el alcance del mismo.

1.1 CONTEXTO DEL TRABAJO

Desde los inicios de la programación se han implantado procesos y metodologías de desarrollo para poder abordar las necesidades de un proyecto software. El aumento de la complejidad del desarrollo, dado por su sofisticación y la cada vez más creciente demanda de los sistemas informáticos, han hecho evolucionar los procesos y metodologías de desarrollo, gestión y mantenimiento, para poder afrontarlos en tiempo, presupuesto y con ciertas garantías de calidad.

Los modelos que se han desarrollado desde los inicios de la programación hasta la actualidad han evolucionado hasta dar lugar a un modelo evolutivo e incrementar, modelo de desarrollo en el que se fundamenta la integración continua.

1.2 MODELO EVOLUTIVO E INCREMENTAL

El primer modelo de desarrollo software del que se tiene constancia como tal, y que sirve de referente a otros modelos posteriores, es el modelo en cascada. Su versión original fue propuesta por Winston W. Royce a principios de los 70 [Royce 1970]. Este modelo propone una serie de fases o etapas de desarrollo conectadas entre sí de manera secuencial y en orden. Esto supone que, hasta que no finaliza una etapa, no empieza la siguiente, y así de principio a fin. Cabe destacar que posteriormente se han desarrollado variantes de este modelo, como el modelo en "V" o el modelo en "W", que aún se siguen usando en proyectos actuales.

EL siguiente modelo representativo de los inicios del desarrollo software es el modelo en espiral. Este modelo fue creado por Barry W. Boehm en 1986 [Boehm 1986]. La principal característica de este modelo, con respecto al modelo en cascada, es la iteración cíclica de todas sus etapas de manera reiterada. En este modelo, reconocido como un modelo evolutivo, se establecen 4 fases: Objetivos, riesgo, desarrollo y planificación. El proyecto pasa por todas sus fases en cada iteración y cada iteración representa un ciclo de mejora del producto final, aumentando su dimensión radial y, por tanto, su coste.

Estos modelos descritos anteriormente no satisfacen las necesidades de los proyectos actuales, o al menos, no en su totalidad. Desde que se establecieron metodologías de trabajo basadas en el manifiesto ágil, y con ellas la programación extrema y el proceso unificado, es necesario un modelo de desarrollo que no esté basado en predicciones sino en procesos adaptables y evolutivos. Este modelo se conoce como modelo de desarrollo evolutivo e incremental, cuyo objetivo es el de desarrollar el producto software en cortas fases de tiempo y en continuas iteraciones incrementales, de manera que se perciba su evolución desde el inicio.

En este modelo se distinguen principalmente 4 etapas por cada iteración: Etapa de inicialización, etapa de elaboración, etapa de construcción y etapa de terminación. Así mismo, se describen una serie de actividades basadas en ciertas disciplinas. Estas disciplinas pueden ser: Requerimientos, análisis y diseño, implementación, pruebas y evaluación. Si bien estas actividades están presentes, en cierta manera, en todas las etapas de cada iteración, son más representativas en unas que en otras. De esta manera se consigue una metodología de trabajo no bloqueante y, por tanto, más flexible desde el principio. En cada iteración se produce un producto entregable.

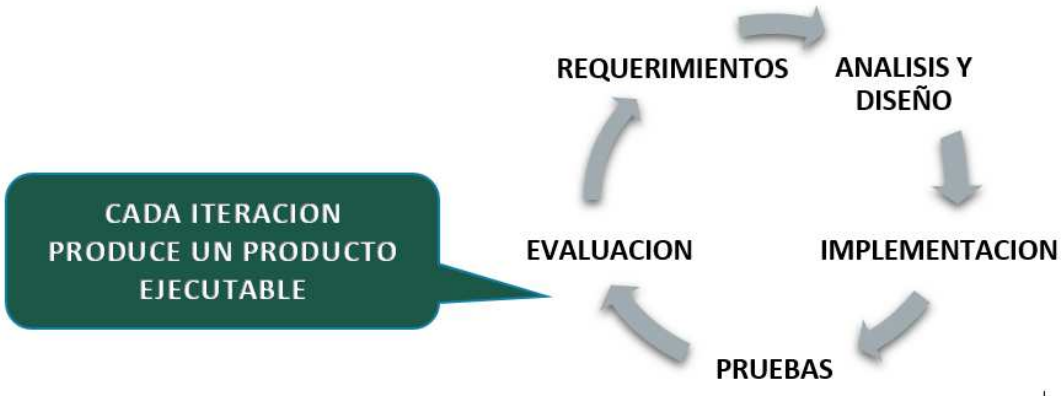


Ilustración 1: Representación del modelo Iterativo e Incremental.

La práctica de la integración continua es un ejemplo de metodología aplicada al modelo de desarrollo iterativo e incremental. Especialmente pensado para el desarrollo de software ágil; métodos como el proceso unificado, UP (*Unified Process*) [Jacobson, 1999], o la programación extrema, XP (*Extreme Programing)* [Beck, 1999], en el que el desarrollo está enfocado y dirigido por las pruebas, TDD (*Test Driven Development*) [Beck, 2002], donde se da más importancia a un software sin errores que a un software bien documentado.

1.3 MOTIVACIÓN Y OBJETIVOS DEL TRABAJO

La motivación de este trabajo se fundamenta en la evaluación de las metodologías aplicadas a la práctica de la integración continua. Estableciendo como base un modelo funcional que represente las principales características de la integración continua, mediante la implementación de un prototipo de proyecto software, se evaluará aquellas funcionalidades, centradas en la calidad del software, que puedan ser mejoradas mediante el desarrollo e implantación de procesos de automatización, con el objetivo de proponer un sistema de validación, verificación e integración lo más automatizado y asistido posible.

El objetivo principal de este trabajo consiste en la definición e implantación de aquellas mejoras sobre el modelo planteado, que puedan solucionar o minimizar los defectos o deficiencias detectadas tras la evaluación. De esta manera se consolidará un modelo, basado en la integración continua, sustancialmente mejorado con este trabajo.

El modelo básico de integración continua supone que cada cambio producido en la rama principal del repositorio tenga que ser evaluado, bajo unos criterios y requisitos previamente establecidos, y a ser posible de manera automática, para asegurar la calidad del producto y la continuidad de desarrollo. Este modelo plantea una serie de defectos que suponen un riesgo en la integridad y la continuidad del desarrollo. Estos defectos o riesgos pueden ser minimizados con la implantación de métodos de automatización, que evalúen cada cambio en el repositorio en varias etapas o fases, antes de su integración, con el objetivo de no comprometer la última versión estable del desarrollo y cumplir con los requisitos planteados.

1.4 PLANTEAMIENTO Y ALCANCE DEL TRABAJO

Partiendo de la base de un sistema de integración continua, en adelante sistema CI, el modelo representado en el prototipo del proyecto software ha de ser capaz de implementar las principales funcionalidades de dicho sistema. En el marco teórico de este trabajo se detallará todas aquellas funcionalidades, así como los requisitos necesarios, que ha de cumplir todo sistema que implemente una metodología basada en integración continua.

Teniendo en cuenta esto, se implementará en primer lugar un sistema CI para un prototipo de desarrollo dado, es decir, un proyecto software de propósito general que nos sirva de evaluación. Este sistema estará compuesto principalmente por un servidor de integración continua, en adelante *CI Server*, un repositorio principal de desarrollo y un servidor de entorno de pruebas que contenga toda la lógica de verificación y validación de los cambios en dicho repositorio. Para ello, se hará uso de herramientas O*pen Source*[1] desarrolladas para tal fin.

El sistema básico de integración continúa planteado ha de ser capaz de detectar los cambios producidos en el repositorio principal, para posteriormente, compilar y lanzar un proceso de pruebas unitarias y detectar así, fallos de compilación y errores en la construcción de la *build*[2]. A este proceso de construcción o artefacto[3], se le pueden añadir más procesos de pruebas para detectar otros posibles fallos software. Tras la verificación de dichos cambios, se notificará los resultados a los correspondientes implicados y, en caso de comprometer la integridad de la rama o de que estos cambios contengan fallos, sus autores serán los responsables de corregir o descartar los cambios correspondientes.

Este modelo inicial de integración continua se compone de una serie de fases o etapas de verificación, validación e integración del software:

[1] Licencia de código abierto. Véase en https://opensource.org/
[2] Construcción de un producto software en un punto del código.
[3] Proceso por el cual se lleva a cabo la construcción de una build.

- ✓ En una primera fase, el sistema CI ha de ser capaz de identificar un cambio en el repositorio para lanzar una serie de pruebas unitarias que verifiquen que no se rompe la compilación y sea válido a nivel de código, según el criterio establecido. De ser así, se integrará en el repositorio principal para la siguiente fase de validación funcional, en el caso contrario se descartará, notificando al propietario y grupos correspondientes.

- ✓ En una segunda fase, el sistema CI realizará una comprobación a nivel funcional del conjunto de cambios verificados en la primera fase, para un periodo de tiempo dado. De manera que se genere una *build* para la realización de pruebas a nivel funcional. Si esta *build* supera cierto umbral de calidad dado por un *scope*[4] de pruebas, según los requisitos de integración definidos, se marcará esa versión como estable o *releasable*[5], conteniendo todos los cambios desde la anterior estable o *releasable*. En el caso de que no superen dicho *scope* o umbral de calidad, se revertirán todos los cambios pertenecientes a esta segunda iteración, aunque hayan sido validos en la primera, y se informará a los correspondientes autores e implicados.

- ✓ Una posible tercera fase supondría evaluar el conjunto de *builds* estables definidas en la segunda fase, para someterlas a pruebas de carga y rendimiento, así como estabilidad y robustez, con el objetivo de obtener una versión óptima, según los requisitos de aceptación definidos. Esto supone la realización de pruebas en entornos similares al de producción y, por tanto, una implicación mayor en recursos, tiempo y proceso de pruebas. Esta iteración queda fuera del alcance de este trabajo, pero se mencionará como posible línea de investigación futura.

[4] Agrupación de casos de prueba con una finalidad común.
[5] Comúnmente establecido como versión estable o entregable.

Una manera de entender mejor el modelo de integración continúa, planteado por fases, es comparándolo con los diferentes niveles de pruebas llevados a cabo en un proceso de verificación y validación del software, previo a la integración. Como se muestra en la siguiente ilustración:

Ilustración 2: Niveles de Pruebas

Este sistema de evaluación de cambios en el repositorio es válido y cumple con los requisitos de la integración continua, pues asegura que los nuevos cambios introducidos no rompen la compilación y cumplen con los criterios funcionales de calidad establecidos. Pero no es óptimo ya que, dado el nivel de agrupación, puede penalizar de manera recursiva los cambios válidos si estos se evalúan conjuntamente con otros que no lo sean.

Se puede decir que, tal y como se ha planteado este modelo, el modelo de sistema de integración continua inicial es capaz de soportar ciertos cambios en el repositorio en ciertos periodos de tiempo, por muy eficiente que sea el sistema de integración.

Este modelo inicial puede ser mejorado mediante la inclusión de una serie de iteraciones automáticas que realicen una segunda evaluación, si al finalizar la segunda fase de validación no se completa con éxito a causa de algun fallo, no pudiendo dar por válido el producto software. El objetivo de evaluar cada cambio en una segunda iteración es la de proponer para descarte aquellos cambios que hayan introducido errores en el repositorio durante esa fase y, por tanto, comprometan la continuidad del desarrollo.

De igual manera, se podría realizar lo mismo para la tercera fase definida, descartando así aquellos cambios que comprometan el nivel de aceptación. En cualquier caso, como se comentó anteriormente en la descripción de una posible tercera fase, este planteamiento quedaría fuera del alcance de este trabajo.

El planteamiento, por tanto, sugiere mejoras para dotar al sistema CI de mayor autonomía y capacidad de validación. El sistema debería ser capaz de penalizar lo menos posible aquellos cambios del repositorio que sean válidos, tanto a nivel unitario como a nivel funcional, pero que hayan sido descartados finalmente por agruparse conjuntamente con otros que no lo sean. Debería ser capaz de identificar el cambio candidato a descarte dentro del conjunto evaluado. Para ello se hará uso de técnicas de iteraciones recursivas y evaluaciones independientes con el objetivo de evaluar los posibles candidatos a descarte en la integración final.

A continuación, se muestra una ilustración indicando las 3 posibles evaluaciones a la que podría estar sometido el sistema de integración continua, planteando mejoras basadas en iteraciones automáticas y recursivas.

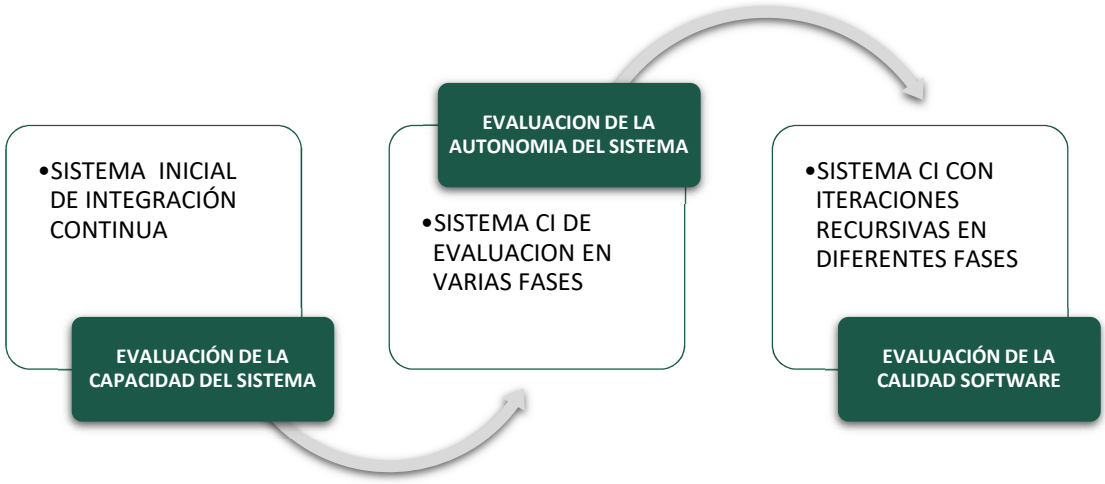


Ilustración 3: Proceso de Evaluación del Sistema CI

2. MARCO TEÓRICO: LA INTEGRACIÓN CONTINUA

Según la definición de algunos autores, la integración continua se describe como:

- ✓ "*Práctica de desarrollo de software en la que los diferentes miembros de un equipo de desarrollo integran frecuentemente su trabajo para obtener un sistema completo o parcialmente completo*" [Herbsleb and Grinter, 1999].

- ✓ "*Práctica de desarrollo software donde los miembros del equipo integran su trabajo frecuentemente, al menos una vez al día. Cada integración se verifica con una build automática (que incluye ejecución de pruebas) para detectar errores de integración tan pronto como sea posible*" [Fowler, 2006].

Esta práctica de desarrollo software está reconocida por muchas metodologías software, forma parte de las recomendaciones del UP (*Unified Process*) [Jacobson, 1999] y fue recogida como una de las 12 prácticas originales de la programación extrema XP (*Extreme Programing)* [Beck, 1999].

Básicamente, el método de la integración continua consiste en la reiteración de los siguientes pasos: Vigilar el código para su generación, construir el producto con el nuevo código verificando su integración, ejecutar las pruebas necesarias de la nueva construcción para su verificación y validación, y finalmente, publicar el nuevo producto con sus resultados.

Ilustración 4: Ciclo iterativo de la Integración Continua

2.1 LA INTEGRACIÓN CONTINUA Y EL DESARROLLO ÁGIL

La terminología del desarrollo ágil del software hace referencia al conjunto de metodologías basadas en el desarrollo iterativo e incremental. Donde tanto las necesidades y requisitos del usuario final como la solución del desarrollo del producto evolucionan mediante la colaboración entre ambas organizaciones. Una de las prácticas más importantes del desarrollo ágil es la integración continua. [Richardson and Gwaltney, 2006]

EL desarrollo ágil tiene su inicio en el manifiesto ágil[6]. Este fue creado en 2001 por 17 expertos convocados por Kent Beck, en el que se debatieron nuevas metodologías y procesos de desarrollo software como alternativa a los métodos tradicionales. Tras la reunión, se concluyeron 4 postulados con los principios de dichas metodologías alternativas, dando lugar al manifiesto ágil.

Según el manifiesto:

"Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

- ✓ *Individuos e interacciones sobre procesos y herramientas*
- ✓ *Software funcionando sobre documentación extensiva*
- ✓ *Colaboración con el cliente sobre negociación contractual*
- ✓ *Respuesta ante el cambio sobre seguir un plan*

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda." [Beck, 2002].

Con la definición de los 4 postulados del manifiesto ágil se adoptaron los 12 principios del software ágil, que se pueden consultar en la propia web del manifiesto.

Con el manifiesto ágil, se desarrollaron nuevas metodologías y procesos de desarrollo como el desarrollo dirigido por pruebas TDD (*Test Driven Development*) [Beck, 2002], en el que el desarrollo de las pruebas se realiza incluso antes que el componente software a entregar.

[6] http://agilmanifiesto.org

2.2 PRÁCTICAS BASADAS EN INTEGRACIÓN CONTINUA

La integración continua solo establece que el software se construya y se pruebe lo más rápido posible para obtener un veredicto temprano y poder detectar así los errores lo antes posible, minimizando los riesgos [Duvall, Matyas and Glover, 2007]. No obstante, muchos autores recomiendan una serie de buenas prácticas en la metodología de trabajo, para poder llevar a cabo este objetivo e implementar así realmente integración continua, beneficiándose de sus ventajas.

EL autor Martin Fowler hace una recomendación de 11 buenas prácticas para construir software con integración continua, en su artículo, *Continuous Integration* [Fowler, 2006]. Estas, se comentan a continuación:

1. "MANTENER UN ÚNICO REPOSITORIO DE CÓDIGO"

Es aconsejable usar el mismo sistema de control de versiones para todos los proyectos software de una organización. Así mismo, se recomiendan que todas las fuentes necesarias utilizadas para construir el producto se almacenen en un único repositorio, normalmente repositorio principal o *master*.

Estas fuentes no solo incluyen el código del software entregable, sino también, los archivos de configuración, los scripts de instalación, las librerías de terceras partes, la información de la base de datos, e incluso el desarrollo del *testing* para las pruebas software.

2. "AUTOMATIZAR LA CONSTRUCCIÓN"

No es un requisito de la integración continua que la compilación y la construcción de la *build* se haga de forma automática, sin embargo, se recomienda que así sea. De esta manera se evitan errores humanos al generar los artefactos[7] de código.

[7] Empaquetado de los componentes de un producto software.

La construcción de la *build* para generar los correspondientes artefactos de código con sus correspondientes características y plataformas, suele incluir la compilación y las copias de los ficheros necesarios, así como la inserción de datos correspondientes en una base de datos.

3. "UTILIZAR PRUEBAS AUTOMATIZADAS"

Al igual que en la anterior recomendación, el proceso de automatización de pruebas no es un requisito impuesto por la integración continua, pero si se recomienda el uso de un mínimo *scope* de pruebas realizadas de manera automática, como proceso de construcción de la *build*, con el objetivo de no introducir nuevos errores en la construcción.

Este proceso de pruebas automático suele incluir un análisis estático de código, así como test unitarios y de componente para su verificación. Aunque no se limita solo a esto, pues se pueden incluir aquellas pruebas funcionales de integración o de aceptación que se consideren oportunas, siempre y cuando no se alargue demasiado el proceso de construcción de la *build*.

4. "ACTUALIZAR EL REPOSITORIO A DIARIO"

Como medida de comportamiento hacia los desarrolladores, se recomienda que estos suban al repositorio principal sus cambios, al menos una vez al día. De esta manera se aseguran que el repositorio local no varíe mucho con respecto al repositorio principal.

Una vez comprueben que sus cambios son válidos en local, deberían subirlos al repositorio principal, antes de incluir más cambios. Los cambios han de ser atómicos y la descripción ha de guardar cierta relación con su contenido.

5. "CADA CAMBIO HA DE CONSTRUIRSE EN LA MÁQUINA DE INTEGRACIÓN"

Esto supone concienciar a todos los miembros de la organización de que han de trabajar siempre con la última versión del repositorio y sus cambios han de compilar con la última versión estable.

Supone también que cada desarrollador ha de clonar su repositorio local con el central constantemente, para evitar conflictos que puedan darse cuando 2 o más desarrolladores editen los mismos ficheros, o partes de código a la vez.

6. "ARREGLAR LOS ERRORES DE LA CONSTRUCCIÓN INMEDIATAMENTE"

Cada desarrollador es responsable de los cambios que sube al repositorio principal. Si estos cambios introducen errores en el repositorio o rompen la compilación, es responsabilidad suya corregirlos o revertirlos hasta que se puedan incluir sin fallos. De lo contrario, bloqueará el trabajo de los demás desarrolladores que no podrán incluir sus cambios en el repositorio principal hasta que este sea corregido.

7. "CONSEGUIR CONSTRUCCIONES RÁPIDAS"

Es importante que el sistema de construcción de la *build* sea capaz de realizar la construcción, incluyendo compilación y pruebas, lo antes posible. De esta manera no se producirán cuellos de botella en el sistema de integración, especialmente en aquellos momentos en el que todo el equipo decida subir sus cambios al repositorio principal al mismo tiempo, cosa que suele pasar con bastante frecuencia.

Otra medida que puede adoptarse es incluir varias etapas en la construcción, de manera que sea la principal la más inmediata, para poder ofrecer la *build* lo antes posible, y las demás etapas, con una mayor carga de pruebas, se realicen posteriormente, e incluso en paralelo si el sistema lo permite.

8. "PROBAR EN UN CLON DEL ENTORNO DE PRODUCCIÓN"

El producto software se ha de probar en un sistema que sea lo más parecido al de producción. Los artefactos generados se han de construir para el hardware soportado, y estos han de ser probados con un entorno definido.

Actualmente, cada vez se utilizan más técnicas de virtualización que proporcionan clones del sistema y entornos en producción, asegurando que las pruebas realizadas para una versión concreta, es válida para un sistema y un entorno definido.

9. "FACILITAR LA OBTENCIÓN DE LA ÚLTIMA VERSIÓN"

Ha de facilitarse la última versión del producto a todos los miembros de la organización, de manera que todos estén trabajando siempre sobre la última versión. También ha de facilitarse que cualquiera pueda ejecutar y probar el producto fácilmente, para poder ver los últimos cambios de la última versión estable.

10. "CONOCER EL ESTADO DE LA ULTIMA VERSIÓN"

De la misma manera que se ha de facilitar la última versión estable del producto software, para su ejecución y la realización de pruebas, es importante notificar de alguna manera el estado de la última versión.

Esto incluye los resultados de las pruebas y las ultimas funcionalidades y características añadidas. Se suele utilizar webs o paneles para que cualquiera pueda consultar esa información cuando lo desee.

11. "AUTOMATIZAR EL DESPLIEGUE"

Es aconsejable tener automatizado este proceso, ya que se evitan errores humanos. Para ello, es común el uso de scripts que automaticen el proceso de despliegue del software, en los momentos oportunos y con los parámetros definidos, establecidos previamente por el personal de configuración del sistema.

Además de ello, es aconsejable disponer de mecanismos de *rollback* para evitar los posibles problemas dados por el despliegue de la nueva versión, y así poder volver al estado anterior fácilmente.

Estas recomendaciones para implementar integración continua representan la visión que comparten algunos autores, como el citado Martin Fowler, para llevar a cabo de manera genérica esta disciplina. En realizad esto dependerá de factores como, la cantidad de equipos e implicados en el proyecto, los recursos que se le pueden dedicar al desarrollo y las pruebas, o las características intrínsecas del proyecto en sí. Realmente el requisito impuesto para que sea integración continua es incluir código constante y obtener un veredicto lo antes posible para intégralo o descartarlo.

2.3 ENTREGA CONTINUA Y DESPLIEGUE CONTINUO

A menudo se hace referencia a los términos de Integración Continua y Entrega Continua como si fueran lo mismo, y en realidad se puede entender así, dado que el objetivo final es la entrega del producto que previamente ha sido integrado en la rama principal de desarrollo. Autores como Jez Humble o David Farley [Humble and Farley, 2011], entienden este concepto conjuntamente, denominándolo "Entrega Continua". El concepto del despliegue continuo adquiere más matices, ya que el proceso de despliegue de un producto software depende de muchos factores, tanto técnicos como de la propia organización, y la mayoría de las organizaciones prefieren usar sus propios métodos, normalmente asistidos por un equipo del personal de operación o configuración.

En cualquier caso, el objetivo de integración continua, entrega continua y despliegue continuo, debe entenderse como un proceso continuo y único. Una tubería o *pipeline* en el que en cada fase se añaden una sucesión de pruebas para una finalidad concreta [Laster, 2017]. Por ejemplo, en la fase del *pipeline* correspondiente al de entrega continua no se prueba lo que ya se haya probado en la fase de integración. El objetivo de esta fase sería la de verificar que el producto sea entregable, sin que necesariamente vaya a desplegarse el resultado final. En cambio, la fase correspondiente al despliegue continuo ha de realizar esta tarea de manera automática, como consecuencia de la validación del entregable por parte de la etapa de la entrega continua.

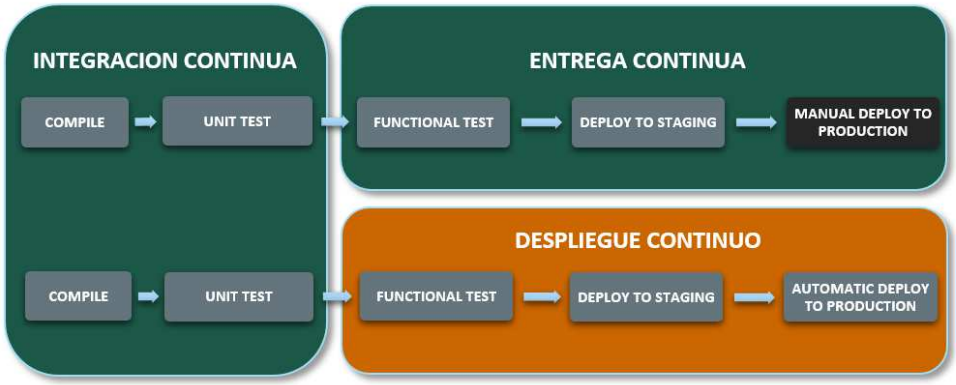


Ilustración 5: Integración Continua, Entrega Continua y Despliegue Continuo.

2.4 VENTAJAS Y DESVENTAJAS

Son muchas las ventajas que ofrece la implantación de la integración continua en una organización software. Algunas de ellas ya han sido comentadas, al mencionar los objetivos principales de la integración continua, otras se pueden deducir de las características de la misma. Pero si hay que enumerar las aportaciones más importantes que ofrece la integración continua, estas serían:

- ✓ Flexibilidad; Al estar enfocado en procesos ágiles y en métodos de desarrollo evolutivos e incrementales, el producto software se adapta a los cambios producidos por las necesidades del proyecto. Como cambios en los requisitos, o en los requerimientos durante el desarrollo.

- ✓ Calidad; Uno de los principales objetivos de la CI es evaluar el producto constantemente para no introducir nuevos fallos. Esto no solo implica verificar los nuevos cambios respecto a sus requisitos, sino también verificar, mediante pruebas de regresión, que no se introducen nuevos fallos. Esto da lugar a una reducción de costes por la detección temprana de fallos introducidos.

- ✓ Reducción del riesgo; El riesgo es intrínseco a los proyectos software por su carácter temporal y único. Tiene su origen en la incertidumbre presente en todos los proyectos [PMI, 2005]. Con la integración continua, se implantan numerosos hitos de entrega a corto plazo, de manera que, la entrega final se realiza desde el principio. Esto hace que se pueda agilizar el proceso, si se prevé que no se van a cumplir los hitos, haciendo uso de procesos ágiles.

- ✓ Reducción de costes; al ofrecer un software de calidad sin errores se reduce el coste derivado de la detección de errores demasiado tarde. "Cuanto más se tarda en detectar un error, más costosa será su reparación" [Grady, 1999].

Pese a todas las ventajas que ofrece la implantación de la integración continua en una organización de desarrollo software, también tiene ciertas desventajas. Estas han de conocerse antes optar por esta metodología e implantarla como modelo de trabajo, de lo contrario, podría no llegar a implantarse del todo y fracasar en sus objetivos. A continuación, se detallan algunas de las desventajas más características de la integración continua:

- ✓ Desarrollo del entorno; El entorno en sí mismo representa un proyecto software, y eso supone un gran esfuerzo y trabajo por parte de la organización. En ocasiones hay que dedicar parte del equipo del proyecto en desarrollar y mantener esta infraestructura. A medida que avanza el desarrollo del proyecto, se requieren mayores necesidades en el sistema de integración y, por consiguiente, estas han de implantarse conjuntamente con el software desarrollado.

- ✓ Sobrecarga del sistema; Se necesita una mayor infraestructura para soportar la reiteración de pruebas de validación e integración, así como el almacenamiento de los artefactos entregables. En relación con el comentario anterior, las necesidades en cuanto a la infraestructura pueden variar a lo largo del proyecto, por lo que han de ser bien redimensionadas en todo momento para que no se produzcan cuellos de botella en momentos de máxima integración, como consecuencia de hitos y fechas de entrega.

- ✓ Degeneración de la arquitectura; Algunos autores alegan que el uso de la integración continua hace que los desarrolladores se centren más en las entregas a corto plazo y eso suponga una degeneración de la arquitectura del proyecto software [Olsson, 1999].

- ✓ Desarrollo de pruebas; No se debe entender como una desventaja en sí, pero si hay que tener en cuenta que, si se quiere ofrecer un software de calidad, es necesario el desarrollo de pruebas para tal fin. Existen herramientas de *testing* que ayudan y agilizan el proceso de pruebas, no obstante, en la mayoría de los proyectos no es suficiente y es necesario un desarrollo de pruebas dirigido al proyecto en cuestión. Es por ello que, procesos de desarrollo como el TDD (*Test Driven Development*) [Beck, 2002], y sus derivados, funcionan muy bien con la integración continua.

Para finalizar esta sección del marco teórico de este trabajo, cuya finalidad es la de entender los objetivos y las motivaciones de la integración continua, como metodología referente del desarrollo software, y concebir esta como una práctica necesaria para poder llevar a cabo la automatización de procesos de verificación, validación e integración del software, se va a tomar como referencia un modelo inicial que implemente integración continua, mediante un prototipo de proyecto software, para su posterior evaluación. Esto se realizará en la siguiente sección.

3. IMPLEMENTACIÓN DE UN SISTEMA DE INTEGRACIÓN CONTINUA

En este capítulo se va a implementar un sistema de integración continua que cumpla con los requisitos definidos en el marco teórico de este documento, capítulo 2. Para ello, en primer lugar, se van a establecer los casos de uso que el sistema ha de cumplir, según las recomendaciones del marco teórico, para llevar a cabo la implementación de dicho sistema y, posteriormente, se detallará un prototipo de proyecto software que sea válido para la representación, evaluación y puesta en marcha en el sistema de integración continua.

Este prototipo de proyecto software, que se comentará en detalle más adelante, básicamente representa un producto software que se pueda compilar, ejecutar y probar, en un sistema o hardware definido, de manera que pueda ser desarrollado y completado a medida que avance el proyecto. Por tanto, parte de la premisa de que es un producto software incompleto. Cabe destacar que este prototipo de proyecto software es un ejemplo o caso de uso, de entre muchos posibles, necesario para la puesta en marcha del sistema de integración continua sobre un proyecto software, y poder realizar así el análisis y la evaluación de este sistema.

3.1 REQUISITOS DEL SISTEMA DE INTEGRACIÓN CONTINUA

A continuación, se van a definir los requisitos necesarios para la implementación de un sistema de integración continua, tomado como referencia las recomendaciones establecidas en apartado 2.2 de este documento, *Continuous Integration* [Fowler, 2006].

Para ello, es necesario realizar un modelo del sistema que represente dichas necesidades mediante la elaboración de un diagrama de casos de uso y así poder evaluar punto por punto, cada una de las recomendaciones establecidas que este sistema ha de cumplir una vez sea implementado. Esto podría modelarse como los requisitos del usuario.

También es importante enumerar los requisitos software y las herramientas necesarias para la implementación de dicho sistema que, aunque la elección y el uso de dichas herramientas software no represente requisitos necesarios para la implementación, si representa un ejemplo de implementación en nuestro caso. Esto podría modelarse como parte de los requisitos del sistema.

3.1.1 REQUISITOS DEL USUARIO

Se entiende por requisitos de usuario aquellas declaraciones en lenguaje natural, de carácter no necesariamente muy técnico, que pueden ser modelados mediante la representación de diferentes diagramas. Como es el caso del diagrama de casos de uso, donde se definen, con un carácter general, los diferentes servicios del sistema y las restricciones bajo las que ha de operar.

En este sentido y, tomando como referencia las 11 recomendaciones planteadas en el marco teórico, se va a plantear un modelo de casos de uso que relacione dichas recomendaciones:

1. "Mantener un único repositorio de código"
2. "Automatizar la construcción"
3. "Utilizar pruebas automatizadas"
4. "Actualizar el repositorio a diario"
5. "Cada cambio se construye en la máquina de integración"
6. "Arreglar errores de la construcción inmediatamente"
7. "Conseguir construcciones rápidas"
8. "Probar en un clon del entorno de producción"
9. "Facilitar la obtención de la última versión"
10. "Conocer el estado de la última versión"
11. "Automatizar el despliegue"

Para ello, se van a establecer unos 9 casos de uso, limitado por un sistema, sistema de integración continua, y dos actores principales que interaccionarán, mediante relaciones de asociación, con todos estos sistemas, directa o indirectamente. De la misma manera, dichos sistemas se relacionan entre sí mediante relaciones de uso, generalización o dependencia.

El diagrama de casos de uso que representa el escenario del sistema de integración continua quedaría como se muestra en la siguiente ilustración:

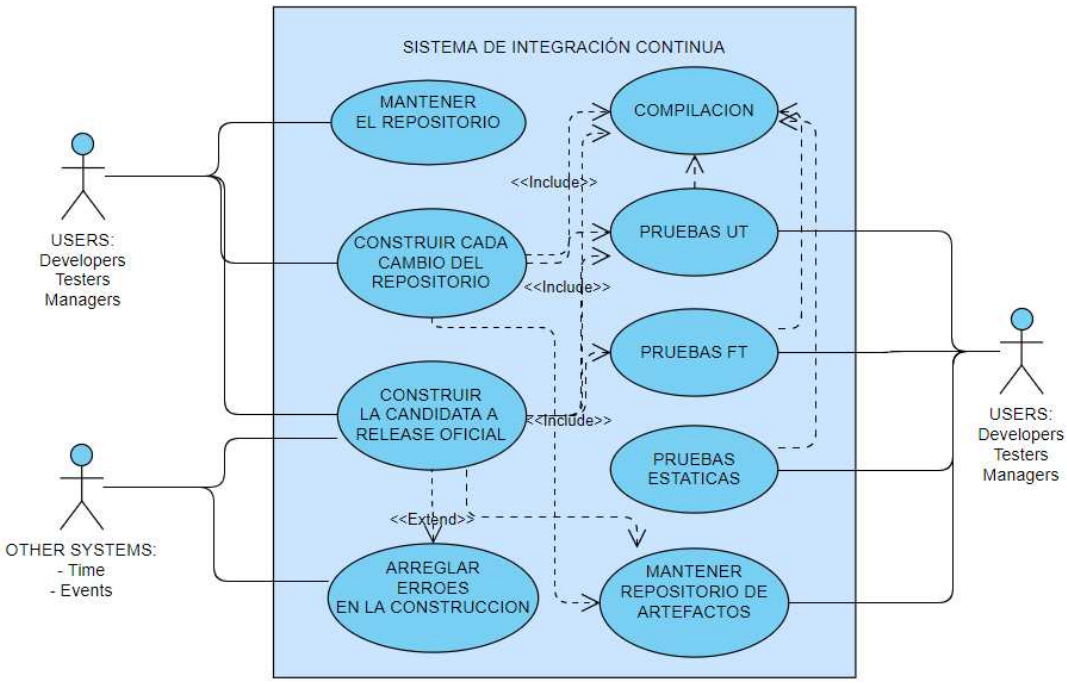


Ilustración 6: Diagrama de casos de uso. Sistema de integración continua.

En este diagrama de casos de uso podemos apreciar 2 actores principales que van a interaccionar con el sistema de integración continua. Estos son; por un lado los usuarios, como los desarrolladores, *testers*, managers, etc., gestionados mediante perfiles y, por otro lado, otros sistemas externos, que representen interacciones desasistidas por una persona, como puede ser el tiempo o, en general, cualquier tipo de evento lanzado por otro sistema.

En cuanto a los casos de uso que componen el sistema de integración continua, tenemos los siguientes:

- ✓ Mantener el repositorio de código. Este caso de uso ha de satisfacer las recomendaciones 1 y 4. Es decir, ha de mantener un único repositorio de código y facilitar a los usuarios que estos puedan actualizar el repositorio a diario. Aunque esta acción parece más bien responsabilidad de los usuarios, el sistema puede favorecer esta acción manteniendo la rama principal del repositorio desbloqueada, es decir, ante un fallo tendrá que bloquearlo para corregirlo lo antes posible y poder volver a desbloquear el repositorio.

- ✓ Construir cada cambio del repositorio. Este caso de uso relaciona las recomendaciones 2, 3, 5 y 7, ya que ha de automatizar la construcción con cada cambio que se añada al repositorio y, para ello, tendrá que compilar y lanzar pruebas automáticamente en la máquina de integración, al menos, pruebas unitarias. De esta manera se podrá realizar las construcciones lo más rápido posible con ciertas garantías de calidad.

- ✓ Construir la candidata a la versión oficial del producto. Las construcciones han de ser rápidas para garantizar la recomendación 7, no obstante, se ha de garantizar con el mayor grado de calidad cada cierto tiempo o cuando se considere, una construcción que garantice una versión estable del producto, mediante un pipeline de pruebas tanto unitarias como funcionales, dando lugar a una candidata a versión o *release*[8] oficial. Esto le lleva al sistema más tiempo por lo que no puede realizarse por cada cambio en el repositorio. Este caso de uso relacionaría las recomendaciones 2, 3, 5 y 8.

- ✓ Mantener el repositorio de artefactos. Este caso de uso se encarga de generar un artefacto software por cada construcción, bien sea por cada construcción de cada cambio en el código, por cada versión candidata o por cada estable. Y ha de gestionarlas para que cualquier usuario o sistema externo pueda acceder a dichos recursos. Este caso de uso relacionaría las recomendaciones 9 y 10.

[8] Proceso de oficialización de una versión de un producto software.

- ✓ Otros sistemas de QA[9], como compilar, ejecutar pruebas estáticas, pruebas unitarias y pruebas funcionales. Estos casos de uso relacionarían las recomendaciones 3 y 8 ya que, su ejecución ha de ser automatizada en un clon del entorno de producción. Esto se consigue mediante la interacción de estos casos de uso, con los de construcción de cada cambio o de la candidata a oficial. No obstante, se da libertad a usuario para que pueda lanzar las pruebas que considere, del artefacto o punto de código del repositorio que quiera, en el clon del sistema de producción, a modo de *testing* asistido.

- ✓ Arreglar errores en la construcción. Este caso de uso relaciona principalmente la recomendación 6. Si bien, esta recomendación dice que el propietario del cambio que rompa la integración por errores es responsable de su corrección, de alguna manera el sistema debería ser capaz de detectar qué cambio es el causante del fallo y descartarlo para no influir en los otros cambios. Este caso de uso, se mejorará en el siguiente capítulo mediante la introducción de mejoras en el sistema de integración continua. De momento, solo se encargaría de notificar al propietario del cambio, que ha introducido un fallo en el repositorio y que lo ha de arreglar lo antes posible.

Nótese que faltaría definir un caso de uso que relacione la recomendación 11, "Automatizar el despliegue". En este sentido, este caso de uso está más relacionado con el despliegue continuo que con la integración continua, quedando fuera del alcance del trabajo, como se mencionó en la introducción del mismo.

3.1.2 REQUISITOS DEL SISTEMA

Los requisitos software del sistema, debe entenderse en este contexto más que un requisito, una recomendación a tener en cuenta ya que, si bien se pueden desarrollar todas las herramientas necesarias para la implementación de un sistema de integración continua, existen numerosas herramientas que pueden ser útiles para llevar a cabo esta tarea.

[9] Siglas en inglés de Quality Assurannce (Aseguramiento de la Calidad).

Como requisito impuesto en este trabajo, se impone que estas herramientas sean de licencia *Open Source*. Estas herramientas van a ser necesarias para llevar a cabo la implementación del sistema de integración continua, por tanto, se hará uso de aquellas funciones que dichas herramientas nos puedan aportar como solución para nuestro objetivo final. A continuación, se enumerarán las que se han utilizado en este trabajo.

Estos productos o herramientas software, con licencia O*pen Source,* son:

- ✓ GIT: Como sistema de control de versiones para el repositorio principal de desarrollo. GIT, https://git-scm.com, es un sistema de control de versiones distribuido, por lo que el repositorio no solo estará contenido en el servidor central sino también en los repositorios locales de cada sistema de desarrollo. Lleva un registro de todos los cambios en el repositorio y cualquiera de ellos se puede revertir, en caso de comprometer la calidad del producto final. Otras alternativas a GIT podrían ser Subversion, https://subversion.apache.org, o Mercurial, https://www.mercurial-scm.org.

- ✓ JENKINS: Como sistema de integración continua. Jenkins, https://jenkins.io, es un sistema de automatización de procesos de desarrollo, especialmente orientado a integración continua, aunque también válido para entrega continua y despliegue continuo. Su principal ventaja es la cantidad de *plugins*[10] de software de terceros que se le pueden añadir para personalizar el sistema. Otras alternativas a Jenkins podrían ser Travis CI, https://travis-ci.org, Circle CI, https://circleci.com, Hudson CI, http://hudson-ci.org/ o Gitlab, https://about.gitlab.com/product/continuous-integration.

- ✓ MAVEN: Como herramienta de automatización de compilación, ejecución de pruebas y empaquetado para nuestro proyecto JAVA. Maven, https://maven.apache.org, es una herramienta de construcción y gestión de proyectos JAVA. Otras alternativas a Maven podrían ser Make, https://www.gnu.org/software/make, o Ant, https://ant.apache.org.

[10] Componente software que añade una funcionalidad concreta a un producto software.

- ✓ JUNIT: Como *framework*[11] para el desarrollo de test unitarios, concretamente JUnit 4. JUnint, https://junit.org/junit4/, es un *framework* para la realización de pruebas unitarias en proyectos JAVA. En función del lenguaje de desarrollo del proyecto, otras alternativas podrían ser PyUnit para Python, http://pyunit.sourceforge.net, o NUnit para .NET, https://nunit.org.

- ✓ Jacoco: Como herramienta de análisis estático de cobertura de código, https://www.jacoco.org/. Jacoco es un software que se integra con Maven para el análisis de código JAVA, principalmente para el análisis de cobertura de código. También se puede integrar con Jenkins a través de plugins. Otras alternativas para elaborar informes de cobertura podrían ser Coverage.py, https://coverage.readthedocs.io/en/v4.5.x , o OpenCppCoverage, https://github.com/OpenCppCoverage/OpenCppCoverage.

- ✓ Otras herramientas que no se van a usar, pero que merecen destacar por su importancia son: NEXUS OSS, como herramienta para el repositorio de artefactos. GERRIT, https://www.gerritcodereview.com, como herramienta de revisión de código. Y SELENIUM, http://www.seleniumhq.org, CUCUMBER, https://cucumber.io, o JMETER http://jmeter.apache.org, como *frameworks* de *testing* para *tests* funcionales y de aceptación.

- ✓ LINUX: Tanto el servidor que contenga el repositorio principal de desarrollo como el sistema de integración continua, así como el principal ejecutor de pruebas unitarias y funcionales, estará contenido en un sistema servidor Ubuntu Linux, 18.04. Sistema virtualizado en una máquina VirtualBox 5.2. Cualquier otro sistema operativo como alternativa sería válido, si puede ejecutar Jenkins, GIT y la versión SDK o, en este caso, JDK compatible a la que se especifique, siendo en nuestro caso la 1.8.

[11] Desarrollo de estandarización para abordar una tarea común.

- ✓ El entorno de desarrollo puede ser cualquier IDE de desarrollo para Java, que contenga las principales librerías de la edición estándar, así como las de JUnit4. En este trabajo se ha optado por Eclipse, por su fácil integración con GIT, Maven, y otras herramientas para la elaboración de diagramas UML. Pero, en definitiva, cualquier otro IDE podría ser válido y se dejaría a la elección del desarrollador.

3.2 MODELO DEL SISTEMA DE INTEGRACIÓN CONTINUA

Una vez establecidos los requisitos necesarios para el desarrollo del sistema de integración continua, es necesario la realización de un modelado del sistema que represente el comportamiento previo a la implementación y satisfaga los requisitos funcionales de usuario impuestos. Si bien, el diagrama de casos de uso representado en la sección anterior forma parte también del modelado de un sistema a alto nivel, se ha decido proponerlo ahí para diferenciar, por una parte, los requisitos y definiciones del modelo y, por otra parte, el comportamiento de este en función de dichos requisitos y definiciones previas.

Es por ello que en esta sección se va a modelar el comportamiento que ha de representar el sistema de integración continua. Sistema que será implementado y evaluado en las siguientes secciones, en función de los modelos de comportamiento definidos ahora.

3.2.1 CONSTRUCCIÓN DE CADA CAMBIO EN EL REPOSITORIO

Como requisito impuesto en la fase de análisis, cada cambio añadido al repositorio principal se tiene que construir en el sistema de integración, y este ha de probarse en un clon del sistema de producción. Las pruebas que se van a correr en cada construcción son pruebas unitarias ya que, con ello aseguramos cierto nivel de calidad a la vez que agilizamos la integración con construcciones rápidas.

El siguiente diagrama de secuencia muestra el comportamiento del caso de uso definido para llevar a cabo la construcción de cada cambio en el repositorio:

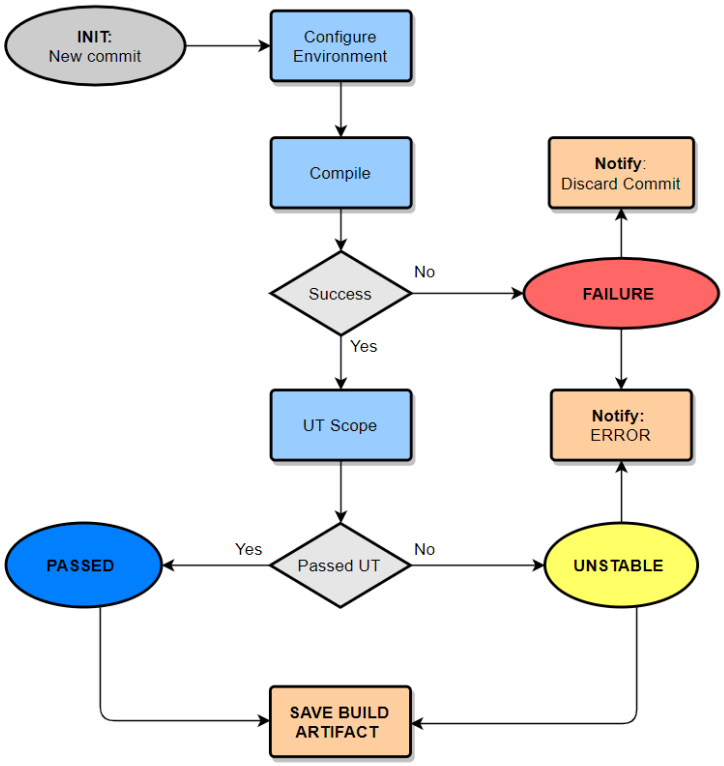


Ilustración 7: Diagrama de secuencia. Construcción de cada cambio del repositorio.

Se puede apreciar que el inicio del proceso se produce cuando hay un nuevo cambio en el repositorio. En ese momento, se ejecuta el proceso de configuración del entorno y de compilación. Si hay algún error en este proceso, se descarta el cambio y se notifica a los implicados del cambio de dicho fallo. En caso de que no falle la compilación, el proceso pasa a la ejecución de pruebas unitarias. La ejecución de dichas pruebas unitarias puede dar lugar a dos resultados, que pasen todas las pruebas, en ese caso el resultado sería "*passed*" o que falle alguna de ellas, en cuyo caso el resultado sería "*unstable*". Hay que tener en cuenta que se puede configurar un umbral de porcentaje máximo de casos fallidos para considerar el resultado como fallido, en lugar de inestable, (en nuestro modelo esto no se va a tener en cuenta).

En el caso de que el resultado sea inestable, no se descartará el cambio y se notificará a los implicados para que procedan a añadir otro cambio que lo solucione. En cualquiera de los dos casos anteriores, dicho cambio será integrado en el repositorio y el artefacto generado se almacenará en el repositorio de artefactos como snapshot [12]de la versión en cuestión.

3.2.2 CONSTRUCCIÓN DE LA *BUILD* CANDIDATA A OFICIAL

La construcción de una *build* candidata a versión oficial puede ser lanzada manualmente por el personal de *release*, o bien, puede ser lanzada cada cierto tiempo, por ejemplo, al final de la jornada laboral, o programada para que se ejecute durante el fin de semana. Se ha de realizar así ya que, teóricamente su ejecución es más larga y pesada para el sistema que la construcción de cada cambio, por el hecho de contener un mayor número de pruebas a ejecutar.

En nuestro caso, para esta construcción se ejecutarán tanto pruebas unitarias como pruebas funcionales y en este caso, a diferencia del anterior, no se admite que ningún test falle, puesto que, no puede existir construcciones inestables para una candidata a versión oficial. Hay que resaltar también que este proceso lleva implícito el cambio de versión de producto, esto se detallará más adelante.

El siguiente diagrama de secuencia muestra el comportamiento del caso de uso definido para llevar a cabo la construcción de cada candidata a la versión oficial:

[12] En este contexto corresponde a la generación de artefactos de código no oficiales.

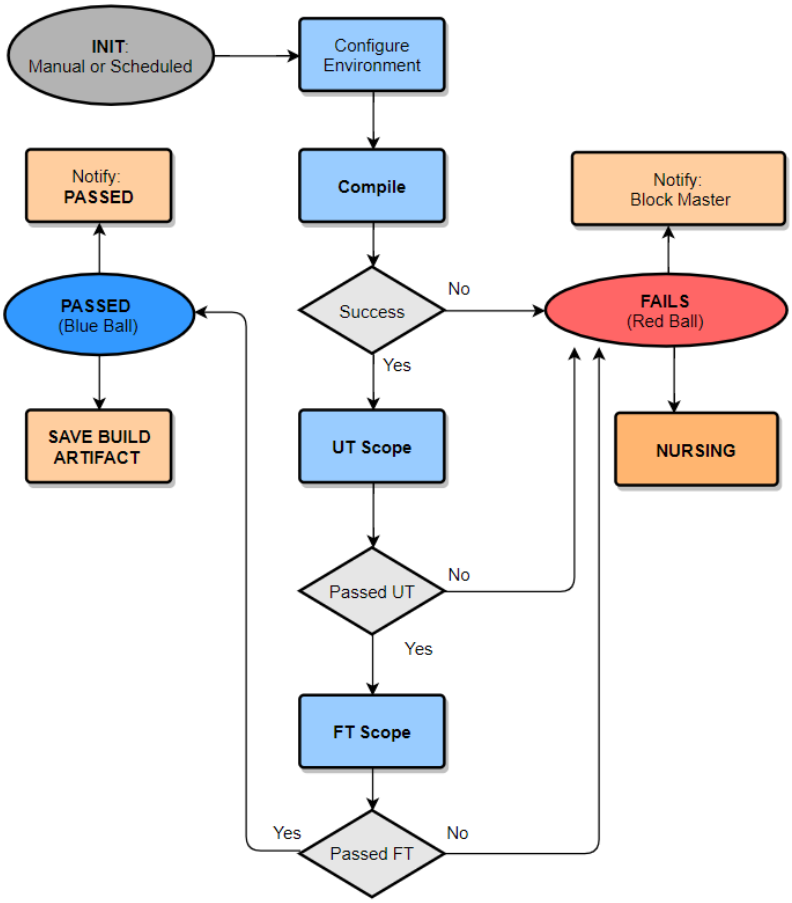


Ilustración 8: Diagrama de secuencia. Construcción de la candidata a versión oficial.

El proceso se inicia, bien por un *trigger*[13] manual, o bien por un *trigger* programado. Continúa con la configuración del entorno y la compilación del producto software. A continuación, ejecuta tanto el *scope* definido para las pruebas unitarias como para las pruebas funcionales, es este orden, y si alguno falla, se bloquea inmediatamente la rama máster del repositorio y se inicia el proceso de "*nursing*[14]", cuyo objetivo es el de identificar los cambios que han producido dicho fallo e intentar arreglar el repositorio. Este proceso de *nursing* forma parte de las mejoras impuestas al modelo inicial y se comentará en el siguiente capítulo. En caso de que el proceso finalice de manera exitosa y, por tanto, hayan pasado todas las pruebas, se oficializa dicha versión como candidata y se almacenan los artefactos generados en el repositorio de artefactos candidatos a *release*.

[13] Re ejecución de un proceso con los mismos parámetros.
[14] Proceso por el cual se identifican y corrigen posibles fallos en el proceso de desarrollo.

3.2.3 DISEÑO DEL PROCESO DE *RELEASE*

El proceso de *Release* se realiza en dos fases, y lleva consigo el proceso de cambio de versión del producto, independientemente de la numeración que se defina, que por lo general consta de versión, subversión y revisión.

En una primera fase, el sistema de integración continua mediante ejecución manual o ejecución programada, realiza el cambio de versión del producto y ejecuta el proceso anterior definido como construcción de la *build* candidata a oficial. Proceso que, en este caso, se abreviará como *build release*. Si este proceso falla, se inicia el proceso de *nursing*, tal y como se ha comentado anteriormente. Si este proceso sigue fallando podrá hacerse un *retry* hasta que se solucione el fallo o se desista y no se pueda completar la acción de *relase*. En el caso de que el proceso de *build release* sea *passed*, se procede al cambio de versión del producto almacenando el artefacto de la *build* en el repositorio de candidatas a oficiales y se genera un nuevo repositorio para los cambios pertenecientes a la nueva versión.

En una segunda fase, se lleva a cabo a la oficialización de la versión candidata a oficial, mediante el proceso "*promote*[15] *release candidate*". Este proceso es no puede ser automatizado, dada su importancia, por lo que ha de realizarse manualmente por un miembro del personal de release. Supone la gestión del repositorio artefactos de candidatos a release, para la build en cuestión, al repositorio de versiones oficiales. En esta fase no se lleva a cabo ninguna evaluación, salvo las que quieran realizarse manualmente o mediante los *Jobs* de QA. Cabe destacar que, en un entorno de producción real, este procedimiento llevaría implícito el desarrollo de la documentación asociada a la release en cuestión. En este trabajo esto no se ha tenido en cuenta.

[15] Proceso de release de una versión de un producto software.

A continuación, se muestra un diagrama de estados finitos[16] perteneciente a la primera fase del proceso de *release* descrito anteriormente:

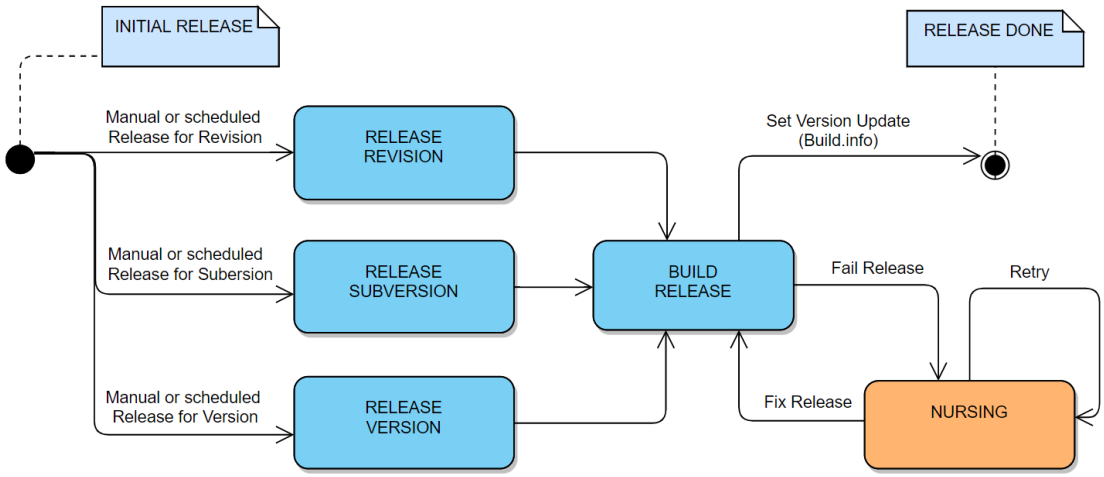


Ilustración 9: Diagrama de estados. Proceso de release.

En este diagrama de estados finitos, se aprecia un estado inicial, que representa la acción cuando se quiere oficializar una release, y un estado final, que representa la acción finalizada con éxito. También se representan, por un lado, 3 estados de release, que podría entenderse como uno solo con parámetros diferentes, y otro lado, el estado de "*build release*" definido anteriormente. También se representa un estado de "*nursing*" que se inicia cuando el estado anterior ha fallado, cuyo comportamiento y análisis se comentará en detalle en el siguiente capítulo. En este caso no está representado el estado fallido de proceso de *release*, por facilitar la comprensión del diagrama.

[16] Un diagrama de estados finitos representa un inicio y un final simpre alcanzable.

3.2.4 GESTIÓN DEL REPOSITORIO DE ARTEFACTOS

El repositorio de artefactos ha de contener todas aquellas *builds* que hayan sido generadas, bien por el proceso de construcción de la *build*, o bien por el proceso de construcción de cada candidata a *release* y, por supuesto, la construcción las *builds* oficiales o *releases*. Las *builds* generadas por cada *commit*[17] o cambio en el repositorio reciben el nombre de "*snapshot*" o *builds* intermedias. Es aconsejable que el nombre del *snapshot* contenga o haga referencia de alguna manera al identificador del *commit* al que se refiere la compilación, de esta manera es más sencilla la asociación a la hora de realizar pruebas posteriores. Además, estas deberían estar agrupadas por la versión a la que pertenecen, de manera que estén contenidas en un directorio que referencie de alguna manera a dicha versión del producto.

En cuanto a las *builds* candidatas a oficiales, estas han de contener la versión de producto que oficializan y alguna referencia inequívoca que las identifique, como por ejemplo la fecha o el *timestamp* de cuando se generaron, ya que puede haber varias *builds* candidatas para la misma versión. Cosa que no pasa con las *build* oficiales, estas están referenciadas inequívocamente por la versión del producto y nada más, aunque puede contener alguna información adicional como indicador de estable u oficial.

A continuación, se muestra un diagrama de estructuras de directorios, que representa cómo se organiza y gestiona el repositorio de artefactos.

[17] Cambio atómico en una rama de un repositorio.

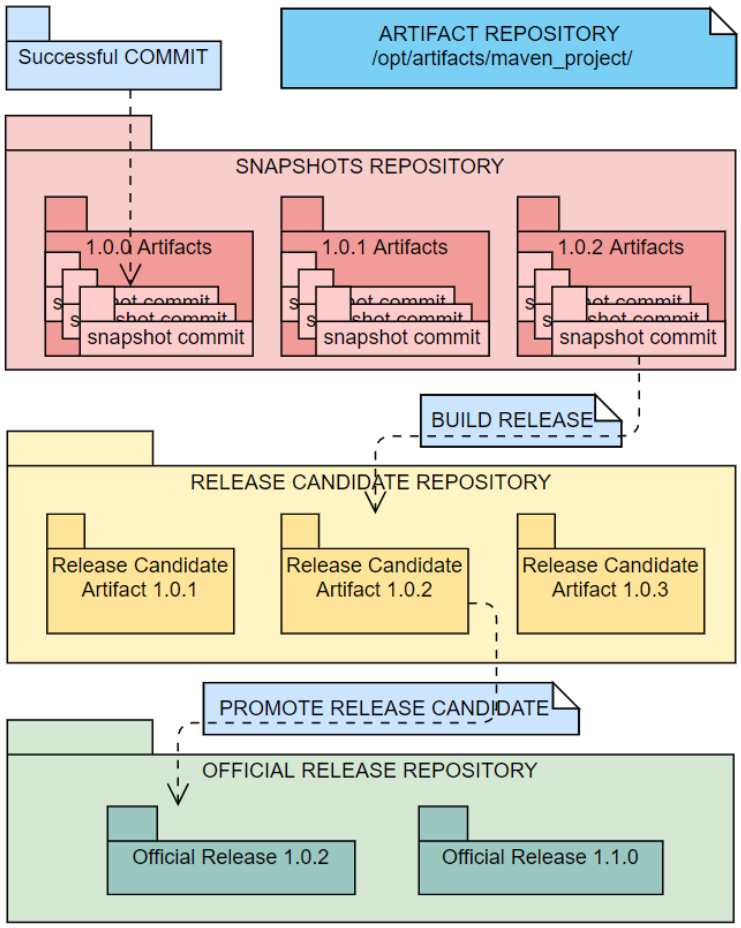


Ilustración 10: Repositorio de artefactos. Diagrama de estructuras de directorios.

En el diagrama se representan 3 contenedores principales, uno para el repositorio de snapshots, otro para el repositorio de *builds* candidatas *a release*, y otro para las *builds* oficiales. El repositorio de *snapshots* contiene también un contenedor por cada versión para almacenar todos los *snapshots* pertenecientes a esa versión. También se representa en el diagrama el proceso *build release* que gestiona la acción de convertir una *build snapho*t a candidata, y el proceso *promote release candidate*, que gestiona la acción de oficializar una *build* candidata a una *build* oficial.

3.3 EJEMPLO DE UN PROTOTIPO DE PROYECTO SOFTWARE

Para la puesta en marcha de un modelo de sistema de integración continua es necesario tener en primer lugar, el punto de partida de un proyecto software que quiera desarrollarse. En este sentido, valdría con un "Hola mundo", si sobre este se va construyendo el producto inicial en función de sus especificaciones iniciales. En el caso que nos ocupa, para la elaboración de este trabajo partimos de un proyecto software que esté a medio desarrollar, pero lo suficientemente avanzado para que este se pueda compilar, ejecutar pruebas unitarias, ejecutar pruebas funcionales en un entorno dado y, en definitiva, todos aquellos procesos necesarios para llevar a cabo la construcción de la *build*.

Este ejemplo de prototipo de proyecto software consiste en el desarrollo de una librería de clases implementadas en Java para ser usadas por una aplicación, que calcule el salario neto mensual de un trabajador de una empresa dada, en función de su tipo de contrato, horas extras trabajadas y ventas realizadas en el mes. Además de ello, emula el comportamiento del uso de otros componentes software proporcionados por la agencia tributaria, para el cómputo de la retención por el impuesto de personas físicas aplicada al salario bruto mensual, en función del tramo correspondiente.

Este prototipo software está contenido en una librería denominada *es.uned.ci*, y contiene una parte de desarrollo de código fuente y otra parte de desarrollo de *testing*. El desarrollo del código fuente de la librería se compone de 5 clases, cuya clase principal es la clase Nomina, esta hace uso de la clase *Aeate*, y esta a su vez con la clase Retención. Además de ello, se define una clase *NominaException*, que hereda de *Exception*, y un enumerador *TipoTrabajador*.

A continuación, se muestra el diagrama de clases UML de la parte del código fuente de la librería *es.uned.ci.*

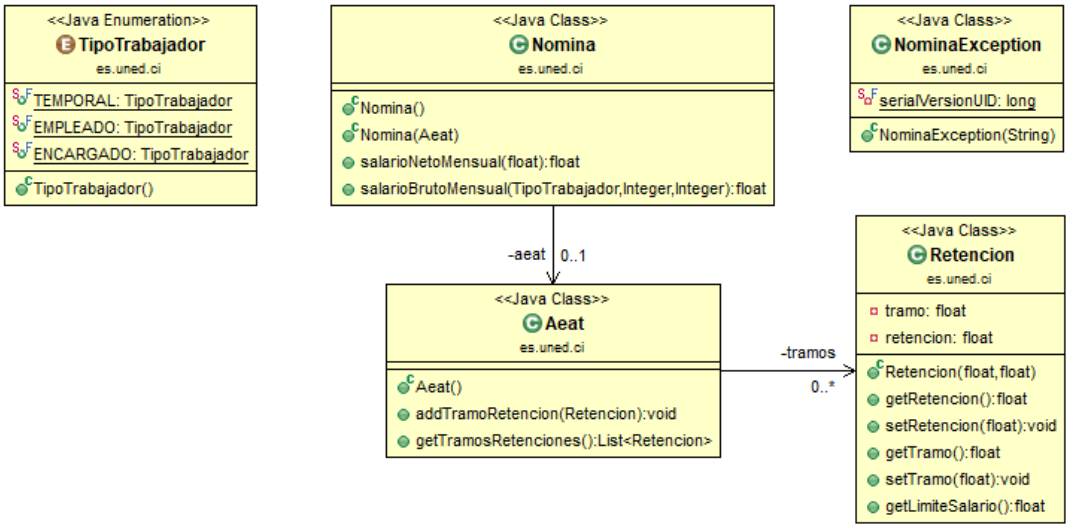


Ilustración 11: Diagrama de Clases UML código es.uned.ci

El desarrollo del código *testing* de la librería se compone de 4 clases, más la clase *NominaException*. Se implementan 3 clases para el desarrollo de los casos de prueba unitarios, haciendo uso de la librería de junit4 [JUnit4 web page, 2019], y una clase para el desarrollo de los casos de prueba funcionales. Las clases que implementan los UT son *NominaTest*, *AeatTest* y *RetencionTest*, cada una de ellas evalúa la clase a la que define. Y la clase que implementa los casos de FT es *runFTest*, que ejecuta el *main* de una supuesta aplicación java.

A continuación se muestra el diagrama de clases UML de la parte del código de *testing* de la librería *es.uned.ci.*

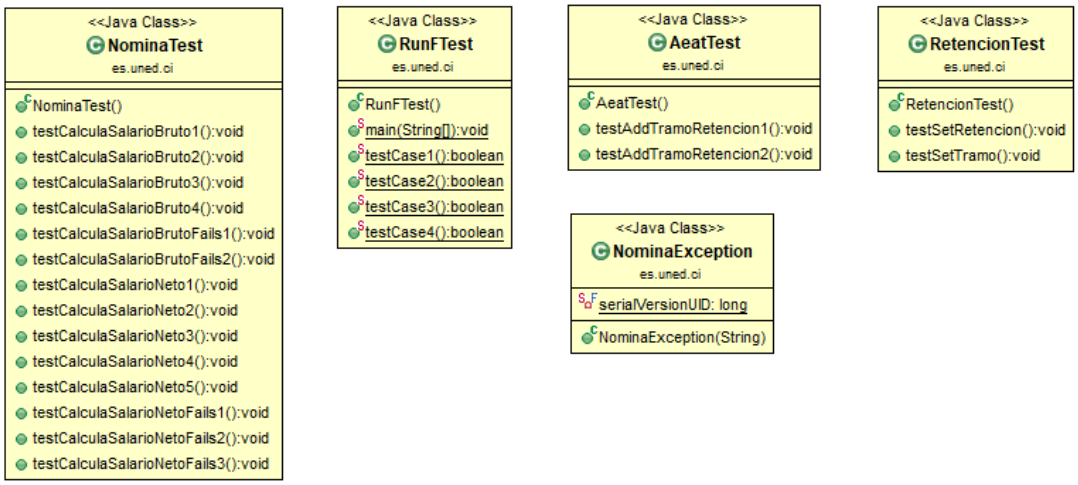


Ilustración 12: Diagrama de Clases UML testing es.uned.ci

El desarrollo total de la librería es.*uned*.ci, tanto de la parte de código fuente como de la parte de *testing*, consta de un total de 9 clases, de las cuales 4 pertenecerían a la parte de *testing*, y 5 a la parte de código fuente. El diseño de la librería *es.uned.ci* quedaría, por tanto, como se muestra en el siguiente diagrama de clases UML.

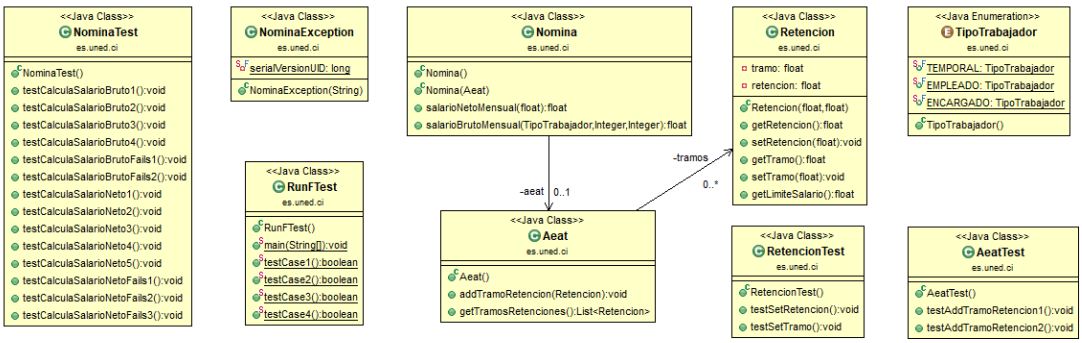


Ilustración 13: Diagrama de Clases UML es.uned.ci

En la siguiente sección se enumerarán todas y cada una de las pruebas desarrolladas, así como la cobertura que ofrecen, para cada uno de los niveles de pruebas definidos.

3.3.1 COBERTURA DE PRUEBAS

De la manera en que está implementado este desarrollo y partiendo de la premisa inicial de que es un desarrollo incompleto, nos es muy útil para acometer el objetivo principal de este trabajo ya que es bastante propenso a errores. Lo importante en este sentido, más del hecho de que esté bien implementado, es que tenga una buena cobertura de pruebas, con el objetivo de que se detecten todos los posibles errores con la ejecución de pruebas unitarias y funcionales.

En el proceso de aseguramiento de la calidad software, una buena cobertura de pruebas viene dada por un buen desarrollo de casos de prueba en todos los niveles de pruebas; estáticas, unitarias, de integración, funcionales y no funcionales y de aceptación. Para nuestro objetivo, nos vamos a centrar en las pruebas unitarias y en las prubas funcionales para el nivel de integración, sin descartar por ello la importancia de los demás niveles de pruebas.

En este sentido, hay que destacar que las pruebas unitarias se caracterizan por evaluar un método de una clase en concreto, y se aíslan del comportamiento global del sistema y de cómo este interacciona con otros sistemas o con el usuario final. Estas pruebas suelen estar ligadas al desarrollo del producto en sí y forman parte de la construcción de la *build*. Las pruebas de integración, en cambio, sí evalúan el comportamiento global del sistema para un entorno definido, en unas condiciones dadas, evaluando una salida para una o varias entradas. Teniendo en cuenta esto, enumeraremos cada uno los casos de pruebas definidos para cada *scope* de pruebas, tanto unitarias como funcionales.

A continuación, se muestra una tabla que representa el conjunto de todos los casos de prueba pertenecientes al *scope* de pruebas unitarias, en total, unos 18 casos. También se representa toda la información asociada a cada caso de prueba, como la clase y el método que evalúa o la entrada y salida esperada para dar veredicto al caso como *pass* o *fail,* y su requerimiento.

TEST CASE	REQ	CLASE EVALUADA	METODO EVALUADO	ENTRADA	SALIDA
UT001	si	Nomina	salarioBrutoMensual	TEMPORAL, 0, 0	800.0
UT002	si	Nomina	salarioBrutoMensual	EMPLEADO, 1000, 5	1200.0
UT003	si	Nomina	salarioBrutoMensual	EMPLEADO, 1500, 1	1220.0
UT004	si	Nomina	salarioBrutoMensual	ENCARGADO, 2000, 50	2700.0
UT005	no	Nomina	salarioBrutoMensual	ENCARGADO, -1, 0	NominaException
UT006	no	Nomina	salarioBrutoMensual	EMPLEADO, 0, -1	NominaException
UT007	si	Nomina	salarioNetoMensual	800.0	720.0
UT008	si	Nomina	salarioNetoMensual	1000.0	880.0
UT009	si	Nomina	salarioNetoMensual	1500.0	1275.0
UT010	si	Nomina	salarioNetoMensual	2000.0	1640.0
UT011	si	Nomina	salarioNetoMensual	2499.0	2049.18
UT012	no	Nomina	salarioNetoMensual	-1	NominaException
UT013	no	Nomina	salarioNetoMensual	799.0	NominaException
UT014	no	Nomina	salarioNetoMensual	5000.0	NominaException
UT015	no	Aeat	addTramoRetencion	(1.0f, -1.0f)	NominaException
UT016	no	Aeat	addTramoRetencion	(-1.0f, 1.0f)	NominaException
UT017	no	Retencion	setRetencion	-1	NominaException
UT018	no	Retencion	setTramo	-1	NominaException

Tabla 1: Scope de casos de prueba UT

Con los casos de pruebas unitarios definidos anteriormente y tras un análisis estático mediante la herramienta de análisis de código proporcionado por Eclipse, se ofrece la siguiente cobertura sobre todos los elementos de código y *testing*, ofreciendo un total de un 86% de cobertura de código.

ELEMENT	COVERTED INSTRUCTIONS	MISSED INSTRUCTIONS	TOTAL INSTRUCTIONS	COVERAGE
Aeat.java	56	26	82	68,30%
Nomina.java	134	16	150	89,30%
NominaException.java	3	0	3	100,00%
Retencion.java	65	2	67	97,00%
TipoTrabajador.java	34	0	34	100,00%
es.uned.ci	292	44	336	86,90%

Tabla 2: Cobertura de las pruebas UT.

Para el desarrollo de pruebas de integración, a continuación, se muestra una tabla que representa el conjunto de todos los casos de prueba pertenecientes al scope de pruebas funcionales, en total, unos 4 casos. Así como toda la información asociada a cada caso de prueba; su descripción, la entrada y salida esperada para dar veredicto al caso como *pass* o *fail,* y su requerimiento.

TEST CASE	DESCRIPCION	REQ	ENTRADA	SALIDA
FT 01	Evaulate Nomina results for TEMPORAL employee	SI	salarioBrutoMensual(TipoTrabajador.TEMPORAL, 100, 9) && salarioNetoMensual(salarioBruto)	salarioNeto = 882.0
FT 02	Evaulate Nomina result for EMPLEADO employee	SI	salarioBrutoMensual(TipoTrabajador.EMPLEADO, 100, 10) && salarioNetoMensual(salarioBruto)	salarioNeto = 1056.0
FT 03	Evaulate Nomina result for ENCARGADO employee	SI	salarioBrutoMensual(TipoTrabajador.ENCARGADO, 1000, 30) && salarioNetoMensual(salarioBruto)	salarioNeto = 1804.0
FT 04	Set new Tramo and Retencion to evaulate new Nomina result	SI	addTramoRetencion(new Retencion(3000.0f, 0.21f)) && salarioBrutoMensual(TipoTrabajador.ENCARGADO, 200, 50) && salarioNetoMensual(salarioBruto)	salarioNeto = 2133.0

Tabla 3: Scope de casos de prueba FT.

Con los casos de pruebas funcionales definidos anteriormente y tras un análisis estático de cobertura de todo el código, se ofrece la siguiente cobertura sobre todos los elementos de código, ofreciendo un total de un 78,27% de cobertura.

ELEMENT	COVERTED INSTRUCTIONS	MISSED INSTRUCTIONS	TOTAL INSTRUCTIONS	COVERAGE
Aeat.java	72	10	82	87,80%
Nomina.java	112	38	150	74,70%
NominaException.java	0	3	3	0,00%
Retencion.java	45	22	67	67,20%
TipoTrabajador.java	34	0	34	100,00%
es.uned.ci	263	73	336	78,27%

Tabla 4: Cobertura de las pruebas FT.

3.3.2 REPOSITORIO DEL PROYECTO

Como se comentó en la introducción de este capítulo, el repositorio principal se va a gestionar mediante el sistema de control de versiones GIT. Este repositorio estará centralizado en el servidor de integración continua, y a su vez, distribuido en todos los repositorios locales de cada sistema de desarrollo. La rama principal del repositorio será la rama master y, opcionalmente se permite que los equipos de desarrollo creen ramas adicionales para la realización de sus desarrollos y pruebas en local.

El repositorio contiene toda la información para construir el proyecto, esto es; Tanto el código fuente como el código de pruebas, así como los ficheros de configuración necesarios y la documentación asociada. Por tanto la estructura de directorios quedaría así: src/, que contiene tanto el código fuente, scr/main, como el código del *testing*, src/test, target/; que contiene ficheros compilados temporalmente y, en la base del repositorio, *maven_project*/; los ficheros de configuración tales como *pom.xml*, *Jenkinsfile* o *build.release*, usados para la configuración y la construcción de la build en sus diferentes etapas, como se verá a continuación.

A continuación, se muestra el árbol de directorios del prototipo de proyecto software con Maven.

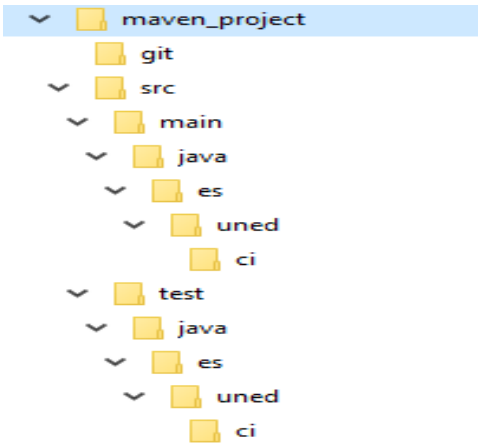


Ilustración 14: Árbol de directorios del prototipo de proyecto software.

3.3.3 COMPILACIÓN Y EJECUCIÓN DE LAS PRUEBAS

El proceso de compilación y ejecución de pruebas unitarias, así como la generación de artefactos, son llevadas a cabo mediante la herramienta de construcción y gestión de proyectos Maven. Esto facilita este proceso dado que toda la configuración necesaria para llevar a cabo la compilación, ejecución de las pruebas unitarias con JUnit y la gestión de los artefactos generados, está contenida en un fichero de configuración, en el mismo repositorio, denominado pom.xml [Maven web page, 2019]. Por tanto, se configura este proceso de manera que se compile los artefactos para poder ejecutar posteriormente los 18 casos de pruebas unitarias, y en caso de éxito, almacenar los *snapshot* de artefactos generados en el repositorio de artefactos, que en este caso estará ubicado en /opt/artifacts/maven_project/snapshots/.

3.4 SISTEMA DE INTEGRACIÓN CONTINUA CON JENKINS

Jenkins es una herramienta de automatización de procesos, especialmente diseñada para integración continua, que nos va a permitir llevar a cabo todos los procesos necesarios de compilación, ejecución de pruebas, gestión de artefactos y, en definitiva, todo lo que necesitemos en nuestro proyecto para evaluar el proceso de *release* del producto software [Jenkins web page, 2019].

Jenkins proporciona una interfaz web en la que, dependiendo de las credenciales del usuario, da acceso a un conjunto de tareas o Jobs, definidos y configurados previamente, organizados por vistas en función de su clasificación. Estas tareas se pueden ejecutar manualmente, pero también se pueden programar para que se lancen periódicamente o tras la ejecución de ciertos eventos previamente definidos. De entre todas las tareas posibles y configurables que se pueden desarrollar en Jenkins, las que nos interesan son aquellas que están relacionadas con los procesos de verificación, validación e integración del software. Es decir, vigilar los cambios en el repositorio, efectuar procesos de compilación, ejecución de pruebas, procesos de *nursing*, generación de artefactos, oficialización de *releases,* etc. Todos ellos, serán comentados en detalle a continuación.

Los tipos de tareas predefinidas que se pueden configurar en Jenkins son muy variados, pero básicamente los que nos interesan son [Jenkins Wiki, 2019]:

- ✓ Proyectos Maven, para la compilación y la ejecución de pruebas de un proyecto JAVA a partir de un fichero de configuración POM.xml.

- ✓ Tareas *pipeline*, para ejecución de varios agentes de compilación y uno o varios procesos de pruebas, incluyendo generación de artefactos y documentación.

- ✓ Tareas *pipeline* multi-rama que, a diferencia de las anteriores, para un mismo repositorio considera diferentes ramas para llevar a cabo sus procesos.

- ✓ Proyectos de estilo libre; es el más usado y configurable en Jenkins, nos permite configurar todas aquellas tareas que se excedan o no encajen con las tareas predefinidas.

- ✓ Además de estas, existen otras como tareas externas, proyectos de multi-configuración, tareas de organización de repositorios, o generador de tareas, entre otros. Estos exceden del propósito de este trabajo.

3.4.1 PROCESO DE VERIFICACIÓN Y VALIDACIÓN

El proceso de verificación y validación se lleva a cabo mediante diferentes *jobs* configurados a partir de tareas de tipo *pipeline*, o bien mediante *jobs* genéricos o de estilo libre. En cualquier caso, el objetivo de cada tarea es la de llevar a cabo la ejecución de casos de prueba definidos para cada nivel de prueba. En nuestro caso corresponde con pruebas estáticas, pruebas unitarias y pruebas funcionales.

Todas ellas están configuradas para ejecutar el contenido del proyecto en un momento dado. Es decir, el HEAD[18] de la rama máster del repositorio principal. Aunque también admite parametrización para poder ejecutar otra rama, por ejemplo, una rama de desarrollo de un *team*, o un *commit* concreto en lugar del HEAD.

Cualquier proceso de ejecución de pruebas, de cualquier nivel, lleva implícito previamente el proceso de compilación y generación de artefactos, a excepción de algunos análisis estáticos, aunque adicionalmente se le puede añadir a dichas tareas parámetros para que ejecuten dicho *scope* de pruebas sobre un artefacto ya compilado, bien sea oficial o un simple *snapshot*.

A continuación, se muestra una vista denominada QA, con las diferentes tareas asociadas a cada nivel de pruebas, en la que se puede apreciar una tarea para el análisis estático de código, otra para el nivel de *Unit Test* y otra para el nivel de *Function Test*.

[18] Se refiere al último cambio en esa rama del repositorio central.

Ilustración 15: Jenkins. Tareas de la vista QA.

ANÁLISIS ESTÁTICO DE CÓDIGO

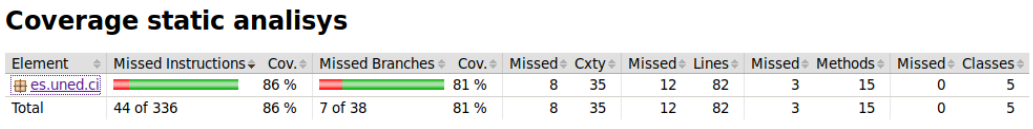
El análisis estático de código, no requiere la ejecución del código fuente, ni siquiera, la generación de artefactos ejecutables ni la compilación del código, aunque, dependiendo del análisis en cuestión, es necesario una pre compilación que asegure que no haya errores de sintaxis. Por tanto, esta fase de análisis suele ser mucho más rápida y directa que el resto de las fases de prueba, es decir, pruebas unitarias, de componente, funcionales y, por supuesto, fase de pruebas de sistema o de aceptación.

El análisis estático de código suele incluir verificaciones de estilo y de diseño, también análisis de seguridad en cumplimento con ciertos criterios o estándares, así como diferentes métricas aplicadas al código para determinar su calidad. Pero lo interesante para este trabajo es el análisis de cobertura del código para determinar la calidad del software, en cuanto al nivel de pruebas se refiere. En este sentido, para determinar el nivel de cobertura que ofrecen los test sobre el código, se hace uso de la herramienta Jacoco. Jacoco es una conocida herramienta *open source*, que evalúa informes del nivel de cobertura de código JAVA, especialmente útil para evaluar test unitarios con JUnit [JaCoCo web page, 2019]. Además, esta se integra perfectamente con proyectos Maven, configurados a través de un fichero de configuración pom.xml, y con Jenkins, a través de diversos *plugins*, aunque esto realmente no es necesario.

La tarea que se ha desarrollado para la elaboración de informes del análisis de cobertura de código es QA_Coverage_Staging, como se puede ver en la ilustración 9. Es una tarea de tipo Maven con un fichero de configuración dedicado denominado jacoco.xml, para diferenciarlo del de la construcción de la *build*, pom.xml.

En él se evalúa la compilación, los test unitarios y la generación de informes de cobertura de código. Si alguno de estos falla, falla esta tarea reportando el error en cuestión. Cabe destacar en este sentido que se puede configurar esta tarea para que evalúe un mínimo de cobertura aceptable y, en caso contrario, reporte la tarea como fallida.

A continuación, se muestra el reporte generado por esta herramienta, indicando la cobertura del código de nuestro prototipo de proyecto software, en función de los test unitarios desarrollados. En el índice de este reporte, *index.htlml,* se puede apreciar que la cobertura total es del 86%. También nos indica la cantidad de instrucciones no cubiertas del total, así como las *missed branches*, entre otros parámetros.

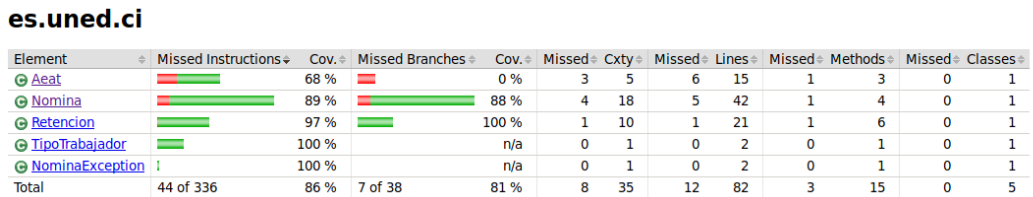


Coverage static analisys

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
es.uned.ci		86 %		81 %	8	35	12	82	3	15	0	5
Total	44 of 336	86 %	7 of 38	81 %	8	35	12	82	3	15	0	5

Ilustración 16: Jacoco. Análisis de cobertura de código.

Si navegamos por el index.html, podemos desplegar el porcentaje de cobertura de cada uno de los elementos de código del que se compone el paquete es.uned.ci. Se puede visualizar, por ejemplo, como la clase Nomina tiene una cobertura en instrucciones del 89%, y del 88% en *branches*.



es.uned.ci

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Aeat		68 %		0 %	3	5	6	15	1	3	0	1
Nomina		89 %		88 %	4	18	5	42	1	4	0	1
Retencion		97 %		100 %	1	10	1	21	1	6	0	1
TipoTrabajador		100 %		n/a	0	1	0	2	0	1	0	1
NominaException		100 %		n/a	0	1	0	2	0	1	0	1
Total	44 of 336	86 %	7 of 38	81 %	8	35	12	82	3	15	0	5

Ilustración 17: Jacoco. Análisis de cobertura del paquete es.uned.ci.

Si accedemos al elemento Nomina podemos desplegar el conjunto de elementos de código que lo componen, es decir, los diferentes métodos y constructores que componen la clase, indicando el nivel de cobertura de cada uno de ellos en los diferentes parámetros.

Nomina

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
salarioBrutoMensual(TipoTrabajador, Integer, Integer)		88 %		88 %	2	11	2	24	0	1
Nomina(Aeat)		0 %		n/a	1	1	3	3	1	1
salarioNetoMensual(float)		100 %		87 %	1	5	0	12	0	1
Nomina()		100 %		n/a	0	1	0	3	0	1
Total	16 of 150	89 %	3 of 26	88 %	4	18	5	42	1	4

Ilustración 18: Jacoco. Análisis de cobertura del elemento Nomina.

Por último, podemos acceder al código del elemento Nomina accediendo a cualquiera de los elementos anteriores. En el código se indica en verde las instrucciones cubiertas, en rojo las que no son evaluadas y, en amarillo, aquellas sentencias de control que no evalúan todas sus opciones, es decir, las *branches missed*.

Nomina.java

```
1.  package es.uned.ci;
2.
3.  import java.util.List;
4.
5.  public class Nomina {
6.
7.      private Aeat aeat;
8.
9.      public Nomina() throws NominaException {
10.         aeat = new Aeat();
11.         }
12.
13.     public Nomina(Aeat aeat) throws NominaException {
14.         this.aeat=aeat;
15.         }
16.
17.         public float salarioNetoMensual(float salarioBrutoMensual) throws NominaException {
18.
19.             if(salarioBrutoMensual < 800)
20.                 throw new NominaException("El salario bruto menusal no puede ser menor de 800");
21.
22.             if(salarioBrutoMensual >  4999)
23.                 throw new NominaException("El salario bruto menusal no puede ser mayor de 5000");
24.
25.             float retencion = 0.0f;
26.
27.             List<Retencion> tramos = aeat.getTramosRetenciones();
28.             for(Retencion tr: tramos) {
```

Ilustración 19: Jacoco. Análisis de cobertura. Desglose de código.

PRUEBAS UNITARIAS

La ejecución de pruebas definidas para el nivel de *unit test*, se puede configurar mediante una tarea del tipo pipeline que ejecute secuencialmente 4 *stages*. El primer *stage* realiza un clon del repositorio en la rama master, si no se le indica lo contrario, para preparar el *workspace*; el segundo ejecuta una serie de comprobaciones del entorno para asegurar los siguientes *stages*; el tercero efectúa la compilación del código fuente, almacenando los artefactos en un directorio target temporal para el siguiente y ultimo *stage*, que supone la ejecución del *scope* de pruebas unitarias. La configuración de esta ejecución se realiza mediante un script en lenguaje *groovy*, contenido en el repositorio principal, al que hemos llamado *Jenkinsfile_UT_staging*.

Como se puede observar en la siguiente ilustración, los diferentes *stages* se ejecutan de manera secuencial por un solo ejecutor. En otros casos se podría configurar esta tarea para que ejecute cada *stage*, siempre que no haya dependencias entre ellas como en nuestro caso, en paralelo y por diferentes ejecutores para reducir el tiempo. En este caso, como se puede apreciar, el promedio de cada *stage* es de 5 segundos para el clonado del repositorio, 2 para la comprobación del entorno, 4 para la compilación, y menos de un segundo para la ejecución de las 18 pruebas unitarias. Si se completan todos ellos con éxito se considera la ejecución válida, en caso contrario será fallida, no realizando ninguna acción posterior salvo la de notificar del error.

Ilustración 20: Jenkins. Pipeline de la tarea QA_UT_staging.

PRUEBAS FUNCIONALES

La ejecución de pruebas definidas para el nivel de *function test*, también se lleva a cabo mediante una tarea de tipo *pipeline*. En este caso, los 3 primeros *stages*, es decir, el de clonado del repositorio, la comprobación del entorno, y el proceso de compilación y generación de artefactos, son idénticos a la tarea anterior de pruebas unitarias. El *stage* que cambia en este caso es el de ejecución de pruebas funcionales.

Como se comentó en la sección anterior, la cobertura de pruebas funcionales, véase tabla 4, el *scope* definido para validación a nivel funcional se compone de una *test suite*[19] compuesta por 4 pruebas funcionales. Y, a diferencia del nivel de *unit test* donde la ejecución de las 18 pruebas unitarias con JUnit no exceden de un segundo, en el caso de las pruebas funcionales estas tienen un promedio de 20 segundos. Tiene sentido ya que, aunque solo son 4 casos, se evalúa una precondición y una post condición, antes y después de la ejecución de las pruebas, lo que supone preparar el entorno, lanzar todas las pruebas con el sistema en ejecución, y dejar el entorno como estaba antes para no influir en las siguientes ejecuciones. Por lo que le lleva al sistema mucho más tiempo que ejecutar pruebas unitarias o de componente, como se puede ver en la siguiente ilustración.

Ilustración 21: Jenkins. Pipeline de la tarea QA_FT_staging.

[19] Agrupación de casos de prueba.

3.4.2 PROCESO DE CONSTRUCCIÓN DE LA *BUILD*

El proceso de construcción de la *build* consiste en construir aquellos artefactos generados para un entorno definido, llevando a cabo tanto los procesos de compilación y comprobación de los targets necesarios, como la ejecución de los diferentes *scopes* de pruebas que se hayan determinado para cada nivel de prueba. Esto quiere decir que, dependiendo de la oficialidad de la *build* y de las necesidades del entorno del target en cuestión, se desarrollará un plan de pruebas u otro, con un *scope* de pruebas más o menos amplio y más o menos pesado, dependiendo del tiempo del que se disponga para su evaluación y de la importancia de dicha *build*.

Lo más común que suele hacerse en los departamentos de *release* para llevar a cabo esta tarea, es definir un *scope* de pruebas más liguero para determinar si es válido cada cambio que se haya introducido en el repositorio, conocido como proceso de verificación [Continuous Delivery Foundation, CDF 2019], en el que se incluyen pruebas de *unit test*, y definir otro *scope* de pruebas más pesado que incluya la ejecución de pruebas funcionales de integración, además de las unitarias, para asegurar la calidad de la *build* del conjunto de cambios en el repositorio previamente verificados. Este proceso de oficialización de la *build* se le conoce como validación, [Continuous Delivery Foundation, 2019]**,** y ha de realizarse frecuentemente, o al menos una vez al día, siempre y cuando haya cambios en el repositorio que validar, ya que no tiene sentido validar el mismo contenido del repositorio varias veces, a no ser que se quiera evaluar inestabilidades. Cabe destacar que, aunque este proceso de validación se realice generalmente de manera automática en intervalos de tiempo definidos, ha de poder configurarse para que se lance según las necesidades del proyecto tantas veces como se necesite, a modo de evaluar el conjunto de cambios necesarios para una determinada *feature* en la que esté trabajando el equipo de desarrollo.

Para el proceso de construcción de la *build* de nuestro proyecto de integración continua con Jenkins se ha desarrollado 2 tareas, clasificadas en una vista denominada *Build*. Una tarea de tipo genérica para la construcción de la *build* por cada *commit*, llamada *Build_commit*, que realice la construcción de una *build* cada vez que se introduzca un cambio en el repositorio, atendiendo al proceso de verificación, y otra tarea de tipo *multibranch pipeline* denominada

Build_release, que realice la construcción de la *build* cada cierto tiempo definido, atendiendo al proceso de validación.

Ilustración 22: Jenkins. Tareas de la vista BUILD

A continuación, se detalla cada uno de los procesos definidos para la construcción de la *build*, construcción de la *build* de cada cambio y construcción de la *build* de *release* cada cierto intervalo de tiempo o manual.

CONSTRUCCIÓN DE CADA CAMBIO

El proceso de construcción de cada cambio, denominado *Build_commit*, ha de realizar el proceso de verificación de cada *commit* en el repositorio lo más rápido posible, con el objetivo de descartar o notificar al propietario del *commit,* los fallos introducidos en el repositorio y así este pueda corregirlos inmediatamente antes de seguir avanzando en su trabajo. Este es el primer nivel de evaluación, cuyo objetivo principal es la de no dejar fallos en el repositorio antes de lanzar el proceso de validación, al menos, al nivel de compilación y verificación de test unitarios.

El sistema de evaluación de esta tarea determina tres posibles resultados. Si ha sido capaz de construir la *build* con éxito, es decir, compilar y generar los artefactos y pasan todas las pruebas de nivel de *unit test* de regresión, se considera que el *commit* introducido en el repositorio está verificado y almacena los artefactos generados como *snapshot* en el repositorio de artefactos correspondiente, que se verá a continuación.

En caso de que haya sido capaz de compilar, pero haya fallado algún *unit test* de la ejecución de regresión, se considera la ejecución fallida, notificando al propietario del *commit* del error introducido y el caso o los casos que han fallado con su cambio, almacenando del mismo modo los artefactos como *snapshot* fallido. En caso de que haya un fallo de compilación o generación de artefactos, considera la ejecución errónea, notificando tanto al autor como a los responsables del departamento de *release* de tal evento y procediendo a descartar los cambios mediante el *revert*[20] de dicho *commit*.

La siguiente ilustración muestra una sección del histórico de ejecuciones para la tarea de la construcción de la *build*, donde se aprecia los tres posibles resultados, bola azul en caso de éxito, bola amarilla en caso de fallo, y bola roja en caso de error.

Ilustración 17: Jenkins. Historial de ejecuciones de la tarea Build_commit.

Jenkins también nos proporciona para este tipo de tareas una gráfica que muestra la tendencia de los resultados de las pruebas, para evaluar la estabilidad de las ejecuciones. Indicando en azul los casos de éxito y en rojo los fallidos.

Ilustración 23: Jenkins. Tendencia de los resultados de la tarea Build_commit.

[20] Cambio en el repositorio que supone revertir un cambio incluido previamente.

Si seleccionamos en la anterior gráfica la build 168, que contenia un fallo de verificacion en un unit test, Jenkins nos muestra un resumen del fallo de verificacion, donde se aprecia que ha fallado un caso de JUnit de la Clase NominaTest, en uno de los test para calcular el salario bruto, obteniendo un valor diferente al esperado.

Ilustración 24: Jenkins. Resultado de una ejecución de la tarea Build_commit

CONSTRUCCIÓN DE CADA RELEASE CANDIDATA A OFICIAL

El proceso de construcción de cada *release* se realiza mediante una tarea de tipo *Multi branch* pipeline denominado *Build_release*, de manera que, aunque de momento solo se esté usando para la rama master de un repositorio, esta tarea está preparada para ejecutar futuras construcciones para diferentes ramas en diferentes repositorios. La configuración de la construcción del pipeline está contenida en el fichero de configuración *Jenkinsfile* del repositorio principal, por lo que su configuración está auto contenida dentro del repositorio.

Como se ha mencionado en otras secciones de este trabajo, el fichero *Jenkinsfile* es un fichero de configuración que contiene un script en lenguaje *groovy*, en el que podemos definir los diferentes *stages* del que se compone el pipeline. El proceso que se ha seguido para definir los diferentes *stages* en este fichero se muestra a continuación:

```
#!/usr/bin/env groovy
pipeline {
    agent any
    environment {…}
    stages {
     stage ("Environment") {
        steps {…}}
     stage ("Compile") {
        steps {…}}
     stage ("Unit Tests") {
        steps {…}}
     stage ("Functional Tests") {
        steps {…}}
     stage ("Deploiment stage") {
        steps {…}}}}
```

Donde, en primer lugar, se le indica al sistema que puede usar cualquier agente, es decir, cualquier ejecutor. Posteriormente se define una sección para preparar el entorno y establecer el valor de las variables necesarias. Y finalmente, se definen un conjunto de *stages* ejecutados secuencialmente, en el que se definen diferentes *steps*, donde reside la lógica de programación para cada tipo de *stage*, bien sea comprobación del entorno, compilación, ejecución de pruebas o despliegue de artefactos.

A continuación, se muestra el histórico de resultados de la ejecución de esta tarea que proporciona Jenkins mediante el *stage view*, indicando, por un lado, el historial de tareas con el resultado de su ejecución, bola azul o roja, y por otro, el despliegue de los *stages* de cada ejecución. Donde se puede apreciar, por ejemplo, que la construcción 78 contiene 2 *commits*, así como la 76 que contiene 9, mientras que la 77 no contiene ninguno por haber sido ejecutada manualmente.

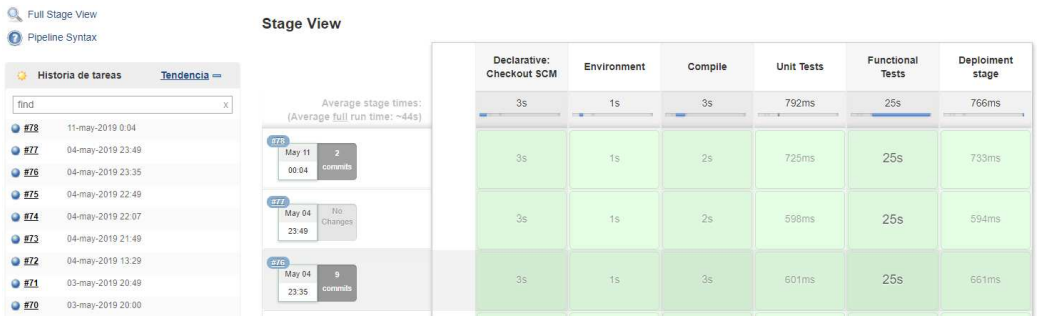


Ilustración 25: Jenkins. Pipeline de la tarea Build_release.

Este proceso de construcción de la *release* es mucho más restrictivo que el de construcción de un *commit*, ya que tiene que asegurar la calidad de una versión candidata a oficial, además de asegurar que la rama master, también conocida como rama LSV (*Last Stable Version*), no contiene ningún tipo de errores ni fallos de verificación ni validación. En este sentido solo se admite 2 resultados; construcción válida indicando el resultado con bola azul, y construcción fallida, indicando el resultado con bola roja.

Las acciones llevadas a cabo si la construcción de la *build* es fallida, bien sea por un fallo de compilación, generación de artefactos o de *testing*, vas más allá de revertir el *commit* del head de la rama, puesto que, estamos evaluando el conjunto de *commit* existentes desde la última ejecución de esta tarea y, cualquiera de ellos puede ser el causante del fallo. En este caso la acción llevada a cabo si la construcción tiene algún fallo, bola roja, es la de bloquear la rama para que nadie pueda entregar nada al repositorio y notificar al departamento de *release* para que procedan a su investigación. El desbloqueo se hará manualmente tras la comprobación en local del *commit* con el *fix* o el *revert* de los *commit* causantes del fallo.

Las acciones llevadas a cabo si la construcción de la *build* es válida, además de indicar el resultado en el histórico de tareas como construcción válida y el correspondiente desglose de los *stages* que componen el pipeline, son las siguientes:

- ✓ Modifica la versión de la *release*, aumentando en 1 el indicador de revisión. Es decir, pasaría de ser, por ejemplo, de la 1.0.1 a la 1.0.2.
- ✓ Modifica los ficheros de configuración de la versión de *release*, es decir, *build*.*release* y *build*.*version* para que estos apunten a la nueva versión.
- ✓ Genera un nuevo *commit* con estos cambios y con el mensaje, siguiendo el ejemplo anterior, "REVISION UPDATE: Release 1.0.2".
- ✓ Genera un nuevo repositorio de artefactos para almacenar los *snapshots* que se generen en la siguiente revisión. Para nuestro ejemplo seria; *snapshots*/1.0.2/.

- ✓ Almacena el artefacto generado en el repositorio de artefactos candidatos a *release*, con el identificador RELEASE_<*timestamp*>-versión. Por ejemplo, RELEASE_1556746540-1.0.1.jar.

Para finalizar, indicar que periodo de ejecución de esta tarea es totalmente configurable, y ha de establecer un valor u otro dependiendo de la estimación de carga de cambios en el repositorio. De manera que, en épocas de estimación de poca carga se establezca un periodo de ejecución de entre 3 y 4 horas y, en épocas de mucha carga, se establezca un periodo de una hora. De esta manera nos aseguraremos que entre *releases* no haya más de 10~15 cambios a evaluar en caso de fallo. Destacar, como se ha comentado anteriormente, que esta tarea puede ser bloqueada a la espera de que entren los cambios necesarios para completar una *feature* de programa, a la vez que se pueda ejecutar manualmente para iniciar este proceso una vez estén dichos cambios verificados en el repositorio.

3.4.3 PROCESO DE RELEASE

El proceso de *release* depende tanto de la organización como de las necesidades y características del proyecto. En este trabajo se intenta que este sea lo más genérico posible, adaptado a las circunstancias de este modelo planteado.

El control de versiones del producto establece una notación numérica compuesta por los siguientes valores:

- ✓ Versión o mayor. Indica la principal versión del producto software, constituye un conjunto de funcionalidades concretas que son cubiertas.
- ✓ Subversión o menor. Cubre un menor conjunto de funcionalidades, siendo un subconjunto del anterior.
- ✓ Revisión. La revisión del producto indica, en nuestro caso, el seguimiento diario del producto, es decir, una manera de medir el avance del proyecto de un producto estable previo a la oficialización.

- ✓ Entrega. En este caso no se ha tenido en cuenta por cuestiones prácticas, pero en ocasiones, es necesario añadir esta notación si se quiere tener en cuenta el número de veces que una entrega se rechaza. Normalmente tiene una notación alfanumérica.

De esta manera la notación correspondiente a una versión del producto software concreta quedaría definida por versión, subversión, revisión y, opcionalmente, entrega. Y su notación quedaría de la siguiente manera:

Version.Subversion.Revision[Entrega]

En el caso que nos ocupa, esta notación iría desde la 1.0.0 hasta la N.M.I, siendo estos números naturales y sin considerar, en principio, un valor máximo, aunque se podría tener en cuenta. En el caso de N, se establece un valor mayor que 0, puesto que no tendría sentido la notación 0.0.0.

El control de versionado del producto, se mantiene mediante ficheros de configuración en el repositorio principal. De esta manera el repositorio es auto contenido en cuanto a la configuración de la versión software se refiere. Estos ficheros de configuración son; *build.version* y *build.release*. Ambos llevan la anotación del producto en diferentes formatos que usaran los *jobs* de Jenkins en sus configuraciones.

En cuanto al proceso de *release* de nuestro proyecto Jenkins, este se lleva a cabo mediante diferentes tareas o *jobs* clasificados dentro de la vista de RELEASE. En el que tenemos una tarea denominada INITIAL_RELEASE que por motivos de seguridad esta desactivada. Esta tarea establece los valores de configuraron por defecto para iniciar el computo de las notaciones de la versión. Además, se definen otras 3 tareas, RELEASE_VERSION, RELEASE_SUBVERSION y RELEASE_REVISION, que se comentarán a continuación. Estas tareas modifican la versión del producto y la notación en cuestión, además de la configuración del repositorio de artefactos según aplique. Y, por último, la tarea PROMOTE_RELEASE_CANDIDATE, que también comentaremos a continuación, cuyo objetivo es la de oficializar una candidata a estable a una versión *release* oficial.

Ilustración 26: Jenkins. Tareas de la vista RELEASE

Los artefactos, tanto de las *releases* oficiales, como las candidatas, así como los *snaphots* de cada *commit*, se guardarán en el servidor de repositorios de artefactos, cada uno en sus correspondientes directorios, como se mostrará a continuación, siendo cada uno de ellos totalmente accesible por cualquier miembro de la organización para su evaluación y comparación.

Ilustración 27: Repositorio principal de artefactos.

La configuración de los artefactos queda de la siguiente manera, al igual que se indica en la ilustración 21:

- ✓ Un directorio de *snapshots* que contiene un directorio por cada versión del producto, ilustración 22. Donde se almacenan todos las *builds snapshots* de cada *commit* perteneciente a esa versión, ilustración 23.
- ✓ Un directorio *release_candidat*e donde se almacenan todas las versiones candidatas a *release* o versión oficial. Se podría entender a estas versiones como versiones beta, que aún no han sido oficializadas, ilustración 24.
- ✓ Un directorio de *releases* oficiales, donde se almacenan las versiones oficiales del producto, ilustración 26.

Ilustración 28: Repositorio de artefactos snapshots.

Como se comentó en la anterior sección, el proceso de construcción de cada cambio o *commit* en el repositorio, siempre que este sea válido o tenga solo errores de *unit test*, genera una *builld spnapshot* que se almacena en el repositorio de artefactos de *snapshots* de la versión a la que pertenece, de manera que esta pueda volver a ser evaluada por el sistema si así procede. Para poder identificar inequívocamente cada artefacto, este se almacena con el *commit* id, seguido de la versión de la *build* enumerada por Jenkins.

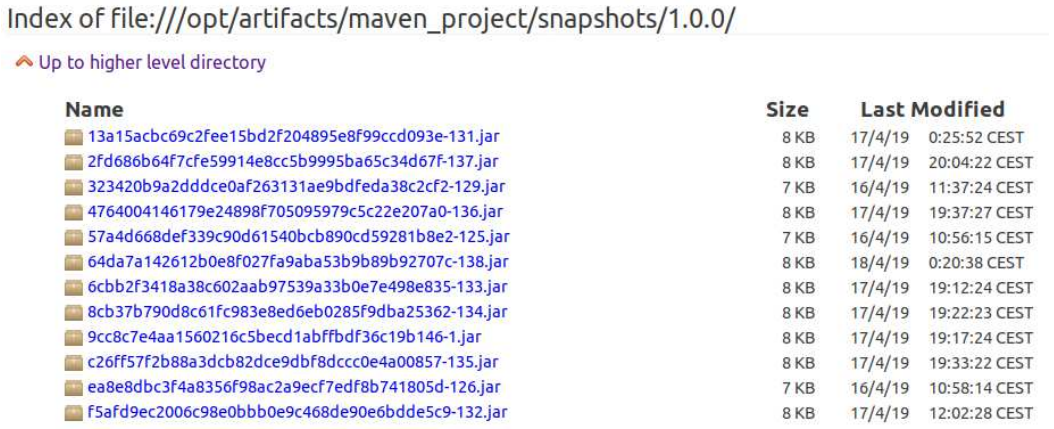


Ilustración 29: Repositorio de artefactos snapshots versión 1.0.0.

En cuanto al proceso de construcción de *release*, comentado en la sección anterior, solo en el caso de que sea válido, bola azul, se genera una *build* candidata a *release*, almacenándola en el repositorio de *release candidate*, además de ejecutar la tarea *de release revision*, como se verá a continuación.

Dado que, se admite que pueda ejecutarse manualmente el proceso de construcción de *release*, el artefacto generado no solo contiene la versión del producto sino también el *timestamp*, de esta manera queda identificado inequívocamente.

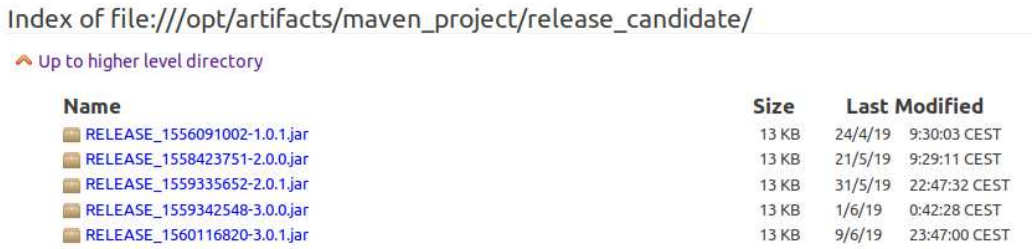


Ilustración 30: Repositorio de artefactos candidatas a release.

RELEASE REVISIÓN

El proceso de *release* de la revisión del producto se lleva a cabo mediante la tarea RELEASE_REVISION. Esta tarea se ejecuta siempre que se ejecute con éxito el proceso de construcción de la *release*, aunque también se puede forzar su ejecución manualmente. Las acciones llevadas a cabo por esta tarea son la de aumentar en uno el valor de la revisión del producto y generar un nuevo repositorio de artefactos para los *snapshots* que se generen pertenecientes a la versión con la nueva revisión.

RELEASE SUBVERSIÓN

El proceso de *release* de la subversión del producto se lleva a cabo mediante la tarea RELEASE_SUBVERSION. Las acciones llevadas a cabo por esta tarea son la de aumentar en uno el valor de la subversión del producto y generar un nuevo repositorio de artefactos para los *snapshots* que se generen pertenecientes a la versión con la nueva subversión. Esta tarea se puede programar para que se ejecute periódicamente, por ejemplo, una vez a la semana, aunque es conveniente que el cambio se realice de forma manual por el personal de *release*.

RELEASE VERSIÓN

El proceso de *release* de la versión del producto se lleva a cabo mediante la tarea RELEASE_VERSION. Las acciones llevadas a cabo por esta tarea son la de aumentar en uno el valor de la versión del producto y generar un nuevo repositorio de artefactos para los *snapshots* que se generen pertenecientes a la nueva versión. Esta tarea se puede programar para que se ejecute periódicamente, por ejemplo, una vez al mes, aunque es conveniente que el cambio se realice de forma manual por el personal de *release*.

PROMOTE RELEASE CANDIDATE

El proceso de promocionar una versión candidata a *release* a una versión *release* oficial, se lleva a cabo mediante la tarea PROMOTE RELEASE CANDIDATE. Esta tarea se va a realizar siempre de forma manual mediante algún miembro del departamento de *release*. Para llevar a cabo la ejecución de dicha tarea es necesario indicarle al *job* como parámetro la versión de la *build* que se quiere oficializar, tal y como se muestra en la siguiente ilustración.

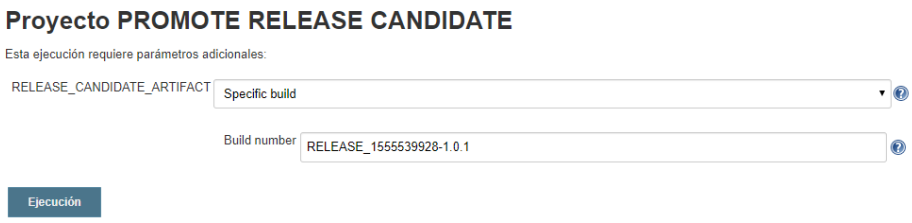


Ilustración 31: Jenkins. Tarea PROMOTE RELEASE CANDIDATE

De esta manera, la *build* en cuestión, en este caso la *buid* candidata a *release* RELEASE_1555539928-1.0.1, pasaría de estar en el repositorio de artefactos de *release_candidate*, a *build* OFFICIAL_RELEASE-1.0.1, en el repositorio de artefactos de *releases*. Como se puede apreciar en las dos siguientes ilustraciones.

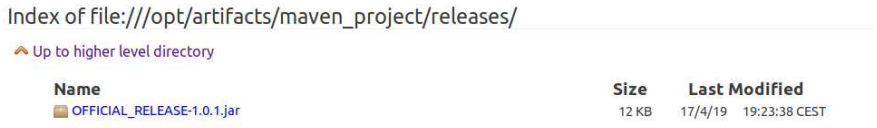


Ilustración 32: Repositorio de artefactos releases oficiales.

3.5 CONSIDERACIONES RESPECTO AL MARCO TEÓRICO

Como se mencionó en la introducción de este capítulo 3, implementación de un sistema de integración continua, se va a proceder a evaluar este modelo, una vez implementado, en consideración con las recomendaciones del autor Martin Fowler, en su artículo *Continuous Integration* [Fowler, 2006].

En el capítulo 2 de este documento, "Marco teórico de integración continua", en la sección 2.2, "Prácticas basadas en integración continua", se definen las 11 buenas practicas que debe cumplir todo sistema de integración continua. Estas serán comentadas, una a una, para comprobar si el modelo implementado cumple con las recomendaciones definidas en el marco teórico.

1. "MANTENER UN ÚNICO REPOSITORIO DE CÓDIGO"

En el modelo planteado se define un único repositorio, tanto para el código fuente como para el código de *testing*, además de los ficheros de configuración. Se configura la gestión de repositorios mediante la herramienta GIT, definiendo el repositorio principal como "repo/git/maven_project.git", en el que se configura la rama principal o máster, también llamada LSV, que contendrá todos los cambios del proyecto. Además, se permite a los equipos de desarrollo que puedan crear ramas de desarrollo, para su uso previo a las integraciones.

2. "AUTOMATIZAR LA CONSTRUCCIÓN"

Se han definido 2 tareas para automatizar la construcción. Una para la construcción de una *build*, llamada intermedia o *snapshot*, por cada cambio que se realice en el repositorio, y otra para la construcción de una *build*, llamada candidata a oficial, que puede ser realizada manualmente o programada cada cierto intervalo de tiempo, dependiendo de la carga de trabajo temporal del equipo de desarrollo. Además de ello, se pueden establecer tanto construcciones automáticas para el cambio de la versión del producto, como construcciones manuales bajo demanda, y lo mismo para el resto de las construcciones.

3. "UTILIZAR PRUEBAS AUTOMATIZADAS"

Las pruebas automatizadas se gestionan mediante la construcción de la *build*. Además de ello, hay disponible una serie de tareas bajo demanda que realizan el proceso de *staging*[21] del nivel de pruebas seleccionado, tanto de una *build* ya compilada como de un cambio o *commit* concreto en el repositorio, a modo de evaluar el aseguramiento de la calidad en el sistema de integración.

4. "ACTUALIZAR EL REPOSITORIO A DIARIO"

Esta recomendación solo puede implantarse en la organización a modo de "buenas practicas", para los equipos de desarrollo. En el modelo planteado no se establece ningún control que obligue a los equipos a que actualicen el repositorio local diariamente. En este sentido lo único que puede hacer el sistema de integración continua es intentar no mantener el repositorio bloqueado por mucho tiempo. Esto se consigue mediante procesos de ágiles y rápidos de respuesta ante fallos en el repositorio, para poder desbloquear LSV lo antes posible y que los equipos puedan seguir integrando.

5. "CADA CAMBIO SE CONSTRUYE EN LA MÁQUINA DE INTEGRACIÓN"

Todos los cambios que se integren en el repositorio principal, se construyen mediante los dos procesos de integración definidos, en la máquina de integración continua CI Server.

6. "ARREGLAR ERRORES DE LA CONSTRUCCIÓN INMEDIATAMENTE"

El modelo planteado de integración continua establece que, si un cambio en la rama master del repositorio rompe la compilación, este no se integra, pero si falla en algún test unitario, si se integra, notificando del error al dueño del cambio para que sea él el que proceda a corregirlo. De no ser así, el sistema bloqueará la integración en el siguiente proceso de la construcción de la *release* candidata.

[21] Etapa de un proceso del pipeline.

Este proceso, tal y como está planteado, debería mejorarse dado que el sistema debería ser capaz de descartar o revertir automáticamente los cambios que rompan o bloqueen la integración continua. Este punto puede mejorarse mediante procesos de *nursing*, que se comentará en el siguiente capítulo.

7. "CONSEGUIR CONSTRUCCIONES RÁPIDAS"

El sistema planteado realiza una construcción de una *build* por cada *commit*, ejecutando solo la compilación de artefactos y la ejecución de pruebas unitarias. Este proceso le lleva al sistema poco tiempo. Cada cierto intervalo de tiempo definido o bien de manera manual, se ejecuta el proceso de construcción de una *build* candidata a *release*, con una ejecución de pruebas más larga, que contiene tanto pruebas unitarias como pruebas funcionales. Lo que le supone al sistema más tiempo que el proceso anterior, aproximadamente del orden de 10 veces más, pues tiene que asegurar en mayor medida la calidad de la *build* candidata. En cualquier caso, se intenta por todos los medios que la construcción sea lo más rápida posible, cosa que no solo depende del planteamiento sino también de la arquitectura hardware del sistema de integración continua.

8. "PROBAR EN UN CLON DEL ENTORNO DE PRODUCCIÓN"

Teniendo en cuenta los requisitos del prototipo software de este trabajo, las pruebas se realizan en el sistema de integración continua. En un entorno real, lo más común sería tener diferentes artefactos para una misma versión, de manera que cada uno de ellos se ejecutase en el entorno correspondiente, similar al de producción. En el modelo planteado, el sistema de producción es el mismo que el de integración, por lo que solo se genera un artefacto por versión y este se prueba en el mismo sistema de integración continua. Si el sistema de integración continua fuese diferente al de producción, al menos habría que considerar la opción de usar sistemas virtualizados o contenedores, si se quiere usar el mismo servidor de integración continua para las pruebas, o usar servidores diferentes para la realización de estas.

9. "FACILITAR LA OBTENCIÓN DE LA ÚLTIMA VERSIÓN"

De la manera que se ha planteado el repositorio de artefactos del sistema, cualquier persona miembro de la organización, bien sea del equipo de desarrollo

o de *testing*, puede obtener cualquier artefacto de cualquier versión, tanto de una versión oficial, como de una versión candidata o *snapshot*. De manera que pueda realizar instalaciones y ejecuciones de las pruebas que considere.

10."CONOCER EL ESTADO DE LA ULTIMA VERSIÓN"

El estado de la última versión, en cuanto a la calidad, es conocida, tanto si esta es estable como si no, ya que en cualquier momento se puede consultar en el sistema integración continua el resultado de las pruebas pertenecientes a dicha versión. También se puede consultar el contenido de la última versión mediante consulta al repositorio principal. No obstante, dadas las circunstancias académicas del modelo planteado, no se ha implementado ningún sistema de documentación asociada a cada versión del producto. En un sistema real, cada versión tendría asociado unas *release notes*[22] con documentación asociada tanto de las pruebas como del contendido.

11. "AUTOMATIZAR EL DESPLIEGUE"

Esta recomendación no queda cubierta pues, como se comentó en la sección del planteamiento y objetivos del trabajo, la parte correspondiente a despliegue continuo, quedaba fuera del alcance del proyecto. Por lo que solo se ha tenido en cuenta los procesos de integración continua y de entrega continua, y no los de despliegue continuo, ya que no se realiza el despliegue en ningún sistema de producción de manera automática.

Una vez analizado punto por punto, todas las recomendaciones recogidas en el artículo *Continuous Integration* [Fowler, 2006], que no deja de ser más que la visión de un autor, reconocido en esta materia, de las recomendaciones para un sistema de integración continua. Se puede concluir que este modelo implantado como sistema de integración continua cumple con los requisitos establecidos en el marco teórico, a excepción del último, siempre y cuando se tenga en cuenta las condiciones y circunstancias académicas para las que se ha desarrollado, principalmente en relación con las dimensiones del prototipo software planteado.

[22] Comúnmente asociado a la documentación de una versión de un producto software.

3.6 CONCLUSIONES DEL SISTEMA DE INTEGRACIÓN CONTINUA

El modelo de sistema de integración continúa implementado para el prototipo de proyecto software propuesto, cumple, en mayor o menor medida, con las recomendaciones que, según algunos autores expertos en esta materia, ha de satisfacer todo sistema que implemente integración continua. Si bien, el modelo planteado cubre solo las necesidades académicas, en cuanto a dimensiones y necesidades se refiere, sí ha de cubrir teóricamente con las necesidades implícitas de cualquier proyecto software de propósito general, desarrollado con metodologías de integración continua.

Por lo tanto, se concluye que el modelo planteado implementa un sistema de integración continua, al que se denominará en el siguiente capítulo, modelo o sistema inicial de integración continua.

Otra cosa a tener en cuenta en este sistema inicial es que, la dimensión del prototipo software hace posible que, tanto la compilación como la ejecución de las pruebas unitarias, así como las funcionales, se puedan realizar en el mismo sistema de integración, en este caso una máquina virtual, y, además, no le lleve al sistema más de unos 50 segundos en completar todo este proceso. En un sistema de real, de un proyecto software real, este proceso es mucho más costoso, teniendo que redimensionar las ejecuciones en diferentes ejecutores de diferentes entornos y en sistemas hardware más complejos.

Si bien, se puede considerar tanto el proceso de construcción de la *build* para cada cambio, como el proceso de construcción de una *build release*, mucho más costoso en tiempo y recursos en un proyecto real que en el prototipo planteado, se ha intentado en todo momento extrapolar dichos tiempos aproximados manteniendo la relación 1 a 5, es decir, de unos 10 segundos para la ejecución de los casos de UT a unos 50 segundos para la ejecución de los casos de FT, ver ilustración 16. Esto quiere decir que, si a un sistema real le llevase entre 5 y 10 minutos la construcción de una *build* para cada cambio en el repositorio, el proceso de construcción de la *build* candidata a *release*, le debería de llevar al sistema, aproximadamente, entre 30 y 60 minutos. Esto dependerá siempre de la cantidad de casos de prueba que se quiera ejecutar para cada nivel y del sope que se defina para cada proceso, pero nos puede servir como referencia.

Otra consideración a tener en cuenta, como se verá en el siguiente capítulo, es que la implementación de un sistema de integración lleva implícito en sí, las características de cada proyecto software para cada organización, de manera que esta puede implantar aquellas mejoras y especializaciones en el modelo que considere beneficioso para su proceso. Esto da lugar a una mayor flexibilidad a la hora de implantar metodologías de integración continua en una organización de desarrollo software.

Cada organización implanta su propia versión de integración continua en función sus necesidades y características del proyecto. En nuestro caso, se le va a dar más importancia al hecho de que se mantenga el repositorio sin fallos ni errores, adaptando y mejorando los procesos de corrección de fallos o *nursing*, que vienen impuestos de por sí en la integración continua, según las recomendaciones definidas en el marco teórico.

4. PROPUESTA DE MEJORAS EN EL MODELO INICIAL

En este capítulo de se va evaluar sobre el modelo inicial, uno de los aspectos más importantes de un sistema de integración continua, la respuesta ante fallos[23]. Es decir, la capacidad de este para encontrar y descartar los cambios que hayan introducido fallos en la rama principal del repositorio, de forma automática o asistida, con el objetivo de mantener el repositorio sin fallos y no comprometer el sistema de integración y el avance del proyecto.

En la introducción de este trabajo, se comentaba que una de las características más importantes de un sistema de integración continua es la constante reiteración de ejecución de pruebas en la rama de integración, con el objetivo de detectar los fallos en el código antes de que llegue a entornos de producción, reduciendo así el coste que conllevaría corregirlos en ese punto. Por otra parte, una de las recomendaciones de las que se ha tomado como referencia para desarrollar el sistema inicial de integración continua, establece que el propietario del cambio que haya introducido el fallo, es el responsable de corregirlo o descartarlo del repositorio principal si así procede[24].

Este planteamiento sugiere dos posibles acciones. En primer lugar, el sistema ha de detectar lo antes posible cuando se haya introducido un fallo en la rama de integración del producto software, antes de oficializar la versión como *release* y, es más, debería ser capaz, en la medida de lo posible, de identificar concretamente el cambio o los cambios que han introducido fallos y qué fallos produce. Y, en segundo lugar, sería muy recomendable automatizar el proceso de descarte o revertir los cambios introducidos entre versiones oficiales, para desbloquear el sistema de integración y que el resto de la organización pueda seguir trabajando e integrando sus cambios mientras los responsables de los fallos arreglan sus cambios en local.

[23] En este contexto, se entiende por fallos a errores de compilación, UT o FT.
[24] Véase apartado 2.2. Recomendación 6: "Arreglar los errores de la construcción inmediatamente"

A este proceso de detección y corrección de fallos en la rama principal del repositorio se le conoce comúnmente en integración continua como "*nursing*", y se ha hecho referencia a él durante todo el documento. Veremos en este capítulo en qué consiste y un ejemplo de implementación.

4.1 EVALUACIÓN DEL MODELO INICIAL

Tal y como se ha planteado el modelo inicial desde el principio y, teniendo en cuenta las recomendaciones que se ha tomado como referencia para desarrollar el sistema inicial de integración continua, el sistema implementado sería capaz de detectar fallos en el repositorio entre intervalos de ejecuciones de construcciones candidatas a *release*. Esto supone el bloqueo de la rama master hasta que se identifiquen qué fallos rompen la integración y se asigne a alguien para que proceda a descartarlos del repositorio o introducir nuevos cambios que los arreglen.

Desde el punto de vista del cumplimento con los objetivos definidos para un sistema de integración continua, este comportamiento es totalmente válido, pues asegura la detección temprana de errores en el sistema de integración y evita que se introduzcan nuevos cambios antes de arreglar los fallos encontrados. Desde el punto de vista de la eficiencia que se le exige a un proceso lo más automatizado posible, como un sistema de integración continua, no lo es, pues fomenta errores por factor humano que pueden poner en riesgo la continuidad y el aseguramiento de la calidad de este sistema, una de las mayores cualidades de la integración continua.

Ante la imposibilidad de continuar con la integración por un fallo encontrado en el sistema de integración, tras ejecutar el proceso de construcción de la *release*, el personal del departamento de *release* debería evaluar cada una de las *builds snapshot* de cada *commit*[25] perteneciente a esa versión, de manera manual o usando los Jobs asistidos de Jenkins, hasta encontrar aquella *build* donde empieza a fallar y, por consiguiente, el *commit* candidato a descarte.

[25] En este contexto, se hace referencia a cambio en el repositorio o commit de manera indiferente, pues es lo mismo.

Esto no asegura que los *commits* introducidos posteriormente no contengan fallos, pues pueden verse afectados de manera que el primer fallo enmascare al resto, por lo que se tendría que volver a evaluar el sistema de nuevo, aumentando el riesgo de error por factor humano. Además, sería un proceso tedioso y largo, y obliga a que haya alguien del departamento dedicado a este tipo de tareas.

Hay que tener en cuenta que, en el peor de los casos, si el personal encargado de evaluar esta tarea no fuese capaz de encontrar el cambio o los cambios que introducen fallos en el repositorio principal, se han de descartar todos los cambios desde la última versión estable, para mantener la rama principal del repositorio sin fallos, penalizando así todos aquellos cambios válidos que no contengan fallos.

Otra cosa a tener en cuenta, en este sentido, es la cantidad de cambios que puede haber en el repositorio desde la anterior versión estable. Si damos por hecho que por cada cambio en el repositorio aseguramos el nivel de UT, y por cada versión estable aseguramos el nivel de FT, si el proceso de validación de la candidata a *release* encuentra un fallo de FT, en el mejor de los casos este se ha introducido en el último *commit* de este intervalo, pero, qué pasaría si ha sido introducido en el primer *commit*, y en el caso de que entre versiones candidatas haya unos 50 *commits*. Desde el luego el personal encargado de evaluar este análisis tomará ciertas represarías contra el autor del cambio del error en el repositorio, en el caso de que logre encontrarlo y, por supuesto, todo ese tiempo estaría bloqueado el proceso de integración y nadie podría subir cambios al repositorio. Una manera de reducir este problema es ejecutar la tarea de construcción de la *release*, al menos, una vez al día, como se comentó en el diseño del sistema de integración continua, de esta manera se asegura que el *scope* de pruebas funcionales se ejecuta una vez al día, con la consecuencia de no acumular demasiados cambios a evaluar entre versiones candidatas a *release*.

Para la evaluación de este sistema de integración continua se va a proponer un ejemplo de un caso en el que se han introducido un conjunto de cambios en el repositorio, unos 8 *commits*, desde la última vez que se oficializo la versión 3.0.0, hasta la ejecución de la construcción de la release, con objetivo de sacar una build candidata a release 3.0.1. Es entonces cuando este proceso de construcción de la release se detiene al detectar errores, tanto de UT como de FT.

A continuación, se muestra el ejemplo descrito anteriormente. Un intervalo de 8 *commits* introducidos en la rama *master* del repositorio, desde el ultimo *commit* perteneciente al de la ejecución de la *build release*, "Version Update 3.0.0". Se puede ver en la descripción de cada *commit*, cómo diferentes usuarios han introducido cambios en diferentes contextos del proyecto, tanto para añadir documentación como para añadir componentes software de la *Feature* 5.

```
(HEAD -> master, origin/master)
6977725 Feature 5: Reconfiguring RetencionTest (user5)
e6f7836 Feature 5: Reconfiguring pom.xml (user5)
2f2fc5f Feature 5: Add new test for new retaining at AeatTest
(user4)
52ad64c Feature 5: Reconfiguring Jenkinsfile (user5)
6017a82 Feature 5: New retaining section at Aeat (user4)
d7f14ed Added featue 5 to release notes documentation (user3)
c6d725b Renamed Funtional Test Cases output message (user2)
a2938ae Added documentation for the release notes 3.0.0 (user1)
8dee19f VERSION UPDATE: Release 3.0.0 (jenkins)
```

Si observamos el historial de tareas del proceso *build commit*, tarea que se ejecuta por cada *commit* y que lanza una compilación y una regresión de UT, vemos como la construcción 209, perteneciente al *commit* "*2f2fc5f Feature 5: Add new test for new retaining at AeatTest*" del usuario 4, es amarilla, indicando que es inestable y, por tanto, fallan casos de UT. Cuando se definió la construcción de la *build* se definió que el proceso no descartaría la construcción si fallase algún caso del *scope* de UT, pero se marcaria como inestable, notificando al propietario para que procediese a su reparación lo antes posible sin bloquear la integración, por cuestiones de agilidad para el resto de los desarrolladores. Otra cosa a tener en cuenta es que, las siguientes *builds* después del fallo, es decir, la 210 y la 211, también están marcadas como

inestables. Esto no quiere decir que contengan fallos, ya que, no han introducido posteriormente ningún *fix*[26] para corregir el de la construcción 209, por tanto el fallo es acumulado en las siguientes construcciones.

En la siguiente ilustración se muestra el historial de tareas del proceso *build commit*, para este intervalo de *commits* que se ha descrito.

Ilustración 33: Jenkins. Historial ejecuciones Build_commit para el intervalo evaluado.

Teniendo en cuenta esto y sabiendo que, en el anterior proceso de construcción de la *release*, para la versión 3.0.0, no hubo ningún fallo de compilación ni de UT ni de FT y, por tanto, se pudo construir la *build* para desplegarla al repositorio de artefactos correspondiente, sin ningún tipo de error, como se aprecia en la siguiente ilustración. Con la información que tenemos, solo podemos saber que en la siguiente construcción de *release* habrá, al menos, un fallo de UT, producido por el anterior *commit* mencionado.

Ilustración 34: Jenkins. Pipeline de la tarea Build_release para la versión 3.0.0.

[26] Se entiende por fix a la introducción de un cambio posterior que arregle el fallo introducido previamente.

Sin embargo, al lanzar el proceso de construcción de la *release*, para la siguiente candidata a estable, cambiando la revisión del producto de 0 a 1, es decir, de la versión 3.0.0 a la 3.0.1, el proceso se detiene porque en el HEAD de la rama máster se encuentran fallos tanto a nivel de UT como de a nivel de FT y, por tanto, no puede completar el proceso de construcción de la *release*. Como se muestra en la siguiente ilustración, resultado de lanzar dicho proceso.

Ilustración 35: Jenkins. Pipeline fallido de la tarea Build_release de la versión 3.0.1.

Así como nos muestra el historial del proceso de *build commit*, (ver ilustración 34), sabíamos que el proceso de *build release* podría encontrar un fallo a nivel de UT introducido por el *commit* 2f2fc5f, pero no sabíamos ni cuántos casos fallarían de UT, ni que hubiese también fallos de FT desde la anterior versión. Ahora, tras la ejecución del proceso de *build release* sabemos que el HEAD de la rama contiene los siguientes fallos:

✓ A nivel de Unit Test:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO]
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   NominaTest.testCalculaSalarioBruto1:17 expected:<900.0>
but was:<800.0>
[ERROR]   RetencionTest.testSetRetencion:16 Se espera Excepcion por
Retencion negativa
 [ERROR] Tests run: 18, Failures: 2, Errors: 0, Skipped: 0
[INFO]
[ERROR] There are test failures.
```

- ✓ A nivel de Functional Test:

```
Running Functional Test Suite for Java Application:

Test Case 1: Evaluating Nomina result for TEMPORAL:
RESULT: TestCase 1 PASSED.

Test Case 2: Evaluating Nomina result for EMPLEADO:
RESULT: TestCase 2 PASSED.

Test Case 3: Evaluating Nomina result for ENCARGADO:
RESULT: TestCase 3 PASSED.

Test Case 4: Set new Tramo and Retencion to evaluate new Nomina:
RESULT: TestCase 4 FAILIED.

RESULT: Functional Test Suite FAILED.
```

Con la información que nos proporciona esta tarea, el personal encargado del departamento de *release* tendría que evaluar, *commit* a *commit*, en su repositorio local, tanto los niveles de UT como los de FT, para todo el intervalo. Generalmente empezaría evaluando el último *commit* introducido, en orden descendente hasta evaluar el *commit* que no tenga fallos. Lo podría hacer de la siguiente manera:

```
# git pull origin master
Desde git+ssh://localhost/repo/git/maven_project
 * branch            master     -> FETCH_HEAD

# git log --oneline | head -n 1
6977725 (HEAD -> master) FEATURE 5: Reconfiguring RetencionTest

# mvn clean test  // PARA EVALUAR UT
# javac -cp .:/usr/share/java/*  ./src/main/java/es/uned/ci/*.java
./src/test/java/es/uned/ci/*.java -d . // PARA COMPILAR EN LOCAL

# java es.uned.ci.RunFTest // PARA EJECUTAR FT
# git checkout e6f7836 // PARA EVALUAR EL SIGUIENTE COMMIT, EN ESTE
CASO "e6f7836 Feature 5: Reconfiguring pom.xml"
```

Y así seguiría evaluando, commit a commit, registrando y apuntando los fallos para deducir aquellos que introducen fallos en el repositorio, con el consiguiente riesgo de equivocarse y descartar un commit que sea válido, y tener que volver

a introducirlo para volver a repetir el análisis de nuevo. Lo que supone un bloqueo en la integración continua, pues los siguientes *commits* no podrán evaluar los casos que estén fallando, por enmascaramiento de error, de manera que se tendrá que bloquear la rama máster del repositorio hasta que se solucione el problema.

Este es un típico proceso de *nursing* manual que se suele llevar a cabo en los departamentos de *release* más de lo esperado. Por muy automatizado que sea el proceso de integración, no se podrán evaluar todos los casos de prueba en todo momento y por cada cambio en el repositorio, siendo en posteriores evaluaciones de más alto nivel, cuando se detectan fallos introducidos anteriormente. Si el error es grave, afecta a casos de prueba con alto nivel de requerimientos o a la propia integración del producto, lo que se suele hacer es bloquear la rama de integración hasta que se solucione el problema, bien revirtiendo el cambio que ha introducido el fallo, que es lo más inmediato y efectivo, o bien introduciendo el cambio que lo solucione. En algunas organizaciones tienen un indicador con luz verde o luz roja, como si de un semáforo se tratase, para notificar al resto del personal el estado de LSV, si está bloqueada y no se puede integrar o si esta desbloqueada con luz verde para poder integrar.

4.2 SOLUCIÓN PROPUESTA: AUTOMATIZACIÓN DEL PROCESO DE NURSING

Si damos por hecho que el caso descrito anteriormente se va a producir con cierta frecuencia, a no ser que los desarrolladores y todos aquellos que introduzcan cambios en el repositorio lo hagan siempre sin ningún tipo de error, cosa que no va a pasar, añadir mejoras al proceso de corrección del repositorio siempre va a ser beneficioso para el proceso de integración continua. Cualquier proceso que asista o arregle la integración, de alguna manera, o simplemente informe de los cambios que introducen errores, evitando que lo tenga que hacer otra persona o facilitándole a esta dicha tarea, sin duda su implantación y uso estaría más que justificado. Es por ello que en esta sección se van a enumerar, desarrollar y evaluar, algunas de estas soluciones basadas en automatización de este proceso, con el objetivo identificar los fallos y proceder a revertirlos en el orden correspondiente.

Existen muchas técnicas para evaluar el proceso de identificación de fallos en un repositorio en un intervalo dado. Igual que lo haría una persona, al detectar un fallo en el HEAD de la rama de dicho repositorio, empezaría a evaluar cada *commit*, bien recorriendo la lista del intervalo a evaluar en orden ascendente, bien recorriéndola en orden descendente, y evaluando cada *commit* por separado. Hay que tener en cuenta que los *commits* anteriores al evaluado afectan al mismo, por lo que siempre hay que evaluar deltas o incrementos de código basados en una *baseline*[27] que no contenga fallos. Esto quiere decir que, por ejemplo, si el *commit* C está encima del B y del A, y el A contiene un cambio que hace que falle un caso de FT, tanto el *commit* B como el C contendrán el mismo fallo y, por tanto, habría que evaluar B sin A y C sin A, a no ser que B contenga otro fallo, entones C tendría que evaluarse sin B y sin A y, por supuesto, todo esto dando por hecho que antes de A no haya ningún otro fallo.

Este hecho descrito da lugar a una gran cantidad de casuísticas posibles, que hacen que los algoritmos que se van a plantear se describan como de *best effort*[28], y más aún, teniendo en cuenta una de las características de GIT en cuanto a la resolución de conflictos se refiere. Esto es, ante cualquier conflicto dado por dos *commits* que intenten modificar partes del mismo fichero, GIT asume que no puede resolverlo automáticamente y propone su resolución a una persona, que deberá decidir cómo debería quedar el fichero resultante. Este caso suele darse cuando se "*mergean*[29]" o "*rebasean*[30]" varias ramas o, como es el caso que nos interesa, cuando se quiere revertir un *commit* que fue añadido al repositorio antes de otro que también modifica el mismo fichero. Este hecho limita parte la automatización del proceso ya que habría que revertir ambos *commits*, penalizando uno de ellos por no poder resolver el conflicto automáticamente. En ese caso lo mejor sería informar del *commit* candidato a revertir para que proceda a hacerse manualmente, sin aplicar ningún *revert* automático.

Una vez descrito el problema y propuesta una posible solución, se va a plantear a continuación el diseño y el modelo del sistema que implemente esta solución.

[27] Se entiende por baseline a la base de código con la que comparar.
[28] Mecanismo por el cual no asegura su éxito al 100, siendo esta su intención.
[29] Proceso de *git merge* de 2 ramas.
[30] Proceso de *git rebase* de una rama en otra.

4.2.1 DISEÑO DE LA SOLUCIÓN PROPUESTA

Para llevar a cabo la implementación de un proceso automático que asista al proceso de construcción de cada *release*, en caso de que este falle, bien sea porque falle algún caso de UT y, por tanto, la *build* sea inestable, bien sea porque falle algún caso de FT y, por tanto, la *build* no sea válida de cara a versión oficial, se va a añadir una nueva vista en Jenkins que contenga los Jobs o tareas que implementen estas acciones.

Ilustración 36: Jenkins. Tareas de la vista Nursing.

En esta vista, llamada *nursing*, se puede apreciar 4 Jobs o tareas, aunque en realidad solo 3 de ellos es de implementación, porque como veremos a continuación, el *job GoF_command_pattern* representa una interfaz que implementan los demás Jobs del *nursing*. Cada uno de ellos va a representar una implementación diferente del modelo planteado con diferentes técnicas, que también se verá a continuación, pero con un objetivo común, el de intentar evaluar la rama máster del repositorio, en el intervalo de *commits* dado desde el ultima construcción estable, con el objetivo de obtener la lista de *commits* que introduzcan fallos de UT y FT y así proceder a revertirlos del repositorio, intentando penalizar lo menos posible aquellos *commits* que sean válidos.

En primer lugar, caracterizamos el comportamiento que ha de tener cualquiera de los procesos de *nursing* que se van a implementar mediante el siguiente diagrama de secuencia, siguiendo el mismo modelo planteamiento para las descripciones del diseño de los Jobs de construcción de la *build*, en el capítulo 3, sección 2.

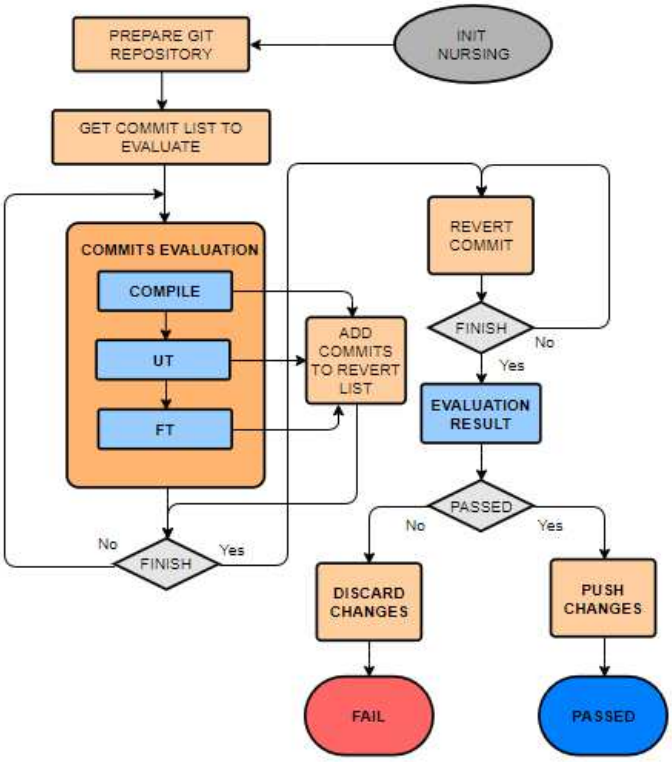


Ilustración 37: Diagrama de secuencia. Proceso Nursing del repositorio.

A grandes rasgos, lo que se puede apreciar en este diagrama de secuencia es lo siguiente: Una vez se inicia, lo primero que hace es clonar el repositorio del proyecto, rama master, para preparar posteriormente el intervalo de *commits* a evaluar, dado por el ultimo *commit* del "*version update*" *de release*. Una vez hecho esto ya está en condiciones de evaluar la lista de commit. Para ello usará el subproceso "*commits evaluation*", que se implementará de una manera u otra dependiendo de la versión.

Una vez creada la lista de posibles *commts* a revertir con el anterior subproceso, aquellos que sean identificados como fallidos mediante las ejecuciones correspondientes de UT y FT, se procederá a aplicar los *reverts* correspondientes en la rama local, para posteriormente evaluar y dar así veredicto al resultado del proceso, indicando si fue capaz o no de reparar la rama. En tal caso aplicará los cambios al repositorio central mediante un *push*, aunque esto podría ser opcional.

Del diagrama anterior también se puede deducir que este proceso se lleva a cabo mediante la ejecución de ciertas acciones que pueden agruparse de la siguiente manera:

- ✓ *Reset*: Con el objetivo de clonar el repositorio y dejar el repositorio local lo más limpio posible para iniciar el proceso.
- ✓ *Compile*: Realiza la compilación del commit que se quiera evaluar, es decir, aquel al que apunte el repositorio en ese momento.
- ✓ *Unit Test*: Lanza el scope de UT sobre los artefactos previamente compilados.
- ✓ *Function Test*: Lanza el scope de FT sobre los artefactos previamente compilados.
- ✓ *Evaluation:* Con la información obtenida tras la compilación y ejecución de pruebas, mediante el uso de las acciones anteriores, aplica las técnicas necesarias para dictaminar qué commits han de revertirse en el intervalo evaluado.
- ✓ *Evaluation UT:* Variante del anterior que solo aplica a los commits afectados por los casos fallos en UT.
- ✓ *Evaluation FT:* Variante del anterior que solo aplica a los commits afectados por los casos fallos en FT.
- ✓ *Revert:* Acción que aplica los cambios revertidos en local al repositorio central mediante un *git push*.
- ✓ *Logging:* Mostrar los cambios efectuados en el repositorio a modo de información mediante un *git log*.

Esta manera de contextualizar la implementación por acciones que, dependiendo de cada técnica se implementará de una manera u otra, pero que todas constarán de las mismas acciones, nos hace pensar en un patrón de diseño óptimo para definir el comportamiento de esta tarea. Por tanto, el diseño entero de esta tarea de *nursing*, podría estar basada en un patrón de comportamiento, perteneciente al GoF[31] [Gamma, Helm, Johnsosn and Vlissides, 1995], patrón comando, *command pattern*[32].

[31] Siglas en inglés de Gang of Four Design Patterns. Para más información véase la referencia bibliográfica.
[32] Patrón de comportamiento, que crea objetos para encapsular acciones y sus parámetros.

El patrón comando permite encapsular métodos en objetos, implementando una interfaz dada, de manera que se pueda externalizar la llamada a dichos métodos. Esto nos permite definir un cliente que realice peticiones sin necesidad de conocer la acción, de la misma manera, se pueden realizar cambios sin impactar en dicho cliente, pues este va a ejecutar la acción sin saber cómo esta está implementada. De esta manera, independiza el momento de la petición del de la ejecución, facilitando así la parametrización de las acciones a realizar.

A continuación, se muestra el diagrama de clases UML que representa el modelo del patrón de comportamiento comando, "*command pattern*".

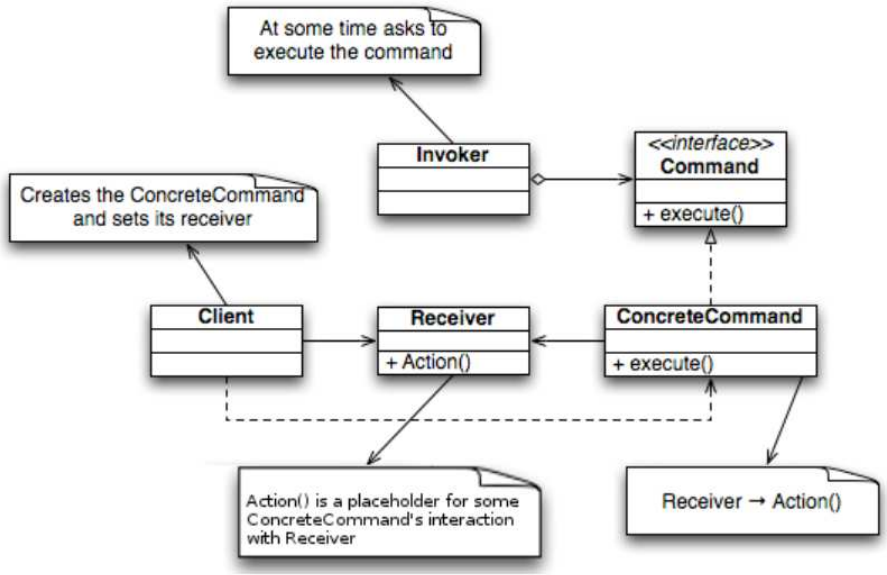


Ilustración 38: Diagrama UML GoF Command Pattern.

En el caso que nos ocupa, siguiendo este modelo de diseño, se va a definir una interfaz con las acciones ejecutar. Estas, serán implementadas por cada una de las clases que represente cada comando definido anteriormente. De manera que el objeto cliente, a través del invocador y el receptor, pueda ejecutar cada acción implementada por en cada clase para cada comando. De esta manera, quedaría así el diagrama de clases UML para el diseño de la implementación propuesta de la tarea *nursing*, que aplicaría a las 3 implementaciones de las 3 versiones diferentes.

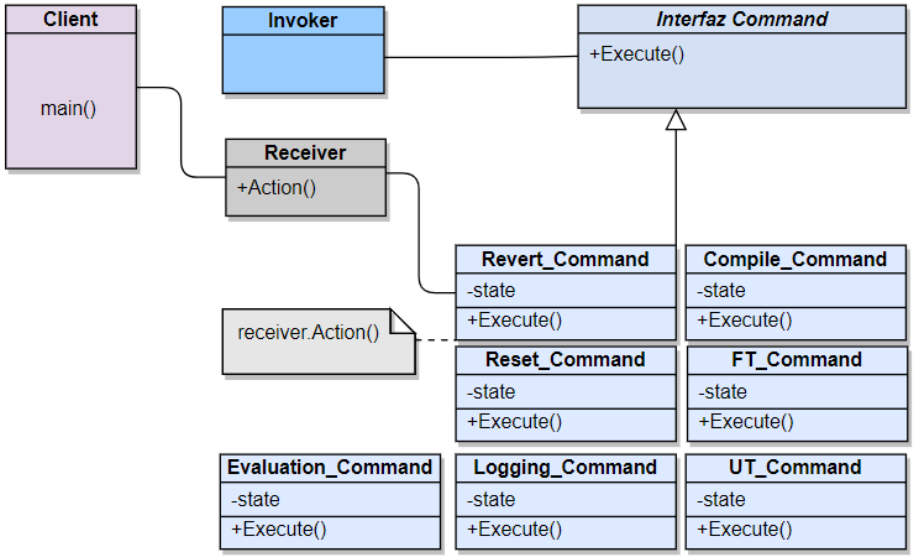


Ilustración 39: Diagrama de clases UML del proceso Nursing.

4.2.2 IMPLEMENTACIÓN DEL DISEÑO DE NURSING PROPUESTO

Lo interesante de este diseño, de cara a la implementación, es ver como se ha encapsulado el código o la implementación de cada acción definida como comando, en cada clase comando. De manera que podamos reusar el mismo diseño para las diferentes implementaciones que hagamos del proceso *nursing*. De esta manera, la invocación de estos comandos, quedaría igual para todas las implementaciones que hagamos de cada versión, quedando la implementación del main(), como se muestra a continuación.

```
def main():

# Declare invoker and receiver objetcts:
    receiver = Receiver()
    invoker  = Invoker()

# Store the secuence of commands to execute:
    invoker.store_command(Reset(receiver))
    invoker.store_command(Compile(receiver))
    invoker.store_command(UnitTest(receiver))
    invoker.store_command(FunctionTest(receiver))
    invoker.store_command(Evaluation(receiver))
    invoker.store_command(Revert(receiver))
    invoker.store_command(Logging(receiver))

# Execute the secuende of commands by method execute commands()
    invoker.execute_commands()
```

Cabe destacar que la implementación se ha hecho en lenguaje Python, a diferencia del resto de *jobs* que se han implementado en Bash, Groovy o, a través del fichero pom.xml siguiendo el formato XML definido por Maven. En este caso, se han usado las librerías de Python 2.7 y para la implementación de la librería que se va a desarrollar, llamada abc, se ha hecho uso de otras librerías como *os*[33] o commands. Dado que estamos implementando acciones que ejecutan comandos de GIT, podríamos haber usado otras librerías como GitPython[34] o pygit2[35], que modelan comandos git para diferentes acciones comunes. En nuestro caso hemos desistido de su uso, ya que la implementación está motivada por otros objetivos, más que los de la ejecución de comandos de GIT.

Las implementaciones de todas las acciones necesarias para cada una de las versiones a desarrollar son iguales, con la excepción del comando *evaluation*, es decir, la clase Evaluation. Esta clase va a contener toda la lógica necesaria para evaluar todos los *commits* en el intervalo definido y descubrir cuáles de ellos causan fallos en el repositorio, tanto de compilación como de UT y FT, de manera que genere una lista para su *revert*, acción llevada a cabo por la clase *Revert*.

Para desarrollar la clase *Evaluation* se han definido 3 técnicas, definidas y clasificadas a continuación. Posteriormente se hará una evaluación de todas ellas y se caracterizarán para una mejor comprensión de su aplicación y uso. Destacando sus ventajas y desventajas, y comparándolas tras su ejecución con el ejemplo propuesto al inicio de esta sección.

Cada uno de los siguientes apartados, representa una implementación de la etapa de evaluación del proceso de *nursing*, correspondiente al comando *evaluation* y, por tanto, la clase Evaluation.

[33] Miscellaneous operating system interfaces. Source code: Lib/os.py
[34] https://pypi.org/project/GitPython/
[35] https://www.pygit2.org/

4.2.3 RECORRIDO LINEAL SIMPLE SIN REITERACIONES

Este es el algoritmo más sencillo para implementar la clase Evaluation. Consiste en un recorrido lineal de todos los *commits* del intervalo a evaluar, desde el último *commit* que fue añadido al repositorio, hasta que evalúe uno que no contenga fallos o bien, llegue al final del intervalo. De esta manera, añade todos aquellos *commits* a la lista de *commits* a revertir, sin tener que volver a reiterar la evaluación dado que la selección ya es firme en la primera iteración. Posteriormente revertirá uno a uno en orden de evaluación, de manera que se reduzca la posibilidad de conflicto entre *commits* a revertir.

Como ventaja presenta una ejecución bastante rápida del proceso de *nursing*, asegurando casi al 100% la corrección de la rama master, ya que es muy poco propenso a dar conflictos por el *revert* de los *commits* evaluados, dado que estos se revierten en orden inverso al que fueron introducidos.

Como desventaja presenta una penalización bastante grande, ya que, en la lista de *commits* a revertir va a penalizar aquellos que se encuentren entre los que contengan fallos, aunque estos sean válidos. Por ejemplo, si en un intervalo de 10 *commits*, el tercero y el sexto contienen fallos, se revertirán los 6 últimos *commits* que hayan sido añadidos al repositorio, penalizando por tanto los *commits* 1, 2, 4 y 5.

Si evaluamos con esta implementación el ejemplo inicial, tras la ejecución de la construcción de la *release* 3.0.1, donde el *job* de construcción de la *bulid* indicaba que en el HEAD de la rama máster del repositorio había 2 fallos de UT y uno de FT, ver sección 4.1 “Evalucacion del modelo inicial”, obtenemos los siguientes resultados, en el que se puede observar como el proceso de *nursing* propone que ser reviertan los 5 últimos *commits* introducidos en el repositorio.

```
728a7ed Revert "Feature 5: New retaining section at Aeat"
b4ac31a Revert "Feature 5: Reconfiguring Jenkinsfile"
e0c40f3 Revert "Feature 5: Add new test for new retaining at
AeatTest"
0f283a7 Revert "Feature 5: Reconfiguring pom.xml"
e476724 Revert "Feature 5: Reconfiguring RetencionTest"
6977725 Feature 5: Reconfiguring RetencionTest
e6f7836 Feature 5: Reconfiguring pom.xml
2f2fc5f Feature 5: Add new test for new retaining at AeatTest
52ad64c Feature 5: Reconfiguring Jenkinsfile
6017a82 Feature 5: New retaining section at Aeat
d7f14ed Added featue 5 to release notes documentation
c6d725b Renamed Funtional Test Cases output message
a2938ae Added documentation for the release notes 3.0.0
```

4.2.4 RECORRIDO LINEAL INVERSO CON ITERACIONES CANDIDATAS A DESCARTE

Esta implementación es bastante más compleja y requiere más procesamiento que la anterior. En una primera evaluación, realiza un recorrido inverso de la lista de *commits* del intervalo a evaluar, empezando por el primero en ser introducido en el repositorio, hasta el último. Evaluando uno a uno y añadiéndolos a una lista de candidatos a descarte, en caso de que fallen. Una vez terminada esta primera iteración, en el que se ha añadido aquellos *commits* evaluados con resultado fallido en una lista de candidatos a descarte, se realiza un recorrido lineal de esta lista, evaluando cada uno de ellos de nuevo, esta vez habiendo aplicado el *revert* del *commit* anterior. Es decir, si de la primera lista de 10 *commits* considera 6 candidatos a descarte, ahora evaluara estos 6, aplicando en primer lugar el *revert* del 6 para evaluar el 5, y aplicando el *revert* del 6 y del 5 para evaluar el 4, solo en el caso de que el 5 haya sido fallido, en caso contrario, solo aplicaría el rever del 6 para evaluar el 4, y así sucesivamente hasta el final de la lista candidata. Esta iteración da por hecho que el primer *commit* en evaluar, el 6 en este caso, siempre va a contener algún error.

Al final de haber recorrido la lista candidata a descarte, habrá intentado revertir solo aquellos *commits* que contengan errores, penalizando lo menos posible los *commits* del repositorio que sean válidos.

Este algoritmo presenta la ventaja de ser más efectivo con los *commits* a revertir que el anterior ya que, al reiterar la evaluación de cada *commit* habiendo revertido previamente los *commits* con fallos, independiza de alguna manera el *commit* a evaluar, dando lugar a una menor penalización de los *commits* válidos del repositorio, aunque ello no asegure al 100% que no penalice ninguno.

Como desventaja presenta una mayor complejidad en el algoritmo, lo que le lleva al sistema a consumir más recursos y a tardar más tiempo en completar la tarea. Otra desventaja, que no presentaba el algoritmo anterior, es que no asegura al 100% el descarte del *commit* mediante el *revert*, ya que estos son revertidos en el mismo orden en el que fueron introducidos, pudiendo dar conflictos al aplicar los cambios. El sistema en este caso notifica de la imposibilidad de aplicar el *revert* candidato.

```
a1318b1 Revert "Feature 5: Reconfiguring RetencionTest"
d72b4c6 Revert "Feature 5: Reconfiguring pom.xml"
33082f2 Revert "Feature 5: Add new test for new retaining at
AeatTest"
c43454f Revert "Feature 5: New retaining section at Aeat"
6977725 Feature 5: Reconfiguring RetencionTest
e6f7836 Feature 5: Reconfiguring pom.xml
2f2fc5f Feature 5: Add new test for new retaining at AeatTest
52ad64c Feature 5: Reconfiguring Jenkinsfile
6017a82 Feature 5: New retaining section at Aeat
d7f14ed Added featue 5 to release notes documentation
c6d725b Renamed Funtional Test Cases output message
a2938ae Added documentation for the release notes 3.0.0
```

4.2.5 RECORRIDO LINEAL INVERSO CON EVALUACIONES REITERADAS INDEPENDIENTES

Este algoritmo es aún más complejo que el anterior pues, al igual que en el caso anterior, realiza un recorrido inverso inicial para obtener la lista de *commits* candidatos a revertir, para realizar una segunda iteración evaluando cada *commit* con la lista candidata. Pero en este caso realiza una tercera interacción anidada a la segunda, para determinar la lista de *commits* a revertir, aplicando dicha lista a cada evaluación e independizando el resto del contenido de la rama mediante un *checkout* en ese punto del repositorio donde apunte el *commit* evaluado.

Este algoritmo asegura que cada *commit* evaluado es independiente del resto pues, por una parte, al realizar un *checkout* del *commit* evaluado, los *commits* por encima de él no interfieren en la evaluación y, por otra parte, el contador de la lista de *commits* a revertir se aplica siempre antes de la evaluación del *commit* a evaluar, para no influir negativamente en él.

Esta es la implementación más precisa para independizar la evaluación de cada *commit*, pero el algoritmo es mucho más complejo y pesado que los anteriores, pudiendo tardar bastante en completar el proceso por la anidación de bucles en la evaluación.

Si aplicamos este algoritmo en el ejemplo, podemos ver que en este caso se han revertido solo 3 *commits*, 1 menos que el anterior y 2 menos que el primero. Por lo que se deduce de este que se no penaliza ninguno y que el primer algoritmo penalizó 2 commits, y el segundo 1. Nótese que los tres algoritmos coinciden en los tres commits a revertir pues, en caso contrario, habrían fallado en su objetivo principal.

```
02d070c Revert "Feature 5: New retaining section at Aeat"
da724d3 Revert "Feature 5: Add new test for new retaining at AeatTest"
caa1b47 Revert "Feature 5: Reconfiguring RetencionTest"
6977725 Feature 5: Reconfiguring RetencionTest
e6f7836 Feature 5: Reconfiguring pom.xml
2f2fc5f Feature 5: Add new test for new retaining at AeatTest
52ad64c Feature 5: Reconfiguring Jenkinsfile
6017a82 Feature 5: New retaining section at Aeat
d7f14ed Added featue 5 to release notes documentation
c6d725b Renamed Funtional Test Cases output message
a2938ae Added documentation for the release notes 3.0.0
```

4.2.6 COMPARATIVA ENTRE LAS TRES VERSIONES

Basándonos en la ejecución del ejemplo anterior para las 3 versiones diferentes, en nuestro sistema de integración continua, se va a realizar una comparativa de las tres ejecuciones, para evaluar el tiempo y la efectividad de cada algoritmo implementado.

En primer lugar, considerando que las tres ejecuciones fueron correctas, como así ha sido, procedemos a evaluar los tiempos que le ha llevado a cada tarea la ejecución del proceso de *nursing*, hasta completarla. En la siguiente ilustración se puede apreciar como la ejecución de cada tarea para el ejemplo dado le ha llevado a la primera implementación unos 3 minutos, a la segunda unos 5 minutos y 50 segundos, y a la tercera algo menos de 6 minutos y medio.

Ilustración 40: Jenkins. Tiempos de las tareas de la vista Nursing.

En segundo lugar, y bajo las mismas condiciones anteriores, hay que tener en cuenta también la efectividad del proceso. En este sentido se ha determinado que realmente hay 3 *commits* que introducen fallos en el repositorio desde la última versión estable del producto. El quinto *commit*, contando dese el HEAD de la rama, "`6017a82 Feature 5: New retaining section at Aeat`", introduce un fallo de FT en el repositorio que no fue identificado por el proceso de construcción de cada cambio, ya que este solo evalúa UT, y solo se pudo identificar como fallo cuando se ejecutó el proceso de construcción de la *release*. Por otra parte, los *commits* 1 y 3 introducen fallos en el código identificados por los test de UT, aunque estos sí fueron identificados cuando se introdujeron en el repositorio, pues el proceso de construcción de cada cambio si evalúa UT.

En cualquier caso, el único algoritmo implementado para la tarea del proceso *nursing* que ha sido capaz de identificar los *commits* que introducen fallos en el repositorio, tanto de UT como de FT, para el ejemplo dado, ha sido el tercero. Pudiendo proceder a la oficialización de la versión 3.0.1 sin penalizar ningún *commit* válido, solo aquellos que introducen fallos en el repositorio. Como se puede ver en la siguiente ilustración, donde se ve que la ejecución de la construcción es válida, tras haber revertido los 3 *commits* más el añadido del *commit* del *version update*.

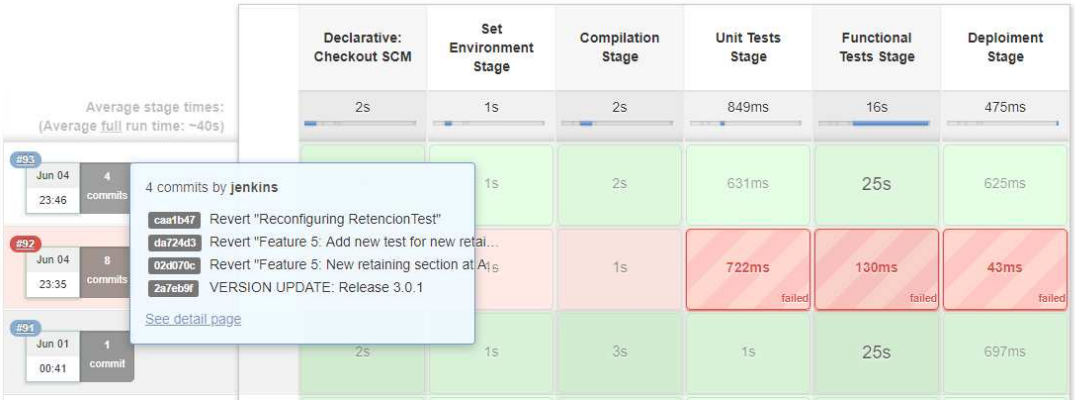


Ilustración 41: Jenkins. Pipeline válido de la tarea Build_release de la versión 3.0.1

Con esta información, podemos concluir la evaluación en la siguiente tabla, en la que se indica el nivel de complejidad y la supuesta tasa de éxito que se le ha dado a cada algoritmo, así como el tiempo que le ha llevado completar la tarea para el ejemplo dado, y los *commits* que se han penalizado en cada caso. Si bien, se puede considerar que, tanto el nivel de complejidad como la tasa de éxito son apreciaciones aproximadas, en el caso del tiempo invertido y los *commits* penalizados son datos obtenidos tras su ejecución en el sistema de integración continúa implementado.

ALTORITMO	COMPLEJIDAD	TIEMPO INVERTIDO	COMMITS PENALIZADOS	TASA DE EXITO
Nursing_1	BAJA	2 min 57 seg	Muchos: 2	Se asegura el 100 %
Nursing_2	MEDIA	5 min 50 seg	Pocos: 1	No se asegura el 100 %
Nursing_3	ALTA	6 min 23 seg	Ninguno: 0	No se asegura el 100 %

Tabla 5: Comparativa de los algoritmos para el proceso Nursing.

4.3 CONCLUSIONES DE LA PROPUESTA DE MEJORAS AL SISTEMA CI

Si un sistema de integración continua no puede evaluar el 100% de las pruebas cada vez que se introduzca un cambio en el repositorio, bien sea por la alta frecuencia de cambios en el repositorio, bien sea por la cantidad de pruebas a evaluar para un proyecto dado, teniendo que redimensionar el proceso de *testing* en varias etapas, como se ha hecho en este sistema, cualquier proceso de automatización y detección de errores del repositorio va a beneficiar siempre al proyecto. Más aun teniendo en cuenta que este tipo de procesos automáticos puede efectuarse en horas de poca afluencia del personal, como en horas nocturnas o fines de semana, de manera que no coincida con la interacción del resto de usuarios del sistema.

La desventaja en este sentido se puede dar cuando un proceso automatizado no pueda determinar la importancia de un cambio en el repositorio a revertir, ya que, puede que el error que se introduzca impacte menos en el proyecto que el hecho de revertir el contenido del cambio en sí. En ese caso, habría que definir un sistema de *flags* o banderas para considerar la importancia de los *commits* a evaluar, y un sistema de pesos para determinar, de igual manera, la importancia del caso evaluado. Se puede dar el caso de que una nueva *feature* integrada sea muy importante para un cliente, pero que esta rompa algún componente software de otra *feature* que usan otros clientes.

En este sentido, lo más común es asegurar la regresión del sistema y determinar que lo que antes funcionaba, sigua funcionando, y si no pude ser así, sacar una rama nueva para la implementación de esta nueva *feature* hasta que pueda integrarse sin impactar en el resto de componentes.

En cualquier caso, un proceso automatizado debe entenderse como una herramienta que asista al personal para ayudar a completar sus tareas, de la manera más efectiva y eficiente posible, no para sustituirle. Pues siempre se van a dar casos que no se hayan contemplado con anterioridad y que requieran de la toma de nuevas decisiones.

5. CONCLUSIONES Y TRABAJOS FUTUROS

Se concluye el trabajo realizado en base al alcance del trabajo y objetivos definidos en la sección introductoria de este documento.

En un primer lugar, se definió como un primer objetivo diseñar e implementar un sistema de integración continua en base a los requisitos y recomendaciones implantados por algunos autores reconocidos en esta metodología. Se verificó en el capítulo 3, cómo el modelo diseñado para un sistema de integración continua y su implementación cumplían con cada una de estas recomendaciones definidas en el marco teórico, dado un prototipo de proyecto software desarrollado previamente, necesario para la puesta en marcha y evaluación de dicho sistema de integración continua.

En segundo lugar, se definió como objetivo principal implementar y evaluar un proceso que mejorase en algún aspecto y de manera objetivamente cuantificable, el sistema de integración continua implementado previamente. En este sentido, se decidió mejorar una de las recomendaciones impuestas a esta metodología, correspondiente a la recomendación 6, "Arreglar los errores de la construcción inmediatamente" de manera directa y otras de manera indirecta, como la recomendación 4, "Actualizar el repositorio a diario", definidas en el marco teórico de este trabajo, capítulo 2. Para ello, se ha hecho uso de diferentes técnicas para implementar un sistema que automatice un proceso de identificación y corrección de errores en el repositorio, en un momento determinado y para un intervalo dado, arreglando así los errores en la construcción de manera desasistida. Hemos visto como la automatización este proceso puede mejorar el sistema en general, ya que esta puede corregir los errores de la construcción mucho más rápido que de hacerse manualmente, hecho que reduce el tiempo de bloqueo de la rama principal del repositorio, posibilitando o fomentando de alguna manera, que los integrantes del proyecto puedan actualizar y subir sus cambios en el repositorio más a menudo.

Para poder desarrollar un sistema de integración continua en base al modelo planteado, ha sido necesario desarrollar previamente un prototipo de proyecto software, de manera que pueda usarse como ejemplo para probar y evaluar el sistema implementado.

Este prototipo tiene un carácter demostrativo, de manera que sus dimensiones se ajustan a las necesidades de este trabajo. Esto da lugar a un menor tiempo de compilación y de ejecución de las pruebas, que lo que le llevaría a un proyecto real. Pero nos ha servido para comprender los tiempos que llevaría un sistema real si extrapolásemos sus dimensiones. En este sentido influyen muchos factores, como el tiempo de procesamiento del servidor que va a compilar y ejecutar las pruebas, o su capacidad de memoria. Pero si, por ejemplo, redimensionásemos este prototipo a un proyecto real 100 veces mayor, teniendo en cuenta que al sistema le lleva unos 10 segundos preparar el entorno, compilar y ejecutar unas 18 pruebas unitarias para cada construcción por cada cambio en el repositorio, al proyecto redimensionado con 1800 pruebas unitarias, le llevaría unos 16 minutos en un sistema virtual, y unos 10 en un sistema real. Manteniendo el mismo porcentaje, para el caso de la construcción de la candidata a estable, con unas 400 pruebas funcionales, le llevaría al sistema real unos 80 minutos, unos 50 en caso de usar máquinas reales en lugar de virtuales. Esto implicaría, por otra parte, que el proceso de corrección del repositorio, desarrollado en el capítulo 4, podría llevarle a un sistema real aproximadamente de entre 5 y 10 horas para un sistema virtualizado, y unas 3, 6 horas para un sistema real, dependiendo del algoritmo empleado. Por tanto, se establece que su implantación en un proyecto software de dimensiones reales es totalmente factible, pues el objetivo de este es ejecutarse en horas de poca afluencia del personal.

Otro hecho a tener en cuenta en relación con lo comentado anteriormente es que, dado que el sistema tarda 10 segundos en realizar la construcción de cada cambio y unos 50 segundos para la construcción de la candidata, sería factible para este prototipo ejecutar todo el proceso de pruebas definido en la construcción de la candidata en cada construcción de cada cambio, haciendo innecesario el proceso de corrección definido en el capítulo 4. Sin embargo, hay que tener en cuenta que, si de nuevo se extrapola a un proyecto real, el proceso de la construcción de la candidata le llevaría al sistema entre 50 y 80 minutos, siendo inviable su ejecución para cada cambio del repositorio. Quedando justificado, por tanto, el modelo definido por 2 etapas para el sistema de integración continúa implementado. Es decir, una etapa de verificación que conllevaría la compilación y ejecución de pruebas unitarias para cada cambio en el repositorio, y otra etapa de validación que ejecutase los casos funcionales agrupando varios cambios verificados previamente en el repositorio.

En el alcance del trabajo se mencionó también una tercera etapa de *testing* relacionada con el nivel de pruebas de sistema, que incluyen pruebas software no funcionales como pruebas de rendimiento, carga, estrés, estabilidad o fiabilidad, entre otras. Esta fase es mucho más compleja ya que intervienen muchos factores, pues no solo tiene en cuenta las características del producto software en sí, sino también cómo este interacciona con otros sistemas. Por tanto, los componentes hardware, así como los componentes de comunicación y del entorno, intervienen bastante en este tipo de pruebas, cuyo principal objetivo es el de la caracterización del producto. Este planteamiento está fuera del alcance del trabajo, sin embargo, presenta un interesante reto como línea de investigación en este sentido, que se detallará a continuación.

5.1 LÍNEAS FUTURAS DE INVESTIGACIÓN

Con la realización de este trabajo se proponen las siguientes líneas de investigación para la realización de futuros trabajos en el contexto de añadir mejoras a la metodología de la integración continua o, en el ámbito de esta mejora añadida al sistema inicial:

Una línea de investigación para mejorar el proceso de *nursing* implementado. Por una parte, añadiendo algoritmos que mejoren el resultado de la búsqueda de errores, bien mediante otros patrones de búsquedas, como la búsqueda binaria en lugar de secuencial o lineal, bien añadiendo otros algoritmos que reduzcan iteraciones no necesarias, con el objetivo de reducir el tiempo de ejecución. En este sentido se podría aplicar también técnicas para determinar y concretar los casos de prueba a ejecutar en función de los fallos obtenidos. De esta manera el proceso de *nursing* solo se centraría en los casos fallidos sin tener que ejecutar el resto. Además de todo ello, el proceso podría mejorarse significativamente si se añade técnicas de virtualización ligera mediante el uso de contenedores como, por ejemplo, *Dockers*[36]. De manera que se este proceso pueda paralelizar la comprobación de cada cambio en el intervalo del repositorio a evaluar, reduciendo el tiempo considerablemente.

[36] Más información en: https://www.docker.com/

Otra línea de investigación estaría enfocada en la realización del proceso de *nursing* para el nivel de pruebas del sistema. Como se comentó anteriormente, este nivel de pruebas es mucho más complejo e intervienen muchos factores, por tanto, requiere de la realización de procesos de *nursing* más elaborados. En muchas ocasiones ni siquiera sería factible su implantación, pues dicho proceso consumiría muchos recursos, y la mayoría de los fallos a nivel de componente y funcional, ya se habrían resuelto en los anteriores niveles. Sin embargo, en esta fase se pueden detectar fallos relacionados con un exceso de consumo de recursos injustificados que, de alguna manera, se han introducido por un mal diseño o una mala implementación de algún componente software, introducido en algún cambio del repositorio, desde que se hizo la última comparativa o se generó la última *baseline* óptima.

Esta línea de investigación podría abordar el caso de la detección de consumo de CPU injustificado si, de alguna manera, pudiese evaluar un impacto proporcional al obtenido en un sistema real mediante el uso de sistemas virtualizados. En ese caso, sí se podría implantar procesos de *nursing*, ya que el proceso de pruebas no consumiría tantos recursos como si de un sistema real se tratase. Está técnica no serviría para elaborar informes de caracterización, pues no serían valores reales, pero si determinaría los cambios que introducen impactos de consumo injustificados, tanto de memoria como de CPU, que posteriormente se verían en un sistema real.

Para finalizar, hay que destacar que la metodología de la integración continua, así como la entrega continua y despliegue continuo, han dado lugar a otras prácticas en ingeniería del software, basadas también en métodos ágiles, que tienen como objetivo la unificación entre el desarrollo software y sus operaciones, en el que la automatización y el monitoreo de los procesos son esenciales. Este es el caso de DevOps[37] *(Development and Operations*):

> *"DevOps es un conjunto de prácticas destinadas a reducir el tiempo entre el compromiso de un cambio en un sistema y el cambio que se coloca en la producción normal, al tiempo que garantiza una alta calidad"* [Bass, Weber, and Zhu 2015].

[37] Más información en: https://devops.com/

REFERENCIAS Y BIBLIOGRAFÍA

[Royce, 1970] Winston W. Royce. "Managing the Development of Large Software Systems". Proceedings of IEEE WESCON 26, pages 1–9, August 1970.

[Boehm, 1986] Barry W. Boehm. "A Spiral Model of Software Development and Enhancement", *ACM SIGSOFT Software Engineering Notes*, Volume 11, Issue 4, Pages 14-24, August 1986.

[Jacobson, 1999] Ivar Jacobson, Grady Booch & James Rumbaugh. *The Unified Software Development Process*: Addison-Wesley, 1999.

[Beck, 1999] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Beck, 2002] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002.

[Herbsleb and Grinter, 1999] Herbsleb, J. D. and Grinter, R. E, (1999), Spliting the organization and dintegrating de code: Conway's law revisited. In ICSE'99: *Proceding of the 21st international conference on Software engineering,* pages 85-95, New York, NY, USA. ACM.

[Fowler, 2006] Martin Fowler. (2006, May 1). Continuous Integration [Online]. Available: https://www.martinfowler.com/

[Richardson and Gwaltney, 2006] Richardson, Jared and Gwaltney, William. *Ship It! A Practical Guide to Successful Software Projects*: Pragmatic Bookshelf, 2006.

[Duvall, Matyas and Glover, 2007] Paul Duvall, Steve Matyas and Andrew Glover. *Continuous Integration, Improving Software Quality and Reducing Risk:* Addison-Wesley, 2007.

[Humble and Farley, 2011] Jez Humble, David Farley. Continuous Delivery. Addison-Wesley, 2011.

[Laster, 2017] Brent Laster. *Continuous Integration vs. Continuous Delivery vs. Continuous Deployment*: O'Reilly Media, 2017.

[PMI, 2005] PMI (2005). *A Guide to the Project Management Body of Knowledge.* Project Management Institute, third edition.

[Grady, 1999] An economic release decisión model: Insights into software Project management. In *In Proceedings of the Applicationes of Software Measurement Conference,* pages 227-239. Orange Park, FL: Software Quality Engineering.

[Olsson, 1999] Olsson, K. (1999). Daily build. The best of both worlds: Rapid development and control. Technical report, Swedish Eng. Industries.

[JUnit4 web page, 2019]. Official JUnit4 web page, 2019. Available online: https://junit.org/junit4/

[Maven web page, 2019]. Official Maven web page, 2019. Available online: https://maven.apache.org

[Jenkins web page, 2019]. Official Jenkins web page, 2019. Available online: https://jenkins.io

[Jenkins Wiki, 2019]. Official Jenkins wiki web page, 2019. Available online: https://wiki.jenkins.io

[JaCoCo web page, 2019]. Official Jacoco web page, 2019. Available online: https://www.jacoco.org/

[Continuous Delivery Foundation, CDF 2019]. The Linux Fundation, CDF 2019. Available online: https://cd.foundation/

[Gamma, Helm, Johnsosn and Vlissides, 1995] Erich Gamma, Richard Helm, Ralph Johnsosn, John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co; 1st edition (January 15, 1995).

[Bass, Weber, and Zhu 2015]. Len Bass, Ingo Weber and Liming Zhu, "*DevOps: A Software Architect's Perspective.*" ISBN 978-0134049847.

ACRÓNIMOS

- ✓ API (Application Programming Interfaze)
- ✓ CD (Continuous Delivery)
- ✓ CI (Continuous Integration)
- ✓ CPU (Control Process Unit)
- ✓ CT (Component Test)
- ✓ FT (Functional Test)
- ✓ GUI (Graphic User Interface)
- ✓ IDE (Integrated Development Environment)
- ✓ ISTQB (International Software Testing Qualifications Board)
- ✓ JDK (Java Development Kit)
- ✓ LFD (Last Functional Delivery)
- ✓ LSV (Last Stable Version)
- ✓ OMG (Object Management Group)
- ✓ OOP (Object Oriented Programming).
- ✓ PMI (Project Management Institute)
- ✓ QA (Quality Assurance)
- ✓ RFI (Ready for Integration)
- ✓ SDK (Software Development Kit)
- ✓ ST (System Test)
- ✓ TDD (Test Driven Development)
- ✓ UML (Unified Modeling Language)
- ✓ UP (Unified Process)
- ✓ UT (Unit Test)
- ✓ VCS (Version Control System)
- ✓ XP (Extreme Programing)

ANEXOS

CLASIFICACIÓN DE CASOS DE PRUEBA SEGÚN ISTQB

ISTQB[38] o Comité Internacional de Certificaciones de Pruebas de Software (International Software Testing Qualifications Board, ISTQB), es una organización mundial no lucrativa, que recoge todo el trabajo de cientos de expertos en pruebas software a nivel mundial, y realiza certificaciones en base a esa experiencia.

Realiza una clasificación de los casos de prueba, así como de los niveles de prueba, en función de sus objetivos, características y casos de uso.

Clasificación de los Niveles de Prueba:

- ✓ Nivel de Pruebas Unitarias. Suelen ser numerosas y relativamente pequeñas. Su objetivo es el de verificar que el código no contiene errores de tipo funcional y no funcional, que afectan a una parte de código concreta. Por lo que su corrección es muy directa, corregidos rápidamente por el desarrollador, ya que que el fallo está muy localizado. Representa el primer nivel de pruebas, con una ejecución muy rápida, generalmente asociada a la compilación del software, y garantizan básicamente la verificación software.

- ✓ Nivel de Pruebas de Componentes. Similar al nivel de pruebas Unitarias, pero con un grado de complejidad añadida, puesto engloba a varias unidades del producto software, grupadas en componentes generalmente en función de su funcionalidad. Su ejecución es relativamente mayor que el nivel anterior y garantizan la verificación software y parte de la validación.

[38] Más información en: https://www.istqb.org/

- ✓ Nivel de Pruebas de Integración. Estas pruebas permiten determinar el funcionamiento del producto software en el entorno para el que fue definido y como interactúa todos los componentes software que componen el producto. A diferencia del anterior nivel en el que se aislaba el funcionamiento por componente. Además, sirve para validar como este interactúa con otros sistemas de terceros, como sistemas operativos, sistemas de archivos, etc. Estas pruebas ser realizan tras la verificación del software y garantizan la validez del producto software. Pueden ejecutarse en sistemas virtualizados ya que aquí no se mide el rendimiento sino, más bien, que el sistema funcione correctamente y cumpla las expectativas especificadas.

- ✓ Nivel de pruebas del sistema. Ultimo nivel de pruebas que garantiza el óptimo funcionamiento de este en el entorno de producción. Se evalúa el comportamiento de este con otros sistemas con los que tiene que interactuar, como las comunicaciones y otros elementos periféricos. Aquí se evalúan otras cualidades del sistema como el rendimiento, la disponibilidad, la estabilidad, la capacidad, etc. Este nivel evalúa la caracterización del software en función de los requisitos definidos para el usuario.

Clasificación de las pruebas en función de su evaluación interna:

- ✓ Se consideran Pruebas de Caja Negra ("black-box testing"), aquellas que no es necesario conocer como se ha implementado el producto software. Se evalúa su comportamiento, no como está hecho el producto.

- ✓ Se consideran Pruebas de Caja Blanca ("white-box testing"), aquellas que si evalúan el contenido del producto software. Se evalúa como se ha implementado el producto, es decir, como se ha hecho.

Clasificación de tipos de pruebas atendiendo a su objetivo y finalidad:

- ✓ En primer lugar, la Pruebas Software Funcionales. Donde se define el comportamiento del sistema, subsistemas y componentes softwares especificados, mediante requisitos o casos de uso. Estas pruebas evalúan lo que debe hacer el producto software. Se pueden ejecutar en cualquier nivel de pruebas prácticamente, siendo más común en el nivel de pruebas de integración.

- ✓ En segundo lugar, las Pruebas Software no Funcionales. Normalmente asociadas con el nivel de pruebas de sistema, incluyen pruebas de Rendimiento, Carga, Estrés, Fiabilidad. Así como también pruebas de calidad y seguridad, como Mantenibilidad, Usabilidad, Portabilidad y Seguridad. Pueden ejecutarse en todos los niveles de prueba y, dado que se evalúa como interactúa el sistema con otros sistemas externos, estas pruebas suelen ser de caja negra.

- ✓ En tercer lugar, las Pruebas Software Estructurales. Su cometido es aplicar técnicas estáticas de análisis de código. Por tanto, aquí se evalúa la cobertura de código, la calidad del mismo, llamadas a métodos, mantenibilidad del código, y diferentes métricas de calidad en función de ciertas especificaciones o recomendaciones. Puesto que se evalúa el contenido interno del producto software, estas pruebas son de caja blanca.

- ✓ Por último, las Pruebas de Regresión. Estas evalúan que todos los componentes software de los que se compone en sistema entero, siguen funcionando tras añadir o modificar un nuevo componente del sistema. El criterio para decidir la extensión de estas Pruebas de Regresión está basado en el riesgo de encontrar o no, defectos en el software que anteriormente estaba funcionando correctamente.

GUÍA DE USO E INSTALACIÓN DE GIT

INSTALAR GIT EN UBUNTU 18.04

1º Actualizar el repositorio e instalar el paquete Git:

```
sudo apt-get update
apt-cache policy git
sudo apt-get install git
```

CREAR EL REPOSITORIO CENTRAL PARA EL PROYECTO

2º En el servidor, crear el repositorio:

```
mkdir /repo/git/
cd /repo/git/
mkdir maven_project.git/
cd repository.git
git init --bare --shared
git config core.sharedRepository true
```

CLONAR EL REPOSITORIO DEL PROYECTO EN LOCAL:

3º Para cada usuario que quiera clonar el repositorio en local:

```
mkdir /home/user/repo_local
cd /home/user/repo_local
git clone git+ssh://user@localhost/repo/git/maven_project.git
git config --global user.name "user"
git config --global user.email "user@uned.com"
git config --global core.editor "vi"
git init
```

COMANDOS ÚTILES DE GIT:

4º Algunos comandos para empezar a usar GIT:

```
git checkout master # Apuntar el repo local a la rama master
git pull --rebase # Actualizar el repo local y poner tus cambios encima
git log --oneline # Visualizar los cambios del repositorio resumidos
git commit -m "commit message" # Crear un commit con dicho mensaje
git stash # Abandonar tus cambios locales
git clean -xfd # Eliminar los cambios no "commiteados" en el repo local
git push origin master # Subir tus commits en local al repo de master
git reset --soft HEAD~n # Retroceder n commits en local sin eliminar
el contenido
git reset --hard HEAD~n # Retroceder n commits en local eliminando el
contenido
```

PROCESO DE INSTALACIÓN DE JENKINS

En primer lugar, es necesario tener instalado JAVA (JRE + JDK):

```
sudo apt update
apt install default-jre
sudo apt install default-jdk
sudo vi /etc/environment
      JAVA_HOME="/usr/lib/jvm/java-11-openjdk-amd64/bin/"
source /etc/environment
```

INSTALAR JENKINS EN UBUNTU 18.04

1º Agregar la clave del repositorio al sistema:

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -
```

2º Agreguar la dirección del repositorio de paquetes de Debian al fichero sources.list del servidor:

```
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
```

3º Actualizar el repositorio e instalar el paquete Jenkins:

```
sudo apt update
sudo apt install jenkins
```

CONFIGURAR JENKINS EN UBUNTU 18.04

4º Lanzar el servicio Jenkins y comprobar su estado. Comprobar Active:

```
sudo systemctl start jenkins
sudo systemctl status jenkins
   jenkins.service - LSB: Start Jenkins at boot time
   Loaded: loaded (/etc/init.d/jenkins; generated)
   Active: active (exited) since Mon 2018-07-09 17:22:08 UTC; 6min ago
   Docs: man:systemd-sysv-generator(8)
   Tasks: 0 (limit: 1153)
   CGroup: /system.slice/jenkins.service
```

5º Abrir el puerto 8080. Comprobar estado:

```
sudo ufw allow 8080
sudo ufw status
      Status: active
      To                         Action      From
      --                         ------      ----
      8080                       ALLOW       Anywhere
      8080 (v6)                  ALLOW       Anywhere (v6)
```

INSTALAR Y CONFIGURAR SSH

6º Descargar, instalar OpenSSH:

```
sudo apt update
sudo apt install openssh-server
sudo apt install openssh-client
```

CONFIGURAR OpenSSH:

7º Editar sshd_config, dependiendo de cada caso:

```
sudo cat /etc/ssh/sshd_config
     Port 22
     HostKey /etc/ssh/ssh_host_rsa_key
     PubkeyAuthentication yes
     IgnoreRhosts yes
     PermitEmptyPasswords yes
     ChallengeResponseAuthentication no
     UsePAM yes
     X11Forwarding yes
     PrintMotd no
     UseLogin no
     AcceptEnv LANG LC_*
      Subsystem       sftp    /usr/lib/openssh/sftp-server
```

8º Relanzar servicio SSH:

```
sudo systemctl enable ssh
sudo systemctl start ssh
```

9º Abrir el puerto que usa OpenSSH. Puerto 22:

```
sudo ufw allow OpenSSH
sudo ufw status
      Status: active
      To                         Action      From
      --                         ------      ----
      OpenSSH                    ALLOW       Anywhere
      8080                       ALLOW       Anywhere
      OpenSSH (v6)               ALLOW       Anywhere (v6)
      8080 (v6)                  ALLOW       Anywhere (v6)
```

10º Comprobar acceso web para Jenkins:

```
Go to http://localhost:8080
```

11º Copiar la contraseña de administrador en el formulario:

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

GoF COMMAND PATTERN TEMPLATE

Plantilla de uso general para implementar el patrón comando en Python:

Este patrón de comportamiento encapsula una solicitud como un objeto, lo que le permite parametrizar a los clientes las diferentes solicitudes, tanto solicitudes de cola como de registro, y admite operaciones que no se pueden deshacer.

```
import nursing
```

```
class Invoker:
    """
    Pide el comando para llevar a cabo la solicitud.
    """

    def __init__(self):
        self._commands = []

    def store_command(self, command):
        self._commands.append(command)

    def execute_commands(self):
        for command in self._commands:
            command.execute()
```

```
class Command(metaclass= nursing.NURSINGMeta):
    """
    Declara una interfaz para ejecutar una operación.
    """

    def __init__(self, receiver):
        self._receiver = receiver

    @nursing.abstractmethod
    def execute(self):
        pass
```

```
class ConcreteCommand(Command):
    """
    Define un enlace entre un objeto receptor y una acción. Implementa la ejecución
    invocando la (s) operación (es) correspondiente (s) en el receptor.
    """

    def execute(self):
        self._receiver.action()
```

```
class Receiver:
    """
    Realiza las operaciones asociadas a la realización de una solicitud.  Cualquier
    clase puede servir como un receptor.
    """

    def action(self):
        pass
```

```
def main():
    receiver = Receiver()
    concrete_command = ConcreteCommand(receiver)
    invoker = Invoker()
    invoker.store_command(concrete_command)
    invoker.execute_commands()

if __name__ == "__main__":
    main()
```

FICHEROS DE CONFIGURACIÓN USADOS EN EL PROYECTO

Jenkinsfile

```
#!/usr/bin/env groovy
pipeline {
      agent any
      environment {
          DISABLE_AUTH = 'true'
      }
      stages {
        stage ("Set Environment Stage"){
            steps {
            echo 'Setting Environment Stage:'
            sh 'id'
            sh 'uname -a'
            sh 'java -version'
            sh 'java -cp .:/usr/share/java/junit4.jar
org.junit.runner.JUnitCore | grep -v test | grep -v Time'
            }
        }
        stage("Compilation Stage") {
          steps {
            echo "Compilation Stage:"
            sh 'javac -cp .:/usr/share/java/*
./src/main/java/es/uned/ci/*.java ./src/test/java/es/uned/ci/*.java
-d .'
          }
        }
        stage("Unit Tests Stage") {
           steps {
            echo "UniT Test Stage:"
            sh 'java -cp .:/usr/share/java/junit4.jar
org.junit.runner.JUnitCore es.uned.ci.AeatTest es.uned.ci.NominaTest
es.uned.ci.RetencionTest'
            }
        }
        stage("Functional Tests Stage") {
          steps {
          echo "Functional Test Stage:"
          sh 'java es.uned.ci.RunFTest'
          }
        }
        stage("Deploiment Stage") {
          steps {
            echo "Deploiment Stage:"
            sh 'jar cvf RELEASE_`date +%s`-`cat build.release`.jar
es/uned/ci/*.class'
            sh 'mv *.jar
/opt/artifacts/maven_project/release_candidate/'
         }
      }
   }
}
```

Pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>es.uned.ci</groupId>
  <artifactId>${release.commit}</artifactId>
  <version>${release.build_id}</version>
<!-- Usage: mvn clean install package -Drelease.artifactId=$Release -
Drelease.version=$Version -Drelease.svm.version=$Revision
-->
  <name>maven_project</name>
  <url>https://www.uned.es</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <artifactId>maven-clean-plugin</artifactId>
          <version>3.1.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-resources-plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>2.22.1</version>
        </plugin>
        <plugin>
          <artifactId>maven-jar-plugin</artifactId>
          <version>3.0.2</version>
          <configuration>
<outputDirectory>/opt/artifacts/maven_project/snapshots/</outputDirectory>
          </configuration>
        </plugin>
        <plugin>
          <artifactId>maven-install-plugin</artifactId>
          <version>2.5.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-deploy-plugin</artifactId>
          <version>2.8.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-site-plugin</artifactId>
          <version>3.7.1</version>
        </plugin>
        <plugin>
          <artifactId>maven-project-info-reports-plugin</artifactId>
          <version>3.0.0</version>
        </plugin>
        <plugin>
        <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-release-plugin</artifactId>
          <version>2.4.2</version>
          <configuration>
           <tagNameFormat>v@{project.version}</tagNameFormat>
```

```
            <autoVersionSubmodules>true</autoVersionSubmodules>
            <releaseProfiles>releases</releaseProfiles>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
  <distributionManagement>
    <repository>
        <id>Artifactory_Releases</id>
        <name>Artifactory-releases</name>
        <url>scp://127.0.0.1/var/artifacts/mvn/releases</url>
    </repository>
    <snapshotRepository>
        <id>Artifactory_Snapshots</id>
        <name>Artifactory-snapshots</name>
        <url>scp://127.0.0.1/var/artifacts/mvn/snapshots</url>
    </snapshotRepository>
  </distributionManagement>
</project>
```

Jacoco.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>es.uned.ci</groupId>
  <artifactId>es.uned.ci.maven</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>
  <name>Jacoco statics coberage analisys</name>
  <url>http://www.jacoco.org/jacoco</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.10</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>
  <build>
    <plugins>
      <plugin>
        <groupId>org.jacoco</groupId>
        <artifactId>jacoco-maven-plugin</artifactId>
        <version>0.8.4</version>
        <executions>
          <execution>
            <id>default-prepare-agent</id>
            <goals>
              <goal>prepare-agent</goal>
            </goals>
          </execution>
          <execution>
            <id>default-report</id>
            <goals>
              <goal>report</goal>
            </goals>
          </execution>
          <execution>
            <id>default-check</id>
            <goals>
              <goal>check</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Build.version(versión inicial)

```
Version 1
Subversion 0
Revision 0
```