E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Desarrollo de un Proceso de Validación y Verificación Software

Grado en Ingeniería
en Tecnologías Industriales

Trabajo Fin de Grado

Laura Merino García

Tutor: Ángel María Andueza Unanua

Pamplona, 9 de junio de 2021

Resumen.

El proceso de validación y verificación software está presente en todos los proyectos en los que existe una parte hardware y software. Las empresas deben obtener metodologías efectivas para la gestión de esta etapa del ciclo de vida del proyecto software si quieren obtener software de calidad. La correcta gestión de los requisitos software que impone el cliente es una de las grandes dificultades de un proyecto. Además, posteriormente se debe realizar todo el proceso de testeo para demostrar al cliente el cumplimiento de los requerimientos especificados. Todo este proceso, realizado de forma manual, es tedioso, por lo que la búsqueda de herramientas, así como metodologías para la automatización del mismo, acaba siendo una tarea fundamental dentro de las empresas. Es por esto por lo que se planteará un nuevo proceso de Integración Continua, teniendo en cuenta los principios de las metodologías ágiles, que ayude a la empresa a agilizar la tarea de validación y verificación software.

Palabras clave: Validación y Verificación software, gestión de requisitos, Integración Continua, Jenkins, Robot Framework.

Abstrac

The software validation and verification process is present in all projects in which there is a hardware and software part. Companies must obtain effective methodologies for managing this stage of the software project life cycle if they want to obtain quality software. The correct management of the software requirements imposed by the client is one of the great difficulties of a project. In addition, later the entire testing process must be carried out to demonstrate to the client compliance with the specified requirements. All this process, carried out manually, is tedious, so the search for tools, as well as methodologies for its automation, ends up being a fundamental task within companies. This is why a new Continuous Integration process will be proposed, taking into account the principles of agile methodologies, which will help the company to speed up the software validation and verification task.

Keywords: Validation and verification software, requirements management, Continuous Integration, Jenkins, Robot Framework

Índice

1. Introducción..10
1.1. Área de trabajo. 10
1.2. Problemática. 11
1.3. Objetivos.. 13
1.3.1. Conocer el marco teórico sobre la calidad software. 13
1.3.2. Diseñar un proceso concreto de integración continua. 14
1.3.3. Demostrar el funcionamiento del proceso. 14
2. Marco teórico..15
2.1. Concepto de Calidad Software.. 15
2.2. Validación y Verificación. 15
2.3. Ciclo de vida de un proyecto software. 16
2.3.1. Modelo en V. 17
2.3.2. Metodologías Ágiles. 19
2.3.2.1. DevOps. 20
2.4. Integración continua (IC). 21
2.4.2. Pipeline. 23
2.5. Análisis del marco teórico. 24
3. Gestión de pruebas y requisitos. 26
3.1. Gestión de requisitos. 26
3.1.1. Clasificación de requisitos. 26
3.1.2. Ingeniería de requisitos. 27
3.2. Gestión de pruebas software 38
3.2.1. Tipos de pruebas software. 39
3.2.3. Creación del plan de pruebas. 44
4. Herramientas empleadas..46
4.1. Control de versiones. 46
4.2. Compilación de aplicaciones. 48
4.3. Pruebas software. 49
4.4. Integración continua 52
5. Diseño de un proceso automático de validación. 55
5.1. Diseño planteado. 55

5.1.1. Protocolo SSH. 57
5.2. Implementación del diseño. 58
5.2.1. Jira + Confluence. 58
5.2.2. Git + Bitbucket 61
5.2.3. Jenkins 64
6. Ejemplo de prueba y Resultados. 80
6.1. Requisito de la aplicación. 80
6.1.1. Ejemplo de prueba creado 81
6.1.2. Compilación de aplicación 81
6.1.3. Pruebas. 83
6.1.4. Creación de pruebas en Robot Framework. 84
6.2. Resultados del diseño. 87
6.3. Validación manual y comparación. 88
7. Conclusiones. 93
8. Líneas futuras. 95
9. Referencias. 97

Índice de figuras.

Ilustración 1. Logo NAITEC 10
Ilustración 2. Modelo en V. 18
Ilustración 3. Esquema DevOps. Fuente [9] 21
Ilustración 4. Esquema pipeline 23
Ilustración 5 Resumen proceso ingeniería de requisitos. 29
Ilustración 6. Tipos de pruebas software. 39
Ilustración 7. Proceso de Integración Continua. 46
Ilustración 8. Logo Git y Bitbucket. 47
Ilustración 9. Funcionamiento Git. 48
Ilustración 10. Logo Docker. 49
Ilustración 11. Logo Robot Framework. 51
Ilustración 13. Logo Jenkins. 52
Ilustración 14. Logo de Jira y Confluence. 54
Ilustración 15. Flujo proceso IC. 55
Ilustración 16. Funcionamiento llaves pública/privada por SSH. 57
Ilustración 17. Consola tras generación de claves. 58
Ilustración 18. Tablero proyecto Jira 59
Ilustración 19. EPIC de requisitos 60
Ilustración 20. Historia de un requisito Jira 60
Ilustración 21. Ejemplo repositorio local. 62
Ilustración 22. Aspecto repositorio. 63
Ilustración 23. Flujo de ramas del repositorio. Fuente [18] 64
Ilustración 24. Contraseña de aplicación de Bitbucket. 66
Ilustración 25. Credencial tipo contraseña en Jenkins. 66
Ilustración 26. configuración clave privada SSH. 67
Ilustración 27. Configuración Docker en Jenkins 68
Ilustración 28 Proyecto Jenkins 69
Ilustración 30. Configuración general Jenkins. 70
Ilustración 31. Configuración Trigger Jenkins. 71
Ilustración 32. Workspace de Jenkins. 74
Ilustración 33. Seleccionar repositorio Jenkins. 74
Ilustración 34. Interfaz RIDE 85
Ilustración 35. Log de prueba Robot Framework 87
Ilustración 36. Reporte prueba Robot Framework 87
Ilustración 37. Menú principal de una ejecución de canalización. 88
Ilustración 38. Flujo de validación test de partida. 89

Índice de Tablas

Tabla 1. Desventajas modelo v y metodología ágil 24
Tabla 2. Ventajas modelo en V y Metodología ágil 25
Tabla 3. Tipos de Requisitos. 27
Tabla 4. Cuestiones para recolección de requisitos 29
Tabla 5. Maneras de generar requisitos. 31
Tabla 6. Tipos de conflictos en Requisitos. 33
Tabla 7. Análisis de calidad documento requisitos. 37
Tabla 8. Análisis calidad documento de pruebas. 45
Tabla 9. Características Robot Framework 51
Tabla 10. Comparación tipos de proyectos en Jenkins. 69
Tabla 11. Formato para escribir Pipelines. 73
Tabla 12. Requisitos de la aplicación. 80
Tabla 13. Test de requisitos. 83
Tabla 14. Características test. 85
Tabla 15. Comparación de Resultado 92

1. Introducción.

La complejidad de los diseños software es algo que está creciendo en los últimos años de forma imparable. Los clientes piden software más sofisticado y con menos errores.

El proceso de validación y verificación es una parte crucial de la vida de un proyecto software y acaba ocupando, aproximadamente, dos terceras partes del tiempo que se dedica al total. Esto hace crucial el estudio de nuevas técnicas que sean capaces de agilizar el proceso de validación y conseguir así, una mejora exponencial de la productividad y la calidad de los diseños.

Para lograr un buen proceso de calidad, se hace necesario plantearse la Calidad del Software desde una perspectiva global, albergando el ciclo de vida completo del producto, teniendo en cuenta tanto la calidad final del mismo como la del proceso de desarrollo. Así, contar con metodologías, estándares, buenas prácticas, normativas y directrices correctas resulta fundamental para construir el armazón que permite desarrollar y poner en producción un producto software de calidad porque de esta forma, los errores y los costes serán menores y la agilidad y eficacia crecerán. Es decir, no basta con probar y probar, sino con probar con criterio. Cuando se une un buen proceso con un buen producto, el éxito está prácticamente asegurado.

1.1. Área de trabajo.

La metodología a desarrollar se va a implementar en NAITEC, empresa navarra dedicada a la investigación en diferentes ámbitos de la automoción y la mecatrónica desde 1989, en ese momento conocida como CETENASA.

ILUSTRACIÓN 1. LOGO NAITEC

En concreto, se trabajó en el área de Movilidad de la empresa, dedicada a la mejora de la seguridad y el confort en vehículos, siempre desde un punto de vista sostenible. Sus principales líneas de trabajo actualmente son el vehículo autónomo y conectado y la movilidad sostenible.

Los desarrollos en esta área de trabajo están completamente ligados con la creación de productos software de alta calidad lo que implica procesos de validación costosos.

1.2. Problemática.

Las dificultades asociadas a una validación ineficiente pusieron de manifiesto la necesidad de comenzar a plantear un cambio en las estrategias existentes en la organización. Esta es una problemática global, toda empresa dedicada al diseño software/hardware la ha experimentado en algún punto de su vida.

El problema que una validación manual ineficiente provoca, en términos de tiempo, puede ser fácilmente representado mediante un ejemplo práctico, basado en [1], sobre la experiencia de testeo en software embebido para así poder comprender la magnitud del problema a afrontar:

El diseño contaba con un total de 500 ciclos de test de regresión donde, por las características de los test automáticos, llevaban un total de 3 horas para realizar un test. Por lo cual, el tiempo por ciclo de validación era:

Tiempo de test automáticos: $500 * 3 = \boldsymbol{1500\ horas\ de\ testeo\ autmático}$.

Por otro lado, una hora de testeo automático equivalía, para realizar los mismos test y por lo cual llegar a la misma calidad de diseño, a 4 días de trabajo. Esto hace que el total de tiempo que sería necesario para la validación sea de:

$$500\ test * 2\ semanas\ de\ trabajo = \boldsymbol{19,5\ años\ de\ testeo\ por\ un\ desarrollador}!!$$

Además de ello, las pruebas manuales llevan asociadas otra serie de dificultades, las cuales serán desarrolladas a lo largo del presente trabajo, lo que las hace completamente ineficientes.

Pero entonces, ¿qué hace a los desarrolladores renegar la combinación de software embebido y nuevas metodologías de desarrollo ágil como DevOps (metodología de desarrollo de proyectos software que posteriormente será explicada) y procesos de validación automático? Para ello se deben considerar los grandes cambios que esto supone, basado en [2].

Testeo: Las pruebas automatizadas son una característica esencial de los nuevos métodos de validación. El gran desafío con las pruebas de software integrado es la falta de acceso a entornos similares a la producción. ¿Cómo se pueden realizar lanzamientos continuos a la producción sin un control de calidad riguroso? ¿Hay alguna manera de incluir el hardware en el bucle de pruebas?

Implementación: El mundo de los embebidos se ocupa con frecuencia de implementaciones personalizadas. Este es un peligro potencial porque, a diferencia del software de aplicación, el entorno en vivo es el mundo físico el cual es aleatorio y cambiante. ¿Cómo pueden las pruebas automatizadas controlar o simular los entornos reales de prueba? ¿Hay seguridad para afirmar que las pruebas han sido suficientes si no se prueba en entornos reales?

Seguridad: Probar software integrado es especialmente delicado porque se realiza en sistemas que son críticos para la seguridad como el control de coches. Los errores en este tipo de aplicaciones tan delicadas no son una opción. ¿Cambiar a una nueva metodología, desconocida para los desarrolladores, capaz de validar un diseño de forma automática pone en riesgo la seguridad de los diseños?

Tiempo: Adaptar nuevas metodologías suele ser rechazado por muchos desarrolladores que han trabajado de forma sistemática en la validación software de forma manual obteniendo "buenos" resultados. ¿El tiempo dedicado a adaptar esta metodología se verá reflejado en la calidad de los resultados? ¿Aparecerán nuevos problemas que serán complejos de solucionar y ocuparán una gran cantidad de tiempo?

En vista de esto, ¿son los sistemas embebidos, así como otro tipo de sistemas, incompatibles con una nueva metodología de validación? El desarrollo de software integrado se ocupa de problemas de seguridad, implementaciones personalizadas y acceso limitado a entornos similares a la producción. Por el contrario, las metodologías ágiles de validación son una forma de crear software ágil, colaborativo y automatizado. Esta aparente incompatibilidad es la razón por la que métodos de integración continua, así como metodologías ágiles como DevOps, se han descartado con frecuencia como un método para embebidos. Sin embargo, se demostrará como la simple adopción de ideas básicas puede traer consigo grandes beneficios.

1.3. Objetivos.

En vista de la problemática y el área donde se trabará, el propósito principal del trabajo es la construcción de una metodología que sirva como base para comenzar a desarrollar una nueva forma de validar y verificar software de forma eficiente y segura.

Para concretar, los objetivos específicos del proyecto son los siguientes:

1.3.1. Conocer el marco teórico sobre la calidad software.

Conocer la teoría sobre el tema a tratar es fundamental. Los programadores y demás responsables de proyectos son los que deben incorporar todas estas bases teóricas a su modo de trabajo para que la metodología se efectiva. Si se conoce lo que se debe hacer, pero no el por qué, es muy probable que tarde o temprano algo acabe fallando.

Por ello, se realizará una breve explicación de los conceptos de mayor importancia dentro de la calidad software, enfocándolos a empresas de pequeño tamaño y concluyendo con las ideas básicas del tema y una discusión.

1.3.2. Diseñar un proceso concreto de integración continua.

En vista de la teoría general sobre el tema, se diseñará a lo largo del trabajo una metodología completa, desde la recolección de los primeros requisitos hasta los reportes de las pruebas que los validan, pasando por la correcta gestión de requerimientos y casos de prueba.

1.3.3. Demostrar el funcionamiento del proceso.

Con el fin de facilitar el entendimiento del proceso y demostrar su funcionamiento, se realizará una demo del diseño generado que además servirá de guía para comenzar a crear nuevos proyectos de validación.

2. Marco teórico

En este apartado se realizará una explicación teórica de todos los conceptos principales que cualquier desarrollador de producto software debe conocer si quiere mejorar en el ámbito general de la validación y verificación software.

2.1. Concepto de Calidad Software

La International Standards Organization, ISO en la norma 8402:1994, define calidad como la *"Totalidad de propiedades y características de un producto, proceso o servicio que le confiere su aptitud para satisfacer unas necesidades expresadas o implícitas."*[3]. En la actualización de la Norma ISO, la 9000:2000, la definición quedó *"Grado en el que un conjunto de características inherentes cumple con los requisitos*". En esta definición se hace especial énfasis en cumplir los requerimientos de los consumidores [4].

El control de la calidad en un producto software lo realiza el propio desarrollador, el cual dispone por lo general de poco tiempo. La calidad software se inicia desde la planificación del proyecto y es una cualidad que debe darse a los largo de todo el desarrollo marcando el conjunto de procedimientos, herramientas y técnicas que se emplearán durante el ciclo de vida. Así, la calidad se entiende como el conjunto de todas las actividades del proyecto. La calidad total depende de la calidad con las que se realicen las fases del trabajo.

Entendiendo la calidad software como un todo, es más fácil poner de manifiesto la necesidad de implementar una metodología trazable de principio a fin para conseguir las características de calidad tanto en desarrollo como en producto.

2.2. Validación y Verificación.

La verificación y validación es el nombre que se da a los procesos de comprobación y análisis que asegura que el software que se desarrolla está acorde a su especificación y cumple las necesidades de los clientes. La V&V es un proceso de ciclo de vida completo. Comienza

con las revisiones de los requisitos, continúa con las revisiones del diseño y las inspecciones de código y acaba en las pruebas de validación.

Validar y verificar no son la misma cosa. Según Boehm [5] las diferencias entre ellas son:

- **Verificación**: ¿Estamos construyendo el producto correctamente? El papel de la verificación comprende comprobar que el software está de acuerdo con su especificación. Se comprueba que el sistema cumple los requerimientos funcionales y no funcionales que se le han especificado.
- **Validación**: ¿Estamos construyendo el producto concreto? La validación es un proceso más general. Se debe asegurar que el software cumple las expectativas del cliente. Va más allá de comprobar si el sistema está acorde con su especificación, para probar que el software hace lo que el usuario espera a diferencia de lo que se ha especificado.

2.3. Ciclo de vida de un proyecto software.

El ciclo de vida del desarrollo software (también conocido como SDLC o Systems Development Life Cycle) se refiere a una metodología con procesos perfectamente definidos para crear software de alta calidad.

La normativa ISO/IEC/IEEE 12207:2017 define ciclo de vida software como: *"un marco común para los procesos del ciclo de vida de los programas informáticos, con una terminología bien definida, a la que pueda remitirse la industria del software. Contiene procesos, actividades y tareas aplicables durante la adquisición, el suministro, el desarrollo, el funcionamiento, el mantenimiento o la eliminación de sistemas, productos y servicios informáticos. Estos procesos del ciclo de vida se llevan a cabo mediante la participación de los interesados, con el objetivo final de lograr la satisfacción del cliente". [6]*

Su origen radica en lo costoso de rectificar los posibles errores que se detectan tarde en la fase de implementación. Utilizando metodologías apropiadas, se podría detectar a tiempo para que los programadores puedan centrarse en la calidad y seguridad del software, cumpliendo los plazos y los costes asociados.

Las metodologías para el desarrollo de software son un modo sistemático de realizar, gestionar y administrar el ciclo de vida para llevarlo a cabo con grandes posibilidades de éxito. Esta sistematización indica cómo se divide un proyecto en partes más pequeñas para normalizar cómo se gestiona el mismo, estas partes son las fases de desarrollo.

Los pasos a seguir pueden organizarse e interactuar entre ellos de diversas formas, de estas interacciones surgen los modelos de proyectos software. Entre ellos destacan el Modelo en V y las metodologías ágiles.

2.3.1. Modelo en V.

Una evolución del modelo en cascada que demuestra como las fases de prueba se relacionan con las de análisis y desarrollo. Este modelo pone de manifiesto como el desarrollo de las pruebas debe efectuarse de manera paralela al desarrollo del programa. En contraste, un modelo clásico centra la atención en lo que se produce, mientras el modelo en V lo hace en la actividad. Es un modelo ideal para proyectos pequeños por su robustez, claridad y sencillez.

Se divide en 4 niveles lógicos. Para cada fase de desarrollo existe una fase correspondiente o paralela de verificación y validación. Este modelo obliga entonces a que por cada fase de desarrollo exista un resultado que se deba verificar.

En la misma estructura que da nombre al modelo, la proximidad entre una fase del desarrollo y su fase de verificación correspondiente hace referencia al tiempo entre una fase y su homólogo de verificación.

El modelo entonces se divide en los siguientes 4 niveles:

- El nivel 1 orientado al "cliente". Indican el inicio y el fin del proyecto. Se compone del análisis de requisitos que proporciona el cliente los cuales se traducen en un documento de requisitos y especificaciones.

- El nivel 2 se dedica a las características funcionales del sistema. El sistema se interpreta como una caja negra caracterizando aquellas funciones que son directa o indirectamente visibles para el usuario.

- El nivel 3 define los componentes hardware y software del sistema final, a cuyo conjunto se denomina arquitectura del sistema. Se definen aquellos elementos que son necesarios para cumplir con las funcionalidades del nivel 2.

- El nivel 4 es la fase de implementación, en la que se desarrollan los elementos unitarios o módulos del programa. Una vez definidos los componentes este es el punto en el que el desarrollador comienza a crear la aplicación.

Cada uno de estos niveles va acompañado con su proceso de validación como se indica en la siguiente figura.

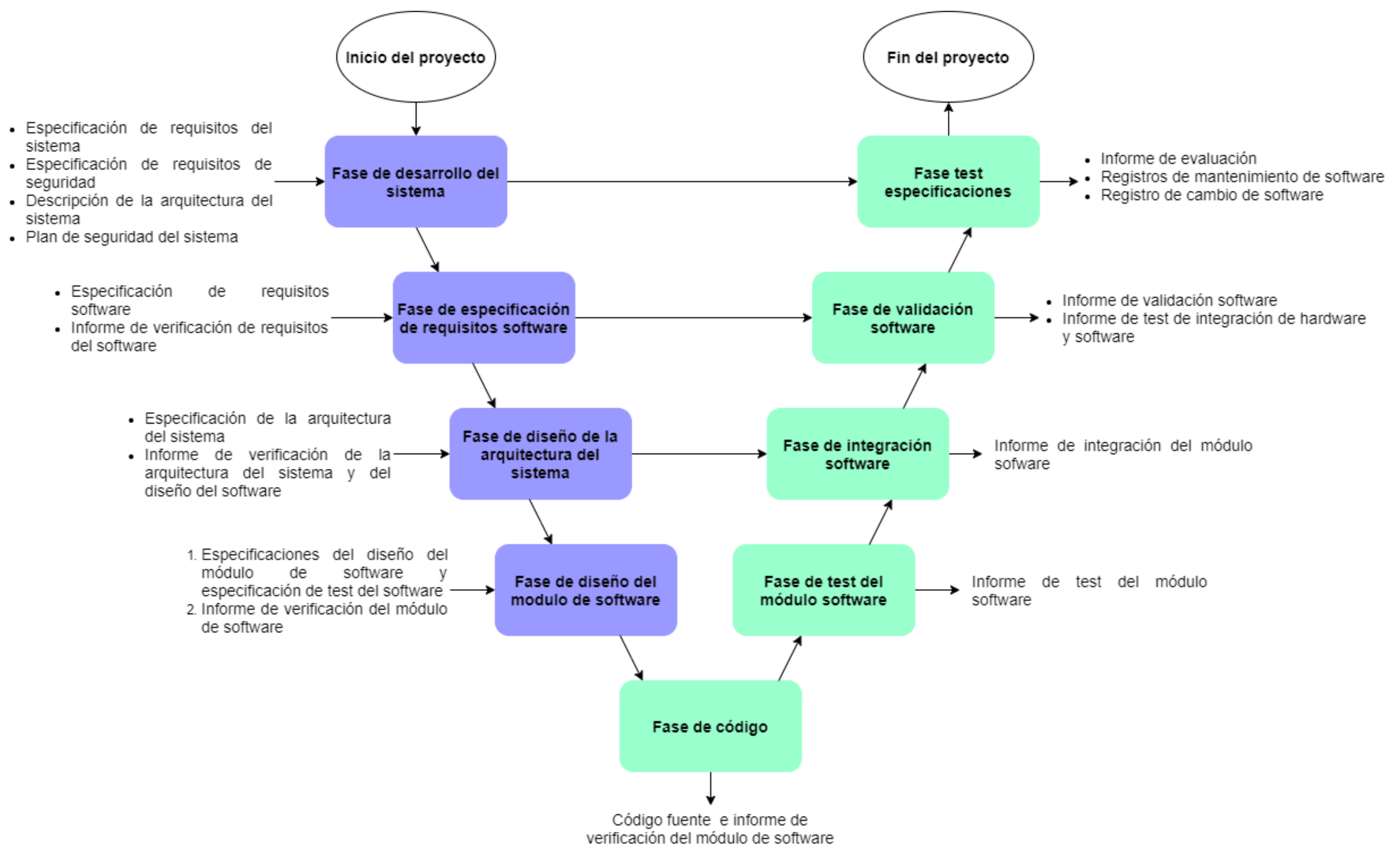


ILUSTRACIÓN 2. MODELO EN V.

Este modelo pone de manifiesto el orden cronológico que deben seguir las fases de un proyecto, impidiendo descender a nivel siguiente si dicho nivel no ha sido completado en la rama de desarrollo. Tampoco permite ascender en la rama de prueba si la validación anterior no ha sido completada.

2.3.2. Metodologías Ágiles.

Las metodologías ágiles son filosofías de trabajo que engloban un conjunto de métodos que permite adaptar el modo de trabajo a las condiciones del proyecto, aportando flexibilidad y eficiencia.

En el año 2001, se reunieron importantes desarrolladores de software y pusieron sus conocimientos en común sobre los mejores métodos de desarrollo creando el Manifiesto Ágil, en el que se establecieron 12 principios, y cuya filosofía se puede resumir en cuatro ideas principales, según [7]:

1) **Valorar a los individuos e interacciones sobre los procesos y herramientas.**

Los procesos y las herramientas son muy importantes, proporcionan métodos para ejecutar el proceso con eficiencia. Sin embargo, estas características pierden todo su valor si no van acompañadas de buenos desarrolladores que son los responsables últimos de realizar las tareas, y cuyo conocimiento y talento serán clave para lograr buenos resultados. Los procesos deben ser una ayuda y una guía, deben adaptarse a la organización y al equipo, y no al revés. Además, no pueden ser rígidos, ya que la sociedad cambia rápidamente, y los profesionales deben poder hacer propuestas innovadoras para agilizar procesos.

2) **Valorar el software funcionando sobre la documentación extensiva.**

El valor que aporta un software en funcionamiento nunca podrá ser superado por la documentación que se haya podido realizar previamente. Durante el proceso de desarrollo es importante ir comprobando las funcionalidades que va adquiriendo el proyecto, tal y como se lleva a cabo en la fase de prototipado de la metodología Design Thinking. Ver anticipadamente el comportamiento de las funcionalidades diseñadas y poder interactuar con los prototipos enriquece el proceso y ofrece la oportunidad de generar nuevas ideas.

La documentación es importante y en muchos casos necesaria, pero nunca debe convertirse en una carga que frene la agilidad de los procesos de producción ni en una barrera para la comunicación directa entre las personas del equipo.

3) Valorar la colaboración con el cliente sobre la negociación contractual.

En un mercado cambiante y en procesos de desarrollo prolongados, la relación del equipo con el cliente no puede limitarse a un contrato, que no puede ser mucho más que una formalidad, sino que el propio cliente debe ser un miembro más del equipo, con una colaboración activa. De esta forma, el proyecto se enriquecerá con el conocimiento y la experiencia del cliente y los posibles cambios que se requieran se llevarán a cabo con mucha mayor rapidez.

La comunicación continua y la flexibilidad del proceso permiten alcanzar un producto de mayor calidad de la forma más eficiente y mejoran la satisfacción del cliente, que habrá estado presente durante todo el proceso.

4) Valorar la respuesta ante el cambio en lugar de seguir un plan.

En el método de trabajo tradicional se valora en gran medida una buena planificación previa y herramientas de control para no desviarse del plan. Este sistema deja de ser adecuado cuando se opera en un entorno cambiante e inestable, en el que hay una evolución rápida y continuada. Las metodologías ágiles valoran más la capacidad de respuesta y una gestión flexible con capacidad de anticipación y adaptación al cambio.

Estas nuevas filosofías ponen de manifiesto una nueva forma de trabajo que ha ido cogiendo fuerza en la gestión de proyectos generando submetodologías, donde destaca DevOps.

2.3.2.1. DevOps.

DevOps surge como un acrónimo del inglés de development (desarrollo) y operations (operaciones). Es una práctica de ingeniería de software que tiene como objetivo unir el desarrollo con la operación software. La principal característica de DevOps es la automatización y el control de todos los pasos de construcción software desde la integración, las pruebas… hasta la implementación. [8].

El proyecto sigue una constante realimentación, como ilustra la imagen 3. Esto se representa como un bucle infinito donde los pasos dentro del ciclo de vida de un proyecto (planificar, construir, implementar, comunicar…) se dan a lo largo de todo el tiempo alimentándose unos de los otros.

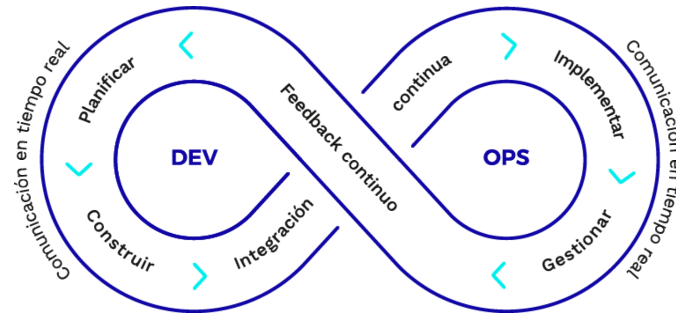


ILUSTRACIÓN 3. ESQUEMA DEVOPS. FUENTE [9]

Ya no se sigue una gestión lineal y la planificación de trabajo realizada va modificándose constantemente a medida que avanza el proyecto.

2.4. Integración continua (IC).

Según Martin Fowler, precursor del concepto de Integración Continua *"La integración continua es una práctica de desarrollo de software en la cual los miembros de un equipo integran su trabajo frecuentemente, como mínimo de forma diaria. Cada integración se verifica mediante una herramienta de construcción automática para detectar los errores de integración tan pronto como sea posible. Muchos equipos creen que este enfoque lleva a una reducción significativa de los problemas de integración y permite a un equipo desarrollar software cohesivo de forma más rápida."*[10].

La integración continua se ha convertido en una práctica necesaria en el desarrollo software por los beneficios que ofrece, donde destacan: reducción de riesgos y de tareas repetitivas y generación de software listo para desplegar. Mediante esta técnica, se monitorea el sistema de control de versiones y cuando es detectado un cambio, es posible que una herramienta compile y pruebe automáticamente los cambios detectados y notifique de inmediato a los involucrados de los posibles fallos.

2.4.1. Prácticas en un proceso de Integración Continua.

Los procesos de integración continua son capaces de mejorar enormemente la gestión de código validando de forma automática y continua. Es por esto por lo que se definen una serie de ejercicios a seguir para asegurar la eficiencia del método, según [11].

- Enviar cambios con frecuencia al repositorio para iniciar el ciclo de validación automática. Para conseguir que esto se haga con periodicidad y control los cambios se hacen de modo que solo se modifique un componente o elemento a la vez, probando además si ese cambio ha desencadenado más errores que los que ha solucionado.

- No enviar cambios cuando hay errores. Si el proceso de construcción falla, no se debe hacer copia del último código en el repositorio.

- Procurar que los procesos sean rápidos. Eliminar las pruebas de bajo valor o con conclusiones indeterminadas para mantener la velocidad del proceso.

- Aislar y proteger el entorno de IC. El sistema IC tiene acceso completo al código y credenciales para la implementación en varios entornos.

- Conseguir que el proceso de IC sea la única forma de realizar implementaciones en la producción. La promoción del código a través de sus procesos de IC requiere que cada cambio demuestre que cumple con los estándares y procedimientos codificados de su organización. Las fallas en un proceso de IC se pueden ver de inmediato y detienen el avance de la versión afectada hasta etapas posteriores del ciclo. Este es un mecanismo de control de acceso que protege los entornos más importantes contra códigos no confiables.

- Mantener la paridad con la producción siempre que sea posible.

- Hacer el trabajo de compilación solo una vez y transmitir el resultado a través del proceso. Un objetivo principal de un proceso de IC es crear confianza en sus

cambios y minimizar la posibilidad de un impacto inesperado. Si el software requiere un paso de construcción, empaquetado o agrupación, ese paso se debe ejecutar solo una vez y el resultado obtenido se debe reutilizar a lo largo de todo el proceso.

2.4.2. Pipeline.

La integración continua actual pasa por pipelines. Un pipeline es una nueva forma de trabajar dentro del mundo de DevOps y la integración continua. Las pipelines o canalizaciones son una forma de organizar los pasos dentro de la automatización de pruebas y, en definitiva, de la integración continua de código basándose en los conceptos claves de las metodologías ágiles.

Mediante las pipelines resulta más sencillo replicar los diferentes pasos con distintas aplicaciones y gestionar mejor los cambios de cada paso. El propósito general de estas canalizaciones es automatizar grandes franjas del ciclo de vida del desarrollo del producto.

Además, las pipelines se pueden enfocar, no solo como un conjunto de procesos capaz de automatizar tareas, si no como una forma de materializar el flujo de DevOps en tareas, ahora específicas de un proyecto, que los desarrolladores deben seguir para conseguir llevar a cabo el trabajo de forma rigurosa.

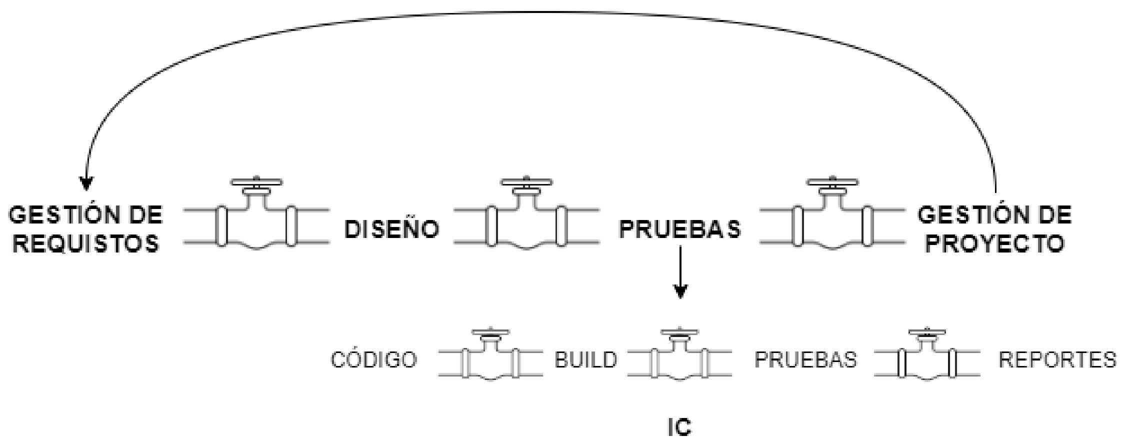


ILUSTRACIÓN 4. ESQUEMA PIPELINE

La canalización irá más allá de lo propiamente destinado a la fase de validación y podrá marca una metodología trazable de principio a fin que comience por el requisito a validar, paso por el proceso de validación y finalice en la gestión del proyecto tras los resultados, dando pasos claros en cada etapa que el desarrollador deberá seguir.

2.5. Análisis del marco teórico.

Llegado este punto ¿cómo se pueden aplicar estas conceptos en un proyecto real? Muchos de los conceptos tratados son en ocasiones cuestiones filosóficas por lo que adaptar esto a la realidad presenta una serie de incertidumbre que dificultan la puesta en marcha de las metodologías en las empresas. Es por esto por lo que ¿es correcto optar por una metodología para implementarla al 100% en un proyecto? Tanto el modelo en V como las metodologías ágiles presentan ciertas desventajas:

Modelo en V	Metodología Ágil
El desarrollo de los proyectos está centrado en la creación de documentos. Puede ocurrir que se dedique demasiado tiempo en la gestión de estos y el diseño del producto acabe en un segundo lugar, donde verdaderamente lo que el cliente espera es el diseño en funcionamiento.	Como la metodología ágil se centra en el software y no en la documentación, pueden surgir problemas de mala comunicación en cuestiones técnicas tanto entre cliente y desarrollador como entre los propios desarrolladores por documentación mal especificada.
No son modelos adecuados para proyectos con requisitos que cambian constantemente.	Los proyectos comienzan si planificación concreta, es difícil determinar la cantidad de tiempo y dinero que será necesario debido a los requisitos en constante cambio.
Se centra más en la gestión del proyecto y no en el desarrollo.	Es necesario un alto nivel de interacción entre cliente y desarrolladores.
La estructura rígida permite una respuesta poco flexible a los cambios, promoviendo el transcurso lineal del proyecto.	Si el desarrollo del proyecto comienza a fallar, es más fácil que acabe sucumbiendo al no contar con un plan perfectamente definido de desarrollo.
Por su rigidez, es posible que el producto final no cumpla con los requisitos del usuario.	

TABLA 1. DESVENTAJAS MODELO V Y METODOLOGÍA ÁGIL

Estas metodologías imponen ideas de trabajo opuestos en la forma de gestionar los proyectos software. Pero como se ha comentado, muchos de estos conceptos son cuestiones filosóficas de trabajo. En numerosas ocasiones será posible que el modelo en V se utilice como base para una metodología ágil, consiguiendo extraer los puntos positivos de cada uno.

Ventajas Modelo en V + Metodologías Ágiles
Mantener una documentación correcta y completa pero flexible. Comprender la importancia de toda la documentación que se genera pero dando cabida a nuevos cambios en todo momento.
Las pruebas se realizan a lo largo de todo el proyecto, incluso de forma diaria como implica la adopción de métodos de integración continua, pero siguiendo el orden de pruebas que marca el modelo en V, pasando de lo particular a lo general.
El proyecto comienza con un plan marcado y será susceptible de cambios, viendo estos cambios como una forma de canalizar el trabajo y no como obstáculos de la producción, como ocurre con modelos más rígidos.
La comunicación se convierte en algo imprescindible. Todos los cambios deben estar visibles para todos los responsables pero a su vez correctamente recogidos, no solo basta con una comunicación oral entre desarrolladores.
Los clientes forman parte del desarrollo para llegar a acuerdos lógicos de diseño.
Las pruebas software permiten ahora una automatización completa gracias al uso de las canalizaciones de testeo.

TABLA 2. VENTAJAS MODELO EN V Y METODOLOGÍA ÁGIL

Tras esto, comienza el punto de generar una metodología que sirva como base para mejorar el proceso de validación y verificación software, partiendo de la misma gestión de documentación de requisitos y pruebas.

3. Gestión de pruebas y requisitos.

Existen ciertos problemas que ocurren en el día a día que dificultan el proceso de gestión de requisitos de un nuevo diseño tales como: clientes indecisos, requisitos mal especificados, documentos complejos de entender y poco colaborativos, nuevos requisitos inesperados… es por esto por lo que generar unas pautas claras para la recolección de requisitos, gestión y validación es imprescindible para comenzar a desarrollar un proyecto de calidad.

3.1. Gestión de requisitos.

Los requisitos de un proyecto software son las diferentes funcionalidades, cualidades... que debe cumplir el sistema. Estos son exigidos por el cliente. Por lo general, el cliente no tiene conocimientos técnicos suficientes, por lo que la comunicación entre desarrolladores y cliente puede ser insuficiente. Esto mismo puede ocurrir entre participantes del mismo equipo de proyecto donde una correcta comunicación es fundamental. Marcar una metodología basada en buenas prácticas que ayude tanto a cliente como a desarrolladores a facilitar la gestión de requisitos de un diseño será el primer paso hacia la calidad del producto.

3.1.1. Clasificación de requisitos.

Todo comienza por la definición y caracterización de los diferentes tipos de requisitos software y hardware.

TIPOS DE REQUISITOS		DESCRIPCIÓN
No funcionales	DE NEGOCIO	Dan una descripción a alto nivel de lo que elsistema debe hacer. Están conformados por los objetivos que el cliente espera de su propuesta, así como puede ser el alcance y el valor esperado del sistema.

	DE USUARIO	Describen el comportamiento que tendrá el sistema cuando el usuario interactúe con él. Se enfocan en las funcionalidades del diseño.
Funcionales	DEL SISTEMA/SOFTWARE	Definen las funcionalidades y características que debe tener el sistema para satisfacer tanto los requisitos de usuario. Son los requisitos que imponen las características y arquitectura del sistema. También marcan los planes de prueba.
	RESTRICCIONES	Son las características que limitarán ciertas decisiones de diseño y de funcionamiento.

TABLA 3. TIPOS DE REQUISITOS.

Los requisitos funcionales definen una función del sistema software o sus componentes. Se describen como un conjunto de entradas, comportamientos y salidas. Estos requisitos deben ser: cálculos, detalles técnicos, manipulación de datos y otras funcionalidades específicas que el sistema debe cumplir. Establecen el comportamiento del software.

Por otro lado, se encuentran los requisitos no funcionales. Se refieren a todos los requisitos que describen características de funcionamiento, sin especificar qué lo hará o como lo hará.

3.1.2. Ingeniería de requisitos.

En la gestión de requerimientos es fundamental aplicar una buena ingeniería de requisitos. La ingeniería de requisitos comprende todas las tareas relacionadas con la determinación de las necesidades o de las condiciones a satisfacer para un software nuevo o modificado, tomando en cuenta los diversos requisitos de las partes interesadas, que pueden entrar en conflicto entre ellos. Sin embargo, la Ingeniería de requerimientos también es contemplada en otras disciplinas, estando fuertemente vinculada con la administración de proyectos. [12]

La ingeniería de requisitos marca 5 pasos a cumplir en la gestión de esta fase [12]:

1. Reconocimiento del problema. Se deben de estudiar inicialmente las especificaciones del sistema y el plan del proyecto del software. Realmente se necesita llegar a comprender el software dentro del contexto del sistema. El analista

debe establecer un canal adecuado de comunicación con el equipo de trabajo involucrado en el proyecto. En esta etapa la función primordial del analista en todo momento es reconocer los elementos del problema tal y como los percibe el usuario.

2. Evaluación y síntesis. En esta etapa el analista debe centrarse en el flujo y estructura de la información, definir las funciones del software, determinar los factores que afectan el desarrollo de nuestro sistema, establecer las características de la interfaz del sistema y descubrir las restricciones del diseño. Todas las tareas anteriores conducen fácilmente a la determinación del problema de forma sintetizada.
3. Modelización Durante. la evaluación y síntesis de la solución, se crean modelos del sistema que servirán al analista para comprender mejor el proceso funcional, operativo y de contenido de la información. El modelo Ingeniería de Requerimientos servirá de pilar para el diseño del software y como base para la creación de una especificación del software.
4. Especificación. Las tareas asociadas con la especificación intenta proporcionar una representación del software. Esto más adelante permitirá llegar a determinar si se ha llegado a comprender el software, en los casos que se lleguen a modelar se pueden dejar plasmados manuales.
5. Revisión. Una vez que se han descrito la información básica, se especifican los criterios de validación que han de servir para demostrar que se ha llegado a un buen entendimiento de la forma de implementar con éxito el software. La documentación del análisis de requerimientos y manuales, permitirán una revisión por parte del cliente, la cual posiblemente traerá consigo modificaciones en las funciones del sistema por lo que deberán revisarse el plan de desarrollo y las estimaciones previstas inicialmente.

Los cuales se pueden desarrollar de la siguiente forma:

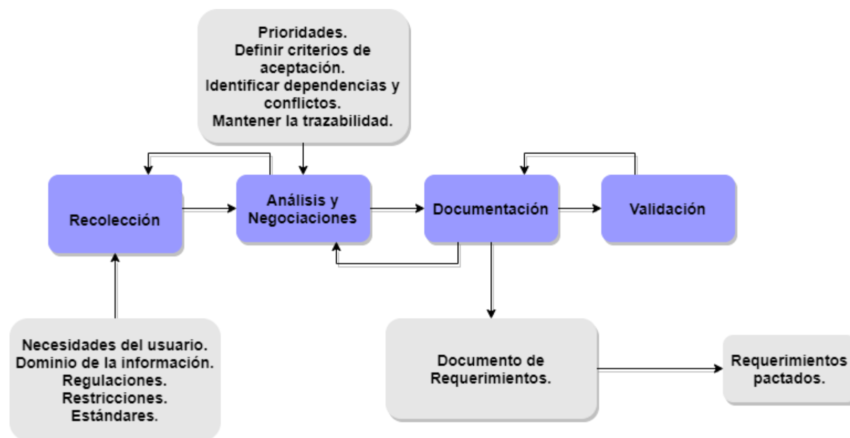


ILUSTRACIÓN 5 RESUMEN PROCESO INGENIERÍA DE REQUISITOS.

Recolección

Las cuestiones básicas que abordarán los desarrolladores son las siguientes:

Funcionalidad	¿Qué se supone que debe hacer el software?
Interfaces externas	¿Cómo interactúa el software con las personas, el hardware del sistema, otro hardware y otros programas?
Rendimiento	¿Cuál es la velocidad, disponibilidad, tiempo de respuesta, tiempo de recuperación de varias funciones de software, etc.?
Atributos	¿Cuál es la portabilidad, corrección, mantenimiento, seguridad, etc. consideraciones?
Limitaciones de diseño impuestas a una implementación	¿Existen los estándares necesarios en vigor, el lenguaje de implementación, las políticas para la integridad de la base de datos, los límites de recursos, los entornos operativos…?

TABLA 4. CUESTIONES PARA RECOLECCIÓN DE REQUISITOS

Según la ISO 830 [13] si los requisitos que se van generando son adecuados, cada uno deberá cumplir las siguientes características:

- Claro
- Correcto
- Consistente
- Coherente
- Comprensible
- Modificable
- Verificable
- Priorizada
- Sin ambigüedades
- Trazable
- Con origen creíble

Una vez conocidas las características, tanto software como hardware, que debe cumplir el producto llega al punto de la recolección de los requisitos para posteriormente agruparlos en un documento formal.

El entendimiento de este apartado se puede mejorar con a un simple ejemplo:

El cliente pide una aplicación que encienda y apague un pin GPIO de una placa cuando se quiera.

Esta descripción son los requerimientos del cliente pero para un desarrollador quedan incompletos ¿Cómo se indicará que el pin debe encenderse? ¿Quién encenderá el pin? ¿una persona, una máquina, etc.? ¿De qué condiciones partirá la aplicación? ¿Es un diseño para una aplicación crítica de seguridad? Y multitud de cuestiones más.

Es por esto por lo que existen diferentes técnicas para generar requisitos de forma completa, objetiva y teniendo en cuenta la mayor cantidad de factores posibles.

Técnicas	Descripción
Casos de Uso	Describen las interacciones entre el usuario y el sistema, centrándose en qué hace el sistema para el usuario, es decir, la totalidad del comportamiento del sistema. Los casos de uso son una técnica para la especificación de los requisitos funcionales.
Prototipos	Son simulaciones de los requisitos. Los prototipos se emplean cuando los analistas de requisitos no pueden continuar con su trabajo porque le faltan datos debido a la falta de información sobre los requisitos o el producto es innovador y se desconocen realmente los requerimientos.
Escenarios	Son contenedores que agrupan diferentes funcionalidades con el objetivo de clarificar cómo funciona el diseño sin demasiado detalle. Mediante los escenarios se pueden simular los entornos donde se desarrollará el sistema y así facilitar la recolección de requisitos.

TABLA 5. MANERAS DE GENERAR REQUISITOS.

De entre ellos, los casos de uso son la forma más común y sencilla de generar requisitos. Las principales prácticas que se deben tener, según [14], en cuenta a la hora de crear los casos de uso son los redactados a continuación. De esta forma se pretende generar una batería de requisitos lo más completa posible para que tanto desarrollador como cliente estén completamente informados de las cualidades y funcionalidades del diseño a crear.

- **Simplificar** la narración: Los casos de uso deben seguir una estructura clara y simple. Los casos de uso no deben ser narraciones si no frases concisas que expliquen lo necesario: cuando (usuario) realice (acción) debe (requisito). los casos de prueba hacen realidad las historias y su uso puede demostrar sin ambigüedad que el sistema hace lo que se supone que haga. Son los casos de prueba los que definen lo que significa implementar exitosamente el caso de uso.

- Entender el **panorama general**. Se debe entender el sistema como un todo para tomar las decisiones correctas acerca de lo que incluye el sistema. Con esto se consigue entender el alcance y el progreso a nivel del sistema.
- **Enfocarse en el valor**. Solo se genera valor si el sistema se usa realmente, así, es mejor enforcarse en cómo se empleará el sistema que en las largas listas de funciones o características que ofrecer.
- **Construir el sistema por partes**. Siempre es un error intentar construir un sistema de una sola vez. El sistema se debe construir por partes, cada una de las cuales tiene un valor claro para los usuarios.
- **Adaptarse para cumplir las necesidades del equipo**. No hay soluciones únicas pero la técnica que se seleccione debe ser lo suficientemente adaptable para cubrir las necesidades actuales del sistema.

Análisis

No solo basta con crear requisitos, estos deben ser analizados. Analizar requisitos se refiere a la actividad de realizar un estudio de cada uno individualmente, así como del conjunto, en los siguientes apartados:

- **Priorizar** requisitos: cuales se deben cumplir en la primera versión y cuales se llevan a cabo en sucesivas versiones. Esta actividad es de gran importancia en etapas de negociación con cliente. La priorización se encarga de asegurar que el producto cubrirá las necesidades mínimas. Deben dividirse los modelos que representan información, función y comportamiento de manera que se descubran los detalles por capas. Las tareas a realizar son:
    - Establecer categorías y prioridades para clasificar requerimientos.
    - Establecer análisis comparativos y evaluación de los diferentes requerimientos generados por los usuarios.
    - Todos los requerimientos deben alinearse bajo el mismo objetivo. Además también se deben alinear bajo el mismo objetivo las personas involucradas para entender cuáles son las prioridades del proyecto.

- Realizar análisis de costos y beneficios para determinar si el requerimiento debe ser considerado.

- Definir **Criterios de Aceptación** de los requerimientos. Aceptación de un requerimiento es cuando la solución planteada es exactamente la que el requerimiento solicitaba, ni más ni menos**.** Este criterio de aceptación es la forma de cuantificar los requerimientos, es decir, como se va a medir el cumplimiento de los requerimientos.

  Aplicado a requerimientos funcionales y no funcionales. En base a estos criterios de aceptación se realizan los planes de pruebas.

- Identificar **dependencias y conflictos** entre requerimientos. Los requerimientos presentan contradicciones debido a su naturaleza informal. La IEEE 830 [13] tipifica los conflictos que se pueden presentar.

Conflicto de conducta	Dos o más requerimientos especifican conductas distintas para el mismo sistema, condiciones y estímulos externos
Conflicto de términos	Términos distintos para mismo concepto
Conflicto de características	Se especifican aspectos contradictorios para la misma característica del sistema.
Conflictos temporales	Dos o más requisitos exigen características temporales distintas.

TABLA 6. TIPOS DE CONFLICTOS EN REQUISITOS.

- Mantener la **trazabilidad**

  Los requisitos deben ser trazables. Es importante conocer aspectos de los requisitos tales como:

  - Su origen (Quién los propuso)
  - Necesidad (Por qué existe)

    - Relación con otros requisitos (Dependencias)
    - Relación con otros elementos (Dependencias)

- **Especificación** de requerimientos.

  Definimos especificar como la descripción de un sistema. Se debe explicar en cada requisito su funcionalidad, restricciones, rendimientos, partes del diseño a las que afecta, tipos de usuarios… de forma clara y precisa.

Documentación

El documento de requisitos es la base en la que se desarrollará el proyecto ya que este se centrará en cumplir los objetivos marcados. La redacción de este documento debe ser entonces lo más correcta posible y no puede dar lugar a dudas. Para ello se deberán seguir una serie de pautas mínimas que ayudarán a esto.

- Deben escribirse de forma técnicamente neutra**.** Lo que el producto hace y no que tecnología se hará para crearlo.
- Evitar adjetivos y adverbios que puedan llevar a confusión. Ejemplo: Se necesita mucha memoria RAM.
- Evitar palabras ambiguas como "debería". Se quiere explicar una obligación, pero puede interpretarse como opción.
- Confirmar que los involucrados en el negocio tienen el mismo entendimiento acerca de los requisitos que la persona que los escribe. La persona que redacta los requisitos puede tener un perfil técnico tal que, tras escribir el documento, solo él sea capaz de entender todas las cuestiones técnicas especificadas en él.
- Utilizar una convención de nombres y definiciones común dentro de la organización. De esta forma se aseguran de que en todos los proyectos se está usando el mismo vocabulario.
- Emplear de plantillas para redactar de forma unificada dentro de la empresa.

Validación

Tras la redacción del documento este debe ser validado. Validar el documento de requisitos es comprobar que toda la metodología de este apartado ha sido llevada a cabo de forma correcta. Las verificaciones que se realizan son las siguientes:

- **Verificaciones de validez.** Un usuario puede pensar que se necesita un sistema para llevar a cabo ciertas funciones. Sin embargo, el razonamiento y el análisis pueden identificar que se requieren funciones adicionales o diferentes. Los sistemas tienen diversos stakeholders (se refiere a la persona o grupos de personas que están interesados en el desarrollo del proyecto) con diferentes necesidades, y cualquier conjunto de requisitos es inevitablemente un compromiso en el entorno del stakeholder.
- **Verificaciones de consistencia.** Comprobar que los requisitos de dicho documento no se contradicen, es decir no se puede dar el caso, que, para una misma funcionalidad del sistema, haya dos requisitos contrapuestos.
- **Verificaciones de completitud.** Verificar que el documento de requisitos contempla todas las tareas y limitaciones sugeridas por el usuario del sistema para este.
- **Verificaciones de realismo**. Comprobar que cada uno de los requisitos puede ser implementado utilizando la tecnología existente en el mercado, teniendo en cuenta el presupuesto y las limitaciones de tiempo para el desarrollo del sistema.
- **Verificabilidad.** Comprobar que cada uno de los requisitos existentes en el documento de especificación de requisitos, es verificable, es decir, que, con determinadas pruebas al sistema, se puede comprobar que el requisito se cumple, de esta manera se reducen los posibles conflictos entre el cliente y el contratista.

Pero si solo se siguen estos criterios teóricos puede que el análisis quede incompleto ya que pueden existir ciertos matices que pasen desapercibidos. Un análisis como el mostrado a continuación puede ser más completo y fácil de aplicar debiendo cumplir todos los criterios.

Criterio
Correcto: La especificación de un requerimiento es correcta si el sistema alcanza todos los requisitos en él especificados.
Se especifican todas las tareas que debe cumplir el software.
Todos los requisitos son relevantes en el proyecto.
Existen criterios de seguridad física.
Existen criterios de seguridad operacional.
Se especifica fiabilidad del sistema, consecuencias en caso de fallo, proceso de recuperación y detección de errores.
Se han especificado los tiempos de respuesta esperado de todas las operaciones necesarias.
Se especifican tiempos de procesamiento y transferencia de datos
Los requisitos llegan a acuerdos de compensación, ejemplo: precio y seguridad.
Están completamente definidos hardware y/o software a emplear
Está completamente especificado el entorno de usuario hardware y/o software.
Cada requisito tiene definido criterios de éxito o fracaso.
No ambiguo: Los requisitos solo deben tener una única interpretación.
Los requisitos son lo suficientemente claros para ser entendidos por cualquier persona de la organización.
Los requisitos funcionales y no funcionales se encuentran separados.
No existen múltiples interpretaciones de un requisito.
Los requisitos no entran en conflicto entre ellos
Completos: Los requisitos son completos si se definen una serie de características: Datos, entradas y salidas.
Se definen todas las interfaces de comunicación con su protocolo y su control de errores.
Se ha realizado algún análisis para localizar requisitos que no han sido incluidos.
Se especifican los puntos incompletos del documento por falta de información.
El producto final, si cumple todos los requerimientos especificados será completamente funcional y completo.
Los requisitos son implementables.
Se especifican todos los posibles valores de entrada que podría tener el sistema y la respuesta de este a ellos.
Se describe la mantenibilidad del sistema.

Se especifica los requisitos para la comunicación entre componentes.
Se especifican de forma correcta las suposiciones, restricciones y dependencias.
Consistencias: El documento concuerda con el resto de documentación de la organización.
Algún requisito se debe explicar con mayor detalle.
Algún requisito se debe explicar con menor detalle.
El documento sigue con el formato que tiene la organización.
Categorizados: Los requisitos están categorizados por importancia, estabilidad y desarrollo en tiempo. **Estabilidad se puede definir como la cantidad de cambios que se espera del requisito a lo largo del proyecto.**
Los requisitos llevan asociado un identificador único siguiendo alguna definición que imponga la organización. Ejemplo: Requisitos software se indican con RS y hardware con RH.
Los requisitos están categorizados en estabilidad, tiempo e importancia.
No existen conflictos entre categorías. Ejemplo: requisito de suma importancia pero inestable.
Verificables: Todos los requisitos de una formo u otra deben ser probados.
El lenguaje con el que se escriben los requisitos son entendibles para los stakeholders.
Cada requisito puede ser probado y a partir de esa prueba determinar si el requisito se satisface de forma plena.
Las pruebas pueden ser automatizadas.
Validar el requisito no supone un gasto mayor del valor que el requisito aporta al diseño.
Modificable: La estructuración y estilo de los requisitos debe permitir incluir modificaciones de forma clara y sencilla.
Los requisitos se identifican de forma única.
Los requisitos redundantes se encuentran identificados.
Los requisitos se especifican por separado, evitando relaciones compuestos.
Trazables: Los requisitos tiene un origen claro y el seguimiento del requerimiento se puede realizar de forma sencilla a lo largo del proyecto.

TABLA 7. ANÁLISIS DE CALIDAD DOCUMENTO REQUISITOS.

3.2. Gestión de pruebas software

Las pruebas de software son las técnicas cuyo objetivo es proporcionar información objetiva e independiente sobre la calidad del producto a la parte interesada. Es una fase que forma parte del proceso de control de calidad. Las pruebas son conjuntos indispensables de actividades dentro del ciclo de vida del producto software.

Los principios en los que se fundamentan las pruebas son los siguientes:

- **Las pruebas mostrarán la presencia de defectos.** Se debe tener en cuenta que la ausencia de fallos no prueba que el software sea correcto. Las pruebas tienen como objetivo detectar errores por lo que no deben planificarse bajo el supuesto de que no se encuentran ninguno.
- **Las pruebas exhaustivas no son posibles**. Por ello la importancia de la priorización de pruebas.
  En lugar de llevar a cabo pruebas exhaustivas se debería utilizar el análisis de riesgo y el uso de priorizaciones para optimizar el esfuerzo en pruebas. No hay que probarlo todo y fijarse en los riesgos ayuda a priorizar las pruebas a diseñar. Por ejemplo, se puede priorizar los casos para ejecutar en primer lugar los elementos más críticos basándose en las consecuencias de coste y/o tiempo si fallase x condición. Se trata de un orden de lanzamiento de las pruebas que va en base a los requisitos definidos por el usuario, marcados por el negocio.

- **Realizar pruebas tempranas**. Estas pruebas deben tener objetivos concretos. Las actividades de prueba, como pueden ser las revisiones de las mismas, deberían llevarse a cabo en paralelo a la especificación y el diseño de software.

- **Bloques de defectos.** Si aparece un defecto es muy probable que existan más a su alrededor.

- **La paradoja del pesticida.** Si se repiten una y otra vez los casos de prueba, se llegará a que el mismo conjunto de casos de prueba no sirva para generar nuevos defectos. Para ellos los casos de prueba se deben revisar de forma regular.

- **Las pruebas dependen del contexto.** Las pruebas dependerán del contexto en el cual se ejecuten; por ejemplo, las que sean para un sistema crítico de seguridad como el financiero, se requiere un mayor número de pruebas en comparación con otras aplicaciones de un nivel de complejidad menor o menos críticos.

- **La falacia de la ausencia de errores**. Localizar y corregir errores no sirve de nada si el sistema construido no cubre las necesidades y expectativas marcadas.

3.2.1. Tipos de pruebas software.

Existen diversos tipos de pruebas que se realizan a los diseños. Cada tipo de prueba proporciona diferentes correcciones dentro del código y permiten en su conjunto conseguir un sistema con los menores fallos posibles.

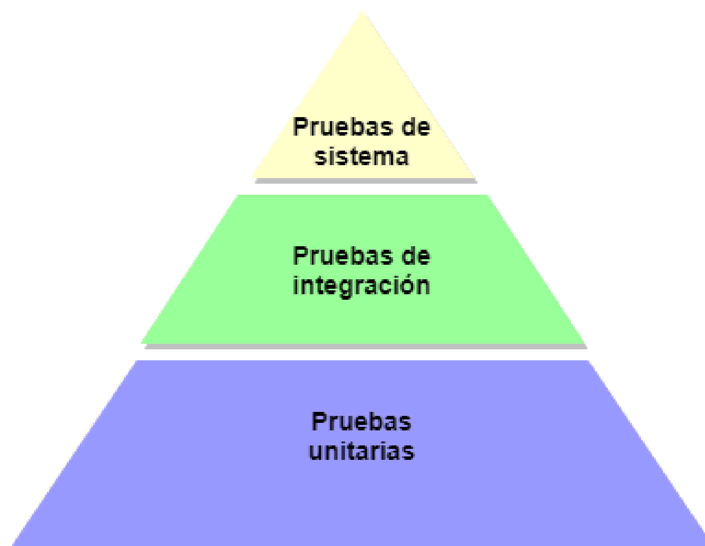


ILUSTRACIÓN 6. TIPOS DE PRUEBAS SOFTWARE.

PRUEBAS UNITARIAS.

Las pruebas unitarias son el primer nivel dentro de las pruebas software. Consiste en la validación de unidades software de forma aislada, es decir, probar una unidad de código con sentido lógico. Estas unidades de código se consideran una unidad de programa, como una función o método de una clase que es invocada desde fuera de la unidad y que puede

invocar a otras unidades. Estas unidades, entonces, deben ser probadas de forma independiente.

Estas pruebas, que se crean a la par que la aplicación, ayudan al desarrollador a conocer el estado del código. Los errores que las pruebas unitarias ponen de manifiesto son los siguientes.

- Errores de ejecución.
- Mermas de rendimiento.
- Código muerto.
- Redundancia de código.
- Incumplimiento de arquitectura.
- Incumplimiento de estándares técnicos a nivel de codificación.
- Mantenibilidad del código.

Las pruebas unitarias son costosas de elaborar y en proyectos con escaso tiempo acaban siendo despreciadas. Pero como se ha visto, son la base para la realización de un código confiable y seguro desde sus inicios además de presentar las siguientes ventajas:

- Las pruebas unitarias permiten localizar los errores a tiempo. Los fallos a medida que avanza el proyecto se vuelven más significativos haciendo que el tiempo y el coste en su reparación acabe siendo mayor que el tiempo que se hubiese empleado en la ejecución de las pruebas unitarias pertinentes.

- Se asegura la calidad del código entregando al cliente un código limpio con un diseño claro.

- Facilita los cambios y favorece la integración. Los test unitarios permiten modificar parte del código sin afectar al conjunto, además permiten la aportación de nuevas funcionalidades sin afectar al conjunto del código.

PRUEBAS DE INTEGRACIÓN.

Es el siguiente paso a dar dentro de la validación del código. Tras haber realizado las pruebas unitarias, la siguiente tarea es testear como estas unidades se integran entre ellas.

El ISTQB (International Software Testing Qualifications Board) dice que estas pruebas se ocupan de probar las interfaces entre los componentes, las interacciones con las distintas partes de un mismo sistema, como el sistema operativo, el sistema de archivos, el hardware y las interfaces entre varios sistemas.

Para realizar esta integración se pueden seguir varias estrategias de las que destacan según [15].

- Top-Down: El primer componente que se desarrolla y prueba es el primero de la jerarquía (A). Los componentes de nivel más bajo se sustituyen por componentes auxiliares para simular a los componentes invocados. En este caso no son necesarios componentes conductores. Una de las ventajas de aplicar esta estrategia es que las interfaces entre los distintos componentes se prueban en una fase temprana y con frecuencia.
- Bottom-Up: En este caso se crean primero los componentes de más bajo nivel (E, F) y se crean componentes conductores para simular a los componentes que los llaman. A continuación se desarrollan los componentes de más alto nivel (B, C, D) y se prueban. Por último dichos componentes se combinan con el que los llama (A). Los componentes auxiliares son necesarios en raras ocasiones. Este tipo de enfoque permite un desarrollo más en paralelo que el enfoque de arriba abajo, pero presenta mayores dificultades a la hora de planificar y de gestionar.
- Big-Bang: integración no incremental, se prueba cada componente por separado y posteriormente se integran todos de una vez realizando las pruebas pertinentes

PRUEBAS DEL SISTEMA

Una vez que se prueban los componentes y su integración se comienza a probar las funcionalidades que deberá cumplir el sistema. Estas pruebas son las encargadas de validar los requisitos del producto. Las pruebas de sistema deben de estudiar los requisitos funcionales y no funcionales y las características de calidad. Para esto se aplican técnicas de caja negra.

Los sistemas de caja negra funcionan teniendo en cuenta solo la respuesta que produce el sistema ante una entrada sin tener en cuenta lo que ocurre en el interior del sistema.

Para sistemas grandes, esto puede ser un proceso de múltiples etapas, donde los componentes se conjuntan para formar subsistemas que se ponen a prueba de manera individual, antes de que dichos subsistemas se integren para establecer el sistema final.

PRUEBAS DE VALIDACIÓN O ACEPTACIÓN.

Ésta es la etapa final en el proceso de pruebas, antes de que se acepte para uso operacional. El sistema se pone a prueba con datos suministrados por el cliente, en vez de datos de prueba simulados. Las pruebas de aceptación revelan los errores y las omisiones en la definición de requerimientos del sistema, ya que los datos reales ejercitan el sistema en diferentes formas que los datos de prueba. Asimismo, las pruebas de aceptación revelan problemas de requerimientos, donde las instalaciones del sistema en realidad no cumplan las necesidades del usuario.

Los tipos de prueba que se realizan son la siguiente:

- Pruebas de aceptación de cliente. Son las pruebas tomando como base el contrato realizado al principio del proyecto. De esta forma se conoce si la expectativas del cliente han sido cumplidas. Suelen ser las mismas que las pruebas del sistema.

- Pruebas operativas. Son las pruebas que se llevan a cabo por los administradores que de sistema. Las tareas que se incluyen son las siguientes: Copias de seguridad,

gestión de usuarios, comprobar las vulnerabilidades, carga de datos, mantenimientos, etc.

- Pruebas alfa-beta. Los usuarios finales pueden ser diversos, es por esto por lo que se realizan las pruebas alfa-beta que van a descubrir errores que solo el usuario final va a encontrar.
    - Pruebas alfa. Son las pruebas que se realizan con un grupo de personas que representan al usuario final, estando los desarrolladores presentes anotando los errores.
    - Pruebas beta. Esta vez el desarrollador no está presenten y serán las personas que cumplen el papel de usuario los que anotarán los errores que ellos encuentran.

3.2.2. Criterios de finalización.

Los criterios de finalización son una serie de criterios que indican que las pruebas realizadas han sido suficientes para validar la aplicación. Se debe buscar una batería de pruebas que garantice que se encontrarán los errores del sistema. Para ellos es importante definir el alcance de las pruebas en base al impacto que tendrían en el futuro si fallasen. La consigna es conseguir que los costes de las pruebas sean menores a los costes de fallo. Pero existen otros criterios de finalización que pueden emplearse según el caso específico de prueba:

- Ratio entre el número de defectos encontrados y el tamaño del sistema por unidad del tiempo de prueba.
- Numero de caminos probados. Proporción entre el número de rutas probadas y su total.
- Numero de defectos durante la producción. Esta métrica indica el número de defectos no encontrados durante el proceso de prueba.
- Efectividad de la detección de defectos. Número total de defectos encontrados durante las pruebas, dividido entre el total de defectos estimados.
- Presupuesto utilizado. Proporción del presupuesto y el coste actual de las pruebas.

- Eficiencia de las pruebas. Número de pruebas requeridas entre el número de defectos localizados.

3.2.3. Creación del plan de pruebas.

La planificación de las pruebas es un punto muy importante a la hora de validar un proyecto. En este plan se especifica todo lo que se va a probar, cómo se va a aprobar, los recursos de los que se dispone, la documentación del proyecto, etc.

Un ejemplo completo de un plan de pruebas lo ofrece el estándar de la IEEE 29119 [16]. Esta normativa define una estructura clara a cumplir en un plan de pruebas para que sea lo más correcta posible.

1. Identificador único.
2. Introducción y resumen de elementos y características por probar.
3. Elementos software que se probarán; se identifican los módulos de entrada, los procesos y las salidas. Y documentos por probar; por ejemplo, el documento de diseño, el de requerimientos...
4. Características por probar: se describe las características del software a examinar.
5. Características que no se probarán como, por ejemplo, condición de error no detectadas, pruebas de carga o estrés.
6. Enfoque general de las pruebas: describir las actividades, técnicas y herramientas para realizar pruebas de diseño, unitarias, integración...
7. Informes por entregar: se detallan los que se deben entregar, con base a la labor que se realiza en el apartado de "software que se probará". Los documentos son, por ejemplo, el informe de pruebas unitarias, de integración, de interfaz gráfica, de rendimiento.
8. Actividades de preparación y ejecución de pruebas: va el cronograma de actividades del equipo de trabajo de pruebas.
9. Necesidades de entorno: se refiere a los requerimientos de software y hardware para realizar la ejecución del plan

10. Necesidades de personal y de formación: son las recomendaciones al tipo de personal que se necesita, se describen las habilidades y el conocimiento que deben tener.
11. Esquema de tiempos: se fija la estimación de las pruebas y se incluye la metodología utilizada por la organización para dicho fin.
12. Aprobaciones: se detalla los empleados o funcionarios aprobadores del plan.

3.2.3.1. Factores de calidad de un plan de pruebas.

Al igual que sucede con los documentos de requisitos hardware y software, el plan de prueba debe ir cumpliendo una serie de características que miden la calidad del producto. Estos criterios se pueden gestionar mediante el siguiente análisis.

Criterios
Calidad.
Todas las pruebas se adaptan al entorno y a los probadores.
Su descripción se puede probar.
Tiene solo los pasos y campos necesarios para su propósito.
Las pruebas son repetibles esperando siempre los mismos resultados.
La prueba es rastreable sabiendo que requisito se está probando.
Las pruebas son medibles retornando con un valor o estado.
Estructura y calidad de prueba.
Cada prueba tiene un identificador único.
Cada prueba tiene un propósito declarado haciendo alusión al requisito que se está probando.
Cada prueba tiene una descripción del método de prueba.
En cada prueba se especifican la información de configuración: prerrequisitos, entorno, datos de entrada, seguridad…
Tiene acciones y resultados esperables.
Cada prueba guarda el estado de la prueba como informe, captura de pantalla…
El entorno de pruebas se mantiene limpio.
Cada prueba tiene los menores pasos posibles sin superar los 15.
La configuración permite modificar los prerrequisitos de las pruebas cuando sea necesario.

TABLA 8. ANÁLISIS CALIDAD DOCUMENTO DE PRUEBAS.

4. Herramientas empleadas.

DevOps se comprende de una serie de herramientas que ayudan a aportar valor al producto. El uso de herramientas es crucial para conseguir la automatización en el proceso de validación.

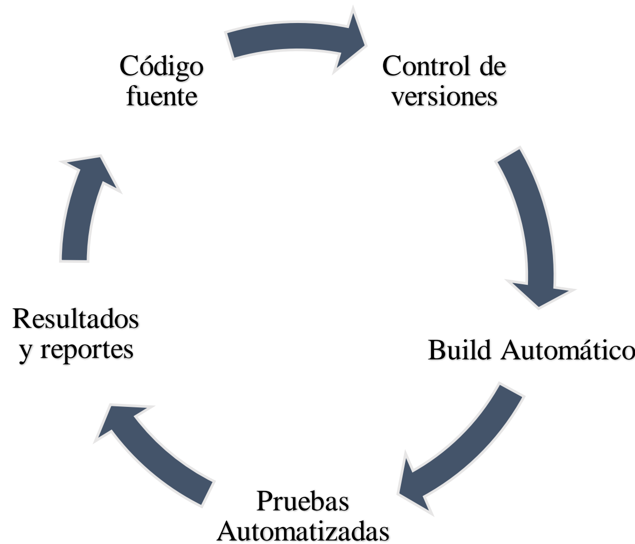


ILUSTRACIÓN 7. PROCESO DE INTEGRACIÓN CONTINUA.

En este apartado, se explicarán las principales herramientas que son necesarias para conseguir un proceso de IC lo más eficiente posible aportando al desarrollo a su vez conceptos de DevOps.

4.1. Control de versiones.

Los sistemas de control de versiones son una categoría de herramientas que ayudan a un equipo de software a gestionar los cambios en el código fuente a lo largo del tiempo. La herramienta de control de versiones realiza un seguimiento de todas las modificaciones en el código en un tipo especial de base de datos. Si se comete un error, los desarrolladores pueden ir atrás en el tiempo y comparar las versiones anteriores del código para ayudar a resolver el fallo al tiempo que se minimizan las interrupciones para todos los miembros del equipo.

Los equipos de software que no utilizan ninguna forma de control de versiones a manudo se encuentran con problemas como no saber qué cambios se han realizado anteriormente o realizar cambios incompatibles entre dos partes relacionadas.

El software de control de versiones es una parte esencial del día a día de las prácticas profesionales del equipo de desarrollo software moderno.

Git + Bitbucket.

Los sistemas de control de versiones han experimentado grandes mejoras en las últimas décadas. De entre ellas destaca Git. Git es un sistema de control de versiones distribuido (VCS), una categoría conocida como DVCS. Al igual que muchos de los sistemas de VCS más populares disponibles hoy en día, Git es gratuito y de código abierto [17].

ILUSTRACIÓN 8. LOGO GIT Y BITBUCKET.

Para el empleo de Git, es necesaria la creación de un repositorio centralizado al cual los desarrolladores puedan acceder.

Cada desarrollador tiene la capacidad de clonar ese repositorio en su equipo local. Así, pueden realizar los cambios necesarios del código en su ordenador personal y posteriormente subirlos a la rama que corresponda al repositorio de control de versiones de Bitbucket.

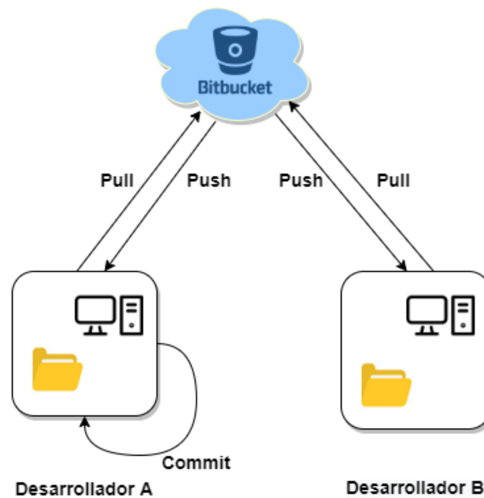


ILUSTRACIÓN 9. FUNCIONAMIENTO GIT.

El repositorio se organiza en ramas. En vez de una única rama maestra, un flujo de trabajo mejorado utiliza otra serie de estructura para registrar el historial de trabajo. La creación de una rama en las herramientas de VCS mantiene múltiples flujos de trabajo independientes los unos de los otros al tiempo que ofrece la facilidad de volver a fusionar ese trabajo, lo que permite que los desarrolladores verifiquen que los cambios de cada rama no entran en conflicto.

4.2. Compilación de aplicaciones.

En la gran mayoría de proyectos software, la aplicación debe ejecutarse en un hardware separado del propio PC del programador. Contar con una herramienta que sea capaz de compilar las aplicaciones en un ambiente separado del hardware para que posteriormente se ejecute en este es un paso dentro de los procesos automáticos de Integración Continua.

Docker.

Docker es una herramienta para la gestión de contenedores. Los contenedores son empaquetados de un programa software que optimiza el espacio ocupado y facilita la migración a diferentes servidores. El propósito de los contenedores es ejecutar varios procesos y aplicaciones por separado para hacer un mejor uso de su infraestructura y, al mismo tiempo, conservar la seguridad.

ILUSTRACIÓN 10. LOGO DOCKER.

Las herramientas de gestión de contenedores como Docker ofrecen un modelo de implementación basado en imágenes. Esto permite compartir una aplicación con todas sus dependencias en varios entornos.

La ventaja que aporta esta herramienta en el proceso de validación de aplicaciones es la capacidad de compilarlas en el ambiente del contenedor y posteriormente ser capaces de ejecutar la aplicación en la placa marcada para el testeo. Desarrollar un nuevo hardware, proveerlo del entorno de compilación y ejecutar la aplicación puede ser algo tedioso. Contando con contenedores portátiles cualquier hardware puede ser testeado fácilmente.

4.3. Pruebas software.

El uso de herramientas dedicadas especialmente a testear aplicaciones hace posible la tarea de validación automática. La propia herramienta es la que ejecuta los comandos del test, comprueba los resultados y genera reportes de las pruebas.

El empleo de herramientas de test lleva consigo una serie de ventajas inmediatas:

Las pruebas automáticas garantizan documentación actualizada. Cuando los casos de prueba están escritos en lenguaje natural, describen las propiedades del sistema implementado de forma inteligible. Al mismo tiempo, también garantizan que el producto

funciona según lo deseado. Por lo tanto, los casos de texto ejecutados correctamente forman una imagen actualizada de las funcionalidades del sistema.

Las pruebas automáticas siempre se ejecutan de la misma manera. Cuando las pruebas se ejecutan manualmente, siempre existe el riesgo de que algunas de ellas permanezcan sin hacer o los resultados de las pruebas no sean fiables debido a un error de escritura cometido por el probador. Las pruebas automáticas, por otro lado, garantizan que el producto funcione siempre de la misma manera. Si las pruebas automáticas garantizan suficientemente la funcionalidad deseada, los recursos liberados de las pruebas manuales se pueden asignar a trabajos más productivos.

Los errores se descubren más rápido. Las pruebas automáticas se ejecutan idealmente después de cada cambio para que los errores del sistema se detecten casi inmediatamente. Cuando los errores se encuentran en el tiempo, el desarrollo es más gratificante y los costos incurridos por la corrección de errores disminuyen significativamente.

Mayor cobertura de prueba. Cuando se ha automatizado un caso de prueba, se agrega a un conjunto de pruebas de regresión, que se puede ejecutar prácticamente tan a menudo como sea necesario. Por lo tanto, cada caso de prueba automático aumenta el conjunto de pruebas que garantiza la funcionalidad del producto implementado.

La automatización de pruebas hace que el desarrollo sea más seguro. Cuando las pruebas automáticas garantizan el funcionamiento del producto implementado, el desarrollo logra una nueva sensación de seguridad. El enfoque del desarrollo puede cambiar más libremente a tomar decisiones técnicas sensatas en lugar de cambios cautelosos, porque el equipo de desarrollo puede estar seguro de que un mensaje de error automático y dirigido señalará cualquier falla.

Robot Framework

Robot Framework es un marco de automatización de código abierto genérico el cual se puede utilizar para la automatización de pruebas.

Robot Framework es abierto y extensible y se puede integrar con prácticamente cualquier otra herramienta para crear soluciones de automatización potentes y flexibles. Ser

de código abierto también significa que Robot Framework es de uso gratuito sin costos de licencia.

ILUSTRACIÓN 11. LOGO ROBOT FRAMEWORK.

Está escrito en Python y basado en keywords y suites. Las keywords son una función que se define en lenguaje Python, la cual se puede llamar desde una suite.

Además Robot Framework cuenta con un editor de test llamado RIDE, en el cual se puede consultar las bibliotecas, variables y keywords.

Las principales características de la herramienta son las siguientes:

Flexible	Robot Framework puede ser adaptado para testear cualquier software o producto. Es el indicado para testear software integrado.
Lenguaje natural	Los test escritos en Robot Framework pueden ser leídos por cualquier persona. Los requisitos se pueden vincular lógicamente sin problemas a las pruebas.
Libre	De software libre, usar esta aplicación para testear es totalmente gratuito.
Popular	Es una herramienta muy popular tanto en grandes como pequeñas empresas.

TABLA 9. CARACTERÍSTICAS ROBOT FRAMEWORK

4.4. Integración continua

Las herramientas de integración continua son un entorno donde poder desarrollar todo el proceso teórico de integración. Funcionan a base de canalizaciones donde se van indicando los pasos automáticos que realizará la herramienta cuando se lance la ejecución de esa canalización.

Jenkins.

En la actualidad existen una gran variedad de alternativas de herramientas para la integración continua donde destaca Jenkins.

ILUSTRACIÓN 12. LOGO JENKINS.

Jenkins es una herramienta web de soporte para la integración continua, de licencia open-source y escrita en JAVA. Las principales características son:

- Es una herramienta sencilla de instalar.
- Soporta trazabilidad con herramientas de gestión de cambios, siendo capaz de generar listas con las modificaciones realizadas.
- Integración de mensajería.
- Posibilidad de crear informes de test.
- Soporta varios proyectos simultáneos.
- Soporte de plugins, como por ejemplo, Robot Framework, Bitbucket…

Además de esto, es una herramienta sencilla de configurar y ampliar gracias a una interfaz intuitiva y la multitud de plugins que integra.

4.5. Gestión de reportes y documentos

Una vez que todo el proceso ha finalizado, llega el momento de gestionar los reportes de las pruebas. Es el paso en el que se gestionan los requisitos del sistema, comprobando su validación o no. En las metodologías ágiles esto debe realizarse a lo largo de todo el proyecto mantenido una comunicación abierta entre todos los participantes del mismo. La información del estado de los requisitos debe actualizarse de forma continua, estando esta información pública para todos los desarrolladores.

Jira + Confluence

Jira Software es una de las herramientas más populares de Atlassian. Jira está diseñada para que los miembros de un equipo de software puedan planificar, supervisar y publicar software de calidad.

Actualmente más equipos desarrollan de forma interactiva, y Jira es el núcleo central para etapas de codificación, colaboración y publicación. Los equipos de control de calidad utilizan las incidencias, pantallas personalizadas, campos y flujos de trabajo de Jira para gestionar los proyectos.

En vista de comenzar a emplear metodologías ágiles, Jira proporciona tableros listos para usar. Los tableros son centros de gestión de tareas asignadas a diferentes flujos de trabajo. Asimismo, los tableros ofrecen transparencia acerca del trabajo en equipo y visibilidad del estado de cada elemento de trabajo.

ILUSTRACIÓN 13. LOGO DE JIRA Y CONFLUENCE.

Confluence es un espacio de trabajo colaborativo que se integra perfectamente con Jira. Las páginas dinámicas ofrecen al equipo un entorno donde crear, capturar, compartir trabajo y colaborar en cualquier proyecto. Confluence está destinado a los equipos de cualquier tipo y tamaño, desde aquellos con proyectos críticos de gran trascendencia que necesitan rigor en sus prácticas hasta aquellos que buscan un espacio para crear una cultura de equipo e interactuar los unos con los otros de una forma más abierta.

La elección de estas herramientas fue debido a su anterior uso en la empresa, especialmente Confluence con fines de repositorio compartido de documentos, ya que estas herramientas son de pago a base de licencias.

5. Diseño de un proceso automático de validación.

El siguiente objetivo del proyecto era la construcción de un proceso de Integración Continua con las herramientas planteadas en el apartado anterior que fuese capaz de validar aplicaciones alojadas en embebidos.

5.1. Diseño planteado.

El esquema de funcionamiento del diseño es el siguiente:

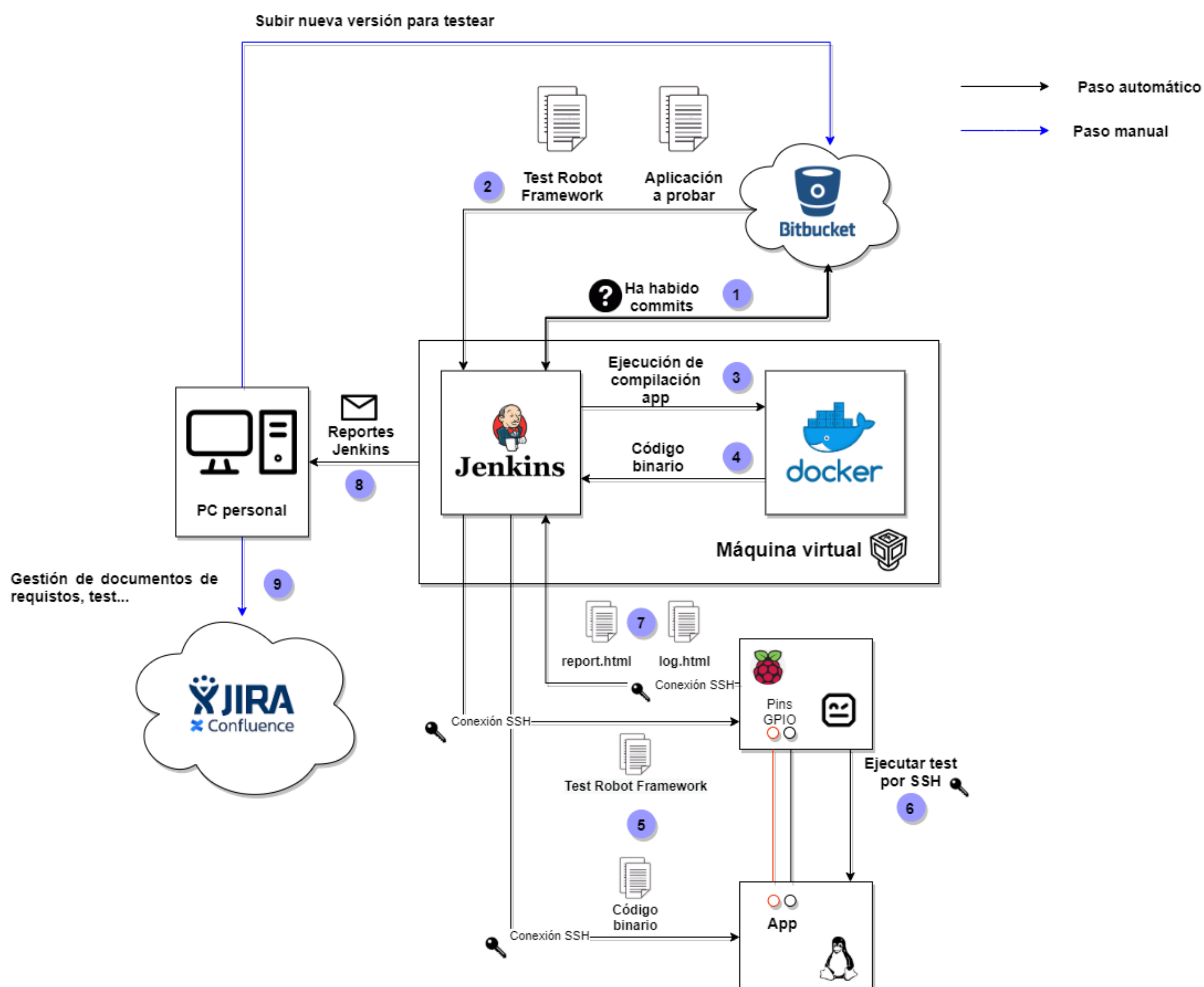


ILUSTRACIÓN 14. FLUJO PROCESO IC.

1. El proceso se lanza si se ha detectado un nuevo *commit* en la rama del repositorio dedicada a las pruebas. De esta forma, solo se deberá lanzar el proceso cuando haya un cambio en dicha rama.

2. El primer paso que debe ejecutar Jenkins es descargar (lo que Git denomina clonar) todo lo que se encuentre en la rama master de Bitbucket. De esta forma, Jenkins genera un workspace propio por proyecto donde se encuentran los archivos del repositorio así como los demás archivos que se vayan generando a lo largo del proceso.

3. La compilación de la aplicación se realiza en una imagen de Docker creada específicamente para simular el entorno de compilación de la aplicación a testear.

4. El código binario, listo para ejecutar en la placa, se guarda en el espacio de trabajo de Jenkins.

5. El test se envía a una Raspberry dedicada a testear la aplicación en la cual se ha instalado el programa Robot Framework. Al mismo tiempo, el código binario se envía al hardware a testear. Este envío se realiza a través del protocolo SSH.

6. Desde Jenkins mediante protocolos SSH se ordena a la Raspberry que ejecute los test de la placa. Cuando los test finalizan, Robot Framework genera unos reportes que son guardados en la propia Raspberry.

7. Los reportes de las pruebas se envían a Jenkins para que estos puedan ser borrados de la Raspberry. También se envían mediante SSH.

8. Finalmente, el último paso que debe realizar Jenkins es enviar un email con los resultados de las pruebas al PC del desarrollador.

9. En Jira y Confluence se gestionan tanto los documentos como los requisitos del proyecto para que estén presentes para todos los partícipes del proyecto.

5.1.1. Protocolo SSH.

La comunicación remota entre los dispositivos es indispensable para poder automatizar el proceso al ejecutarse de forma completamente ajena al desarrollador. Desde la máquina virtual donde se ejecuta la canalización, es necesario el intercambio de archivos y la ejecución de comandos. Es por esto por lo que se emplea el protocolo SSH.

SSH (o Secure Shell) es el nombre de un protocolo y del programa que lo implementa cuya principal función es el acceso remoto a un servidor por medio de un canal seguro en el que toda la información está cifrada.

A diferencia de otros protocolos, tales como FTP o Telnet, SSH encripta la sesión de conexión, haciendo imposible que alguien pueda obtener contraseñas no encriptadas. Esta seguridad en la conexión hace que este protocolo sea el más indicado para las conexiones remotas en entornos empresariales donde la seguridad informática es primordial.

Para realizar la conexión de la forma más cómoda posible se emplean las llamadas claves públicas y privadascon un cifrado simétrico. Su funcionamiento esquematizado es el siguiente:

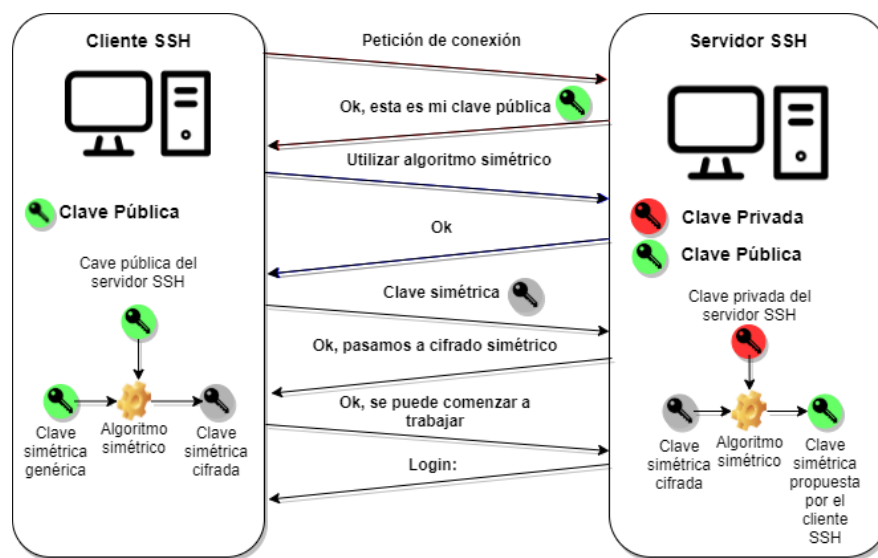


ILUSTRACIÓN 15. FUNCIONAMIENTO LLAVES PÚBLICA/PRIVADA POR SSH.

De esta manera, quien envía un mensaje tiene que cifrarlo con la llave pública del otro usuario y, para que este descifre el mensaje, tiene que usar su llave privada. De igual manera en el sentido contrario, el usuario que responda al mensaje tiene que cifrarlo con la llave

pública del primer usuario y éste tiene que usar su propia llave privada para ver la respuesta. Esto permite tener conexiones completamente seguras.

El uso de clave pública y clave privada en SSH implica un cifrado criptográfico asimétrico que una password simple no es capaz de ofrecer. Además, hace posible que los usuarios puedan acceder al server SSH sin tener que recordar contraseñas largas.

El uso del par de claves pública/privada en SSH, implica que la clave privada permanece solo en el equipo del cliente. Ejecutando en la consola del cliente `ssh-keygen` se generarán dos archivos (id_rsa y id_rsa.pub) con las claves que posteriormente serán tratadas para su correcto uso.

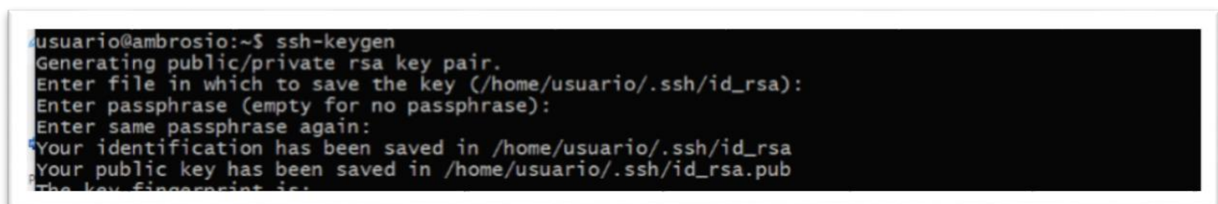


ILUSTRACIÓN 16. CONSOLA TRAS GENERACIÓN DE CLAVES.

5.2. Implementación del diseño.

Una vez finalizada la explicación del diseño a implementar, el siguiente paso es definir la configuración y uso de las herramientas. Se realiza una pseudo guía que pueda ayudar a los futuros desarrolladores de un proyecto a comenzar a montar un sistema más complejo de validación.

5.2.1. Jira + Confluence.

La gestión del proyecto y de requisitos se realiza mediante Jira. La forma en la que se realiza esta gestión de requerimientos es muy sencilla, intuitiva y efectiva si los desarrolladores llegan a un acuerdo en la forma de gestionar las tareas que proporciona el propio Jira. El aspecto de un proyecto es el siguiente:

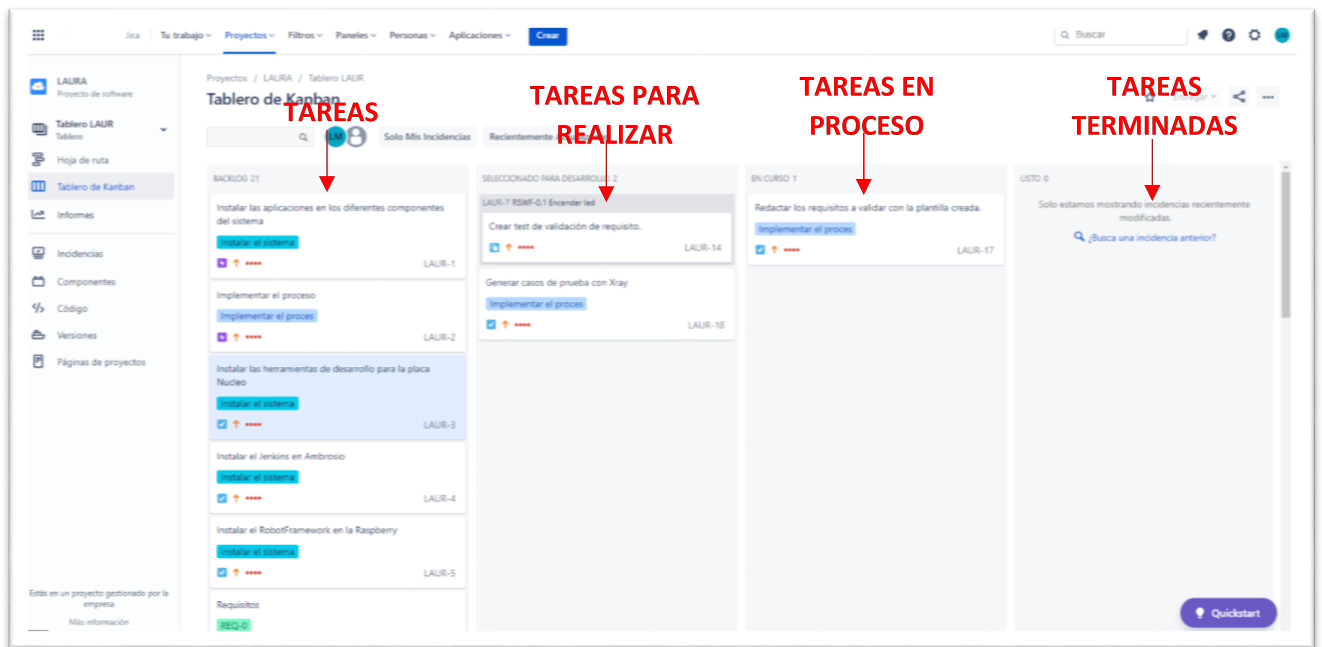


ILUSTRACIÓN 17. TABLERO PROYECTO JIRA

Como se observa, funciona en base a diferentes tipos de tareas que se van definiendo y moviendo a lo largo de los estados del tablero.

La forma más efectiva e intuitiva de gestionar los requisitos es seguir el siguiente método:

- Los *EPICS* de Jira se emplean como agrupaciones de tareas grandes. Uno o varios *EPIC*, dependiendo de la cantidad de requisitos (puede existir uno para funcionales y otro para no funcionales), deben englobarlos a todos.
- Dentro de estos *EPICS* se van introduciendo las diferentes *HISTORIAS* que hacen referencia a un único requisito.

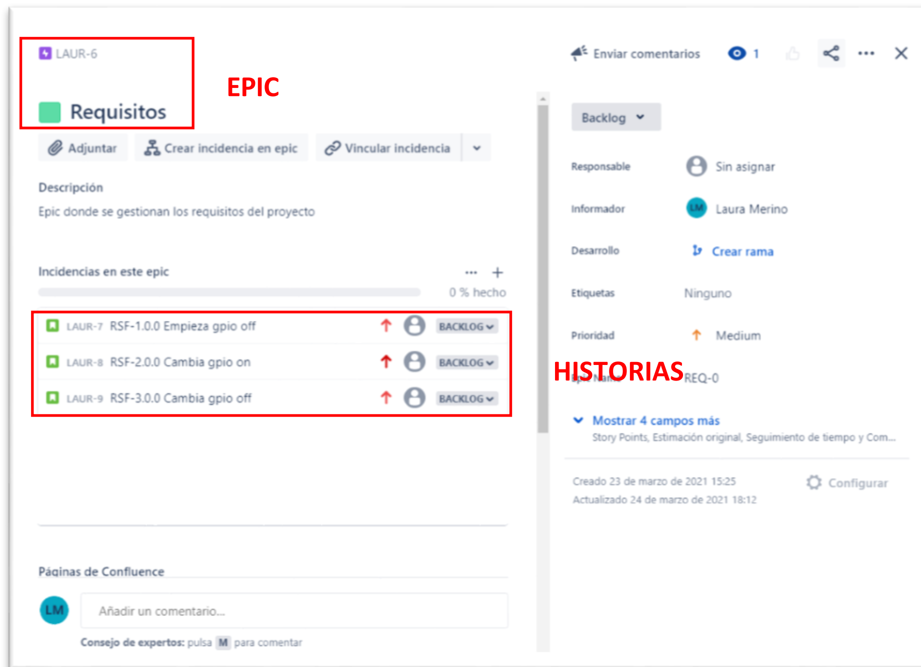


Ilustración 18. EPIC de requisitos

Como se observa, se genera una tarea tipo EPIC donde se encuentran las HISTORIAS con el nombre de cada requisito.

- Las TAREAS serán la lista de objetivos que harán que, una vez completadas todas ellas, el requisito se dé por validado.

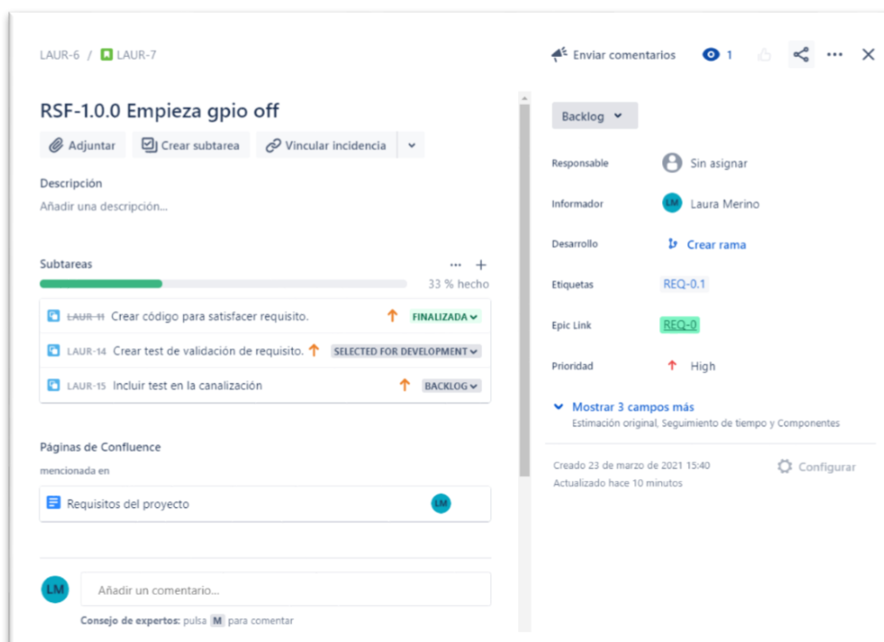


Ilustración 19. Historia de un requisito Jira

En el requisito se puede comprobar el estado de cada subtarea de forma directa. El propio desarrollador es quien debe modificar el estado de la tarea y así hacerlo visible para los demás participantes del proyecto.

De esta forma se consigue un control global del estado del proceso de validación y verificación. La herramienta siempre está activa, cambiando las tareas de columna según los puntos que se vayan llevando a cabo y añadiendo comentarios en los diferentes *EPICS*, *HISTORIAS…* que clarifiquen el estado del proyecto.

Jira no es una herramienta de gestión de requisitos, esa no es su función principal. Aun así, un correcto uso puede ayudar a los equipos de desarrollo relativamente pequeños a mantener un seguimiento completo de los requisitos, una comunicación abierta entre participantes y la posibilidad de comenzar a incorporar principios básicos de metodologías ágiles a sus desarrollos centrando la atención en el software que se va desarrollando.

Confluence se emplea para gestionar los documentos del proyecto. Crea un repositorio de documentos en la nube que puede ser gestionado por todos los participantes del proyecto y además es capaz de conectarse directamente con Jira y proporcionar enlaces entre el desarrollo y la documentación. Los documentos se encuentran presentes en la herramienta y mediante simples etiquetas el desarrollador puede comprobar el estado del requisito que se encuentra leyendo, dirigiéndose así a Jira.

5.2.2. Git + Bitbucket

La existencia de estas herramientas es crucial en equipos de desarrollo software, su configuración debe realizarse al comienzo de cualquier proyecto, antes siquiera de crear una mera línea de código.

Una vez instalado Git en el PC del desarrollador, se crea una carpeta del proyecto que se clonará en la nube de Bitbucket.

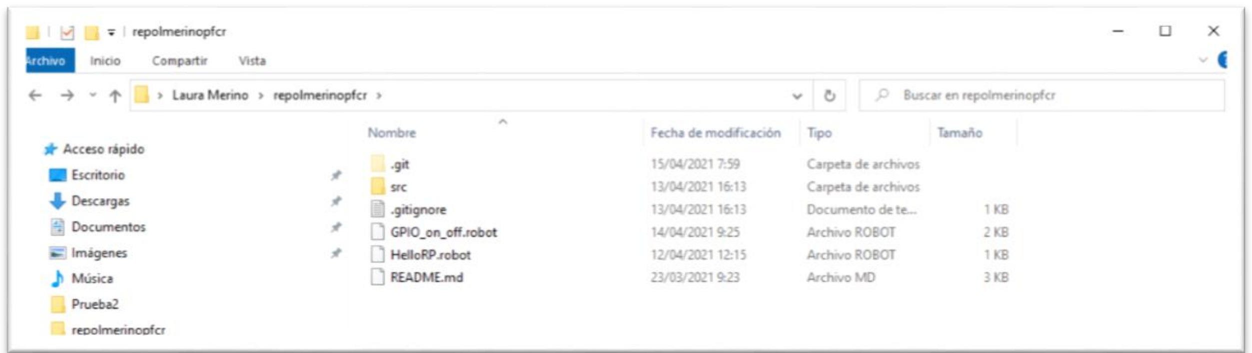


ILUSTRACIÓN 20. EJEMPLO REPOSITORIO LOCAL.

Para poder realizar esta conexión segura entre el repositorio local y el remoto se emplean llaves SSH. Para ello se modifica el archivo *config* dentro del directorio .ssh de la siguiente forma.

```
Host bitbucket.org
        HostName bitbucket.org
        IdentityFile ~/.ssh/id_rsa.pub
        User
```

Tras haber realizado los cambios, cuando se desee probar la aplicación bastará con ejecutar el siguiente código para subir los cambios de la carpeta a la rama master del repositorio remoto:

```
git add -A
git commit -m "Nuevo cambio"
git push origin master
```

Bitbucket tendrá entonces la misma forma que la del repositorio local.

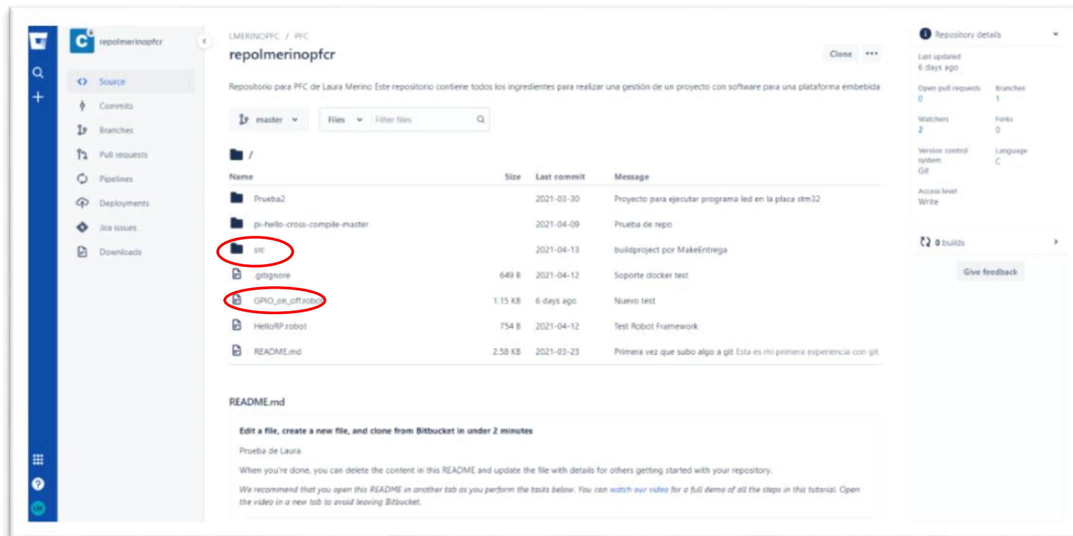


ILUSTRACIÓN 21. ASPECTO REPOSITORIO.

Como se ha comentado, los repositorios se organizan en diferentes ramas de desarrollo. Estas ramas tienen una funcionalidad concreta y son las siguientes, segúm [18]:

- **Maestra**: Esta rama es la que almacena el historial de publicación oficial.
- **Desarrollo**: Se emplea como rama de integración para funciones. Es la rama maestra no oficial.
- **Rama Release o rama de publicación**. Una vez que el desarrollo ha adquirido suficientes funciones para una publicación se bifurca una rama de versión a partir de una de desarrollo. Cuando se crea esta rama, se inicia el siguiente ciclo de publicación, por lo que no pueden añadirse nuevas funciones tras este punto; solo las soluciones de errores, la generación de documentación y otras tareas orientadas a la publicación deben ir en esta rama. Una vez que esté lista para el lanzamiento, la rama de publicación se fusiona en la maestra y se etiqueta con un número de versión. Además, debería volver a fusionarse en la de desarrollo, que podría haber progresado desde que se inició la publicación.
- **Rama Feature**. En esta rama se deben incluir nuevas características, nuevos requisitos o nuevas historias de usuario.
- **Rama Hotfix o ramas de corrección**. Es una rama dedicada a depurar el código de errores críticos que deben resolverse. Se utilizan para reparar rápidamente las publicaciones de producción. Estas ramas se basan en la

maestra en vez de en la de desarrollo. Una vez el problema se ha solucionado la rama se fusiona con la master y la de desarrollo o la de publicación La nueva rama maestra debería ir acompañado con una nueva versión.

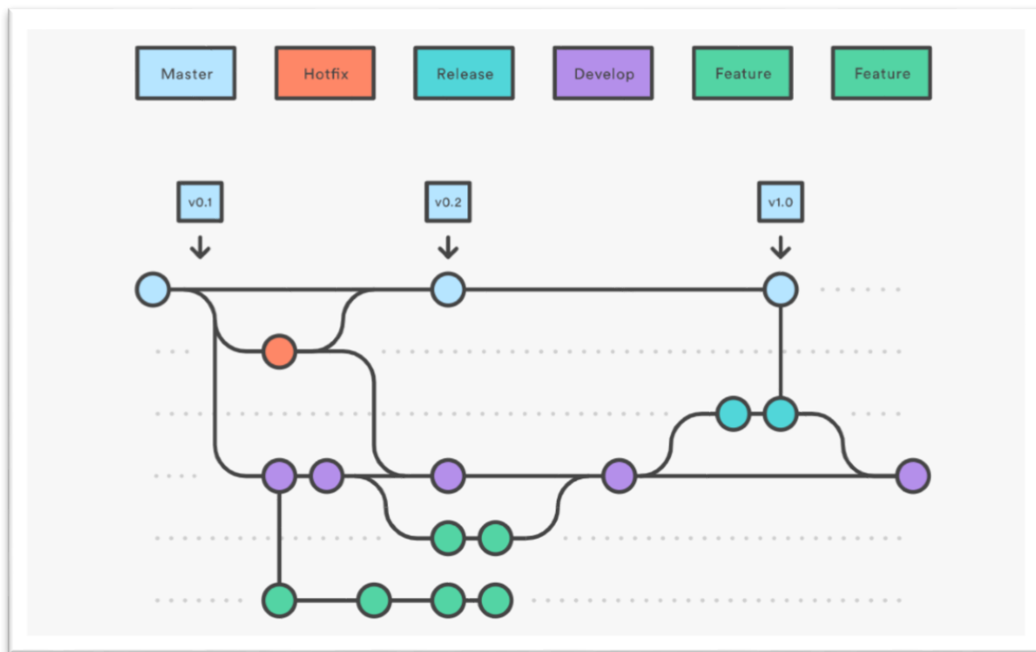


ILUSTRACIÓN 22. FLUJO DE RAMAS DEL REPOSITORIO. FUENTE [18]

5.2.3. Jenkins

Jenkins será la herramienta que consiga automatizar todo este proceso. Trabajará en una máquina virtual creada para este fin. Tras su instalación, bastará con iniciar una nueva sesión y acceder para comenzar a configurar el servidor y crear la primera canalización.

➢ Plugins.

Los plugin son extensiones de Jenkins que aportan diferentes funcionalidades. Estos se instalan desde *Panel de Control > Administrar Jenkins > Administrar Plugins* y en la barra de búsqueda se introduce, seleccionando la pestaña de *Todos los plugins,* aquellos que se quieren instalar.

Según las primeras funcionalidades que se desearon dar a la herramienta se instalaron las siguientes extensiones.

- Bitbucket Plugin: Para integrarse con el repositorio remoto de Bitbucket.
- Docker plugin: Integra Jenkins con Docker
- Publish Over SSH: Incluyen diferentes comandos de SSH.
- Robot framework: Para poder interactuar con los resultados de los test de robot framework desde el propio Jenkins.
- SSH Agente Plugin.
- SSH plugin.
- Emailestex: Para poder enviar correos electrónicos.

➢ Configurar credenciales.

Jenkins tiene la opción de crear y guardar credenciales. Estas pueden ser desde passwords + username hasta claves privadas para conexiones SSH.

- Conexión a Bitbucket.

Una forma de conseguir esto es añadir en Bitbucket una contraseña de aplicación. Las contraseñas de aplicación son contraseñas sustitutas de una contraseña de usuario que se pueden utilizar para la integración de herramientas y así evitar introducir contraseñas reales. Además estas contraseñas se pueden configurar con diferentes permisos de lectura y escritura en diferentes campos.

Se creará una contraseña dedicada para la integración con Jenkins.

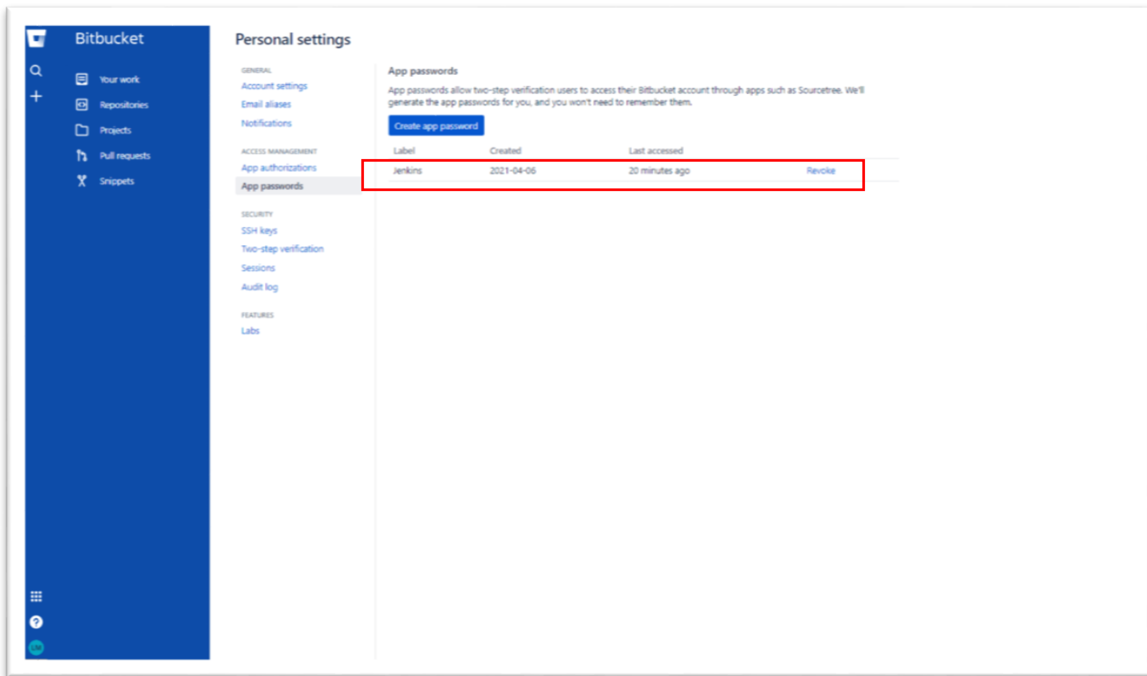


ILUSTRACIÓN 23. CONTRASEÑA DE APLICACIÓN DE BITBUCKET.

Se podrá entonces crear una nueva credencial en Jenkins que permita a la aplicación conectarse a Bitbucket de forma directa sin la necesidad de introducir contraseñas.

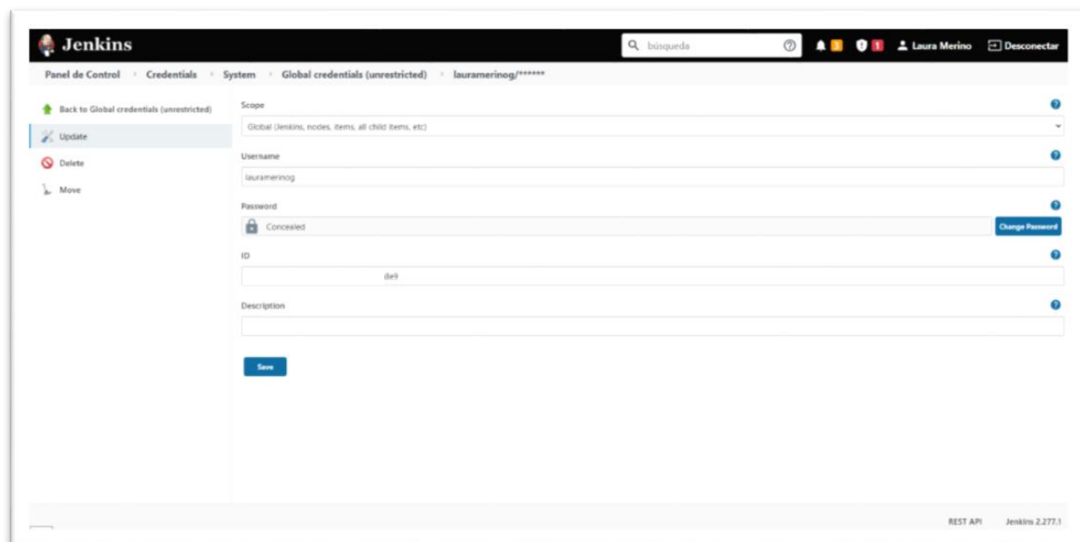


ILUSTRACIÓN 24. CREDENCIAL TIPO CONTRASEÑA EN JENKINS.

➢ **Conexión por SSH.**

Ciertos plugins de Jenkins para conexiones SSH vienen con una configuración de credenciales propias. Este es el caso del plugin Publish Over SSH. Para que la conexión sea posible se debe introducir la clave privada y los datos del sistema remoto en el menú de la configuración.

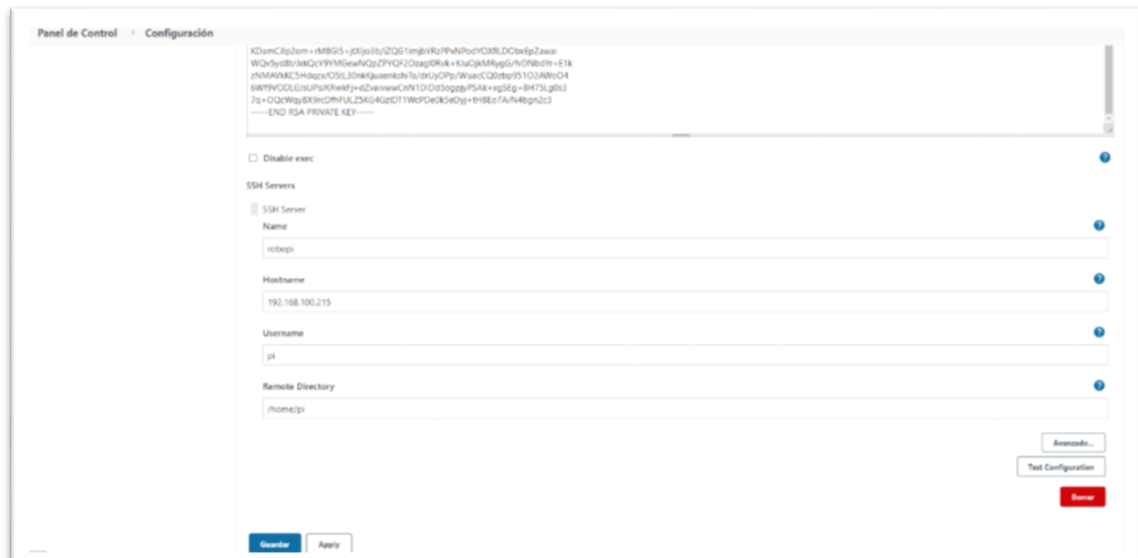


ILUSTRACIÓN 25. CONFIGURACIÓN CLAVE PRIVADA SSH.

- **Conexión con Docker.**

Docker y Jenkins se conectan mediante TCP, para ello se debe incluir una nueva nube donde la dirección sea tcp://<direcciónIP>:2375

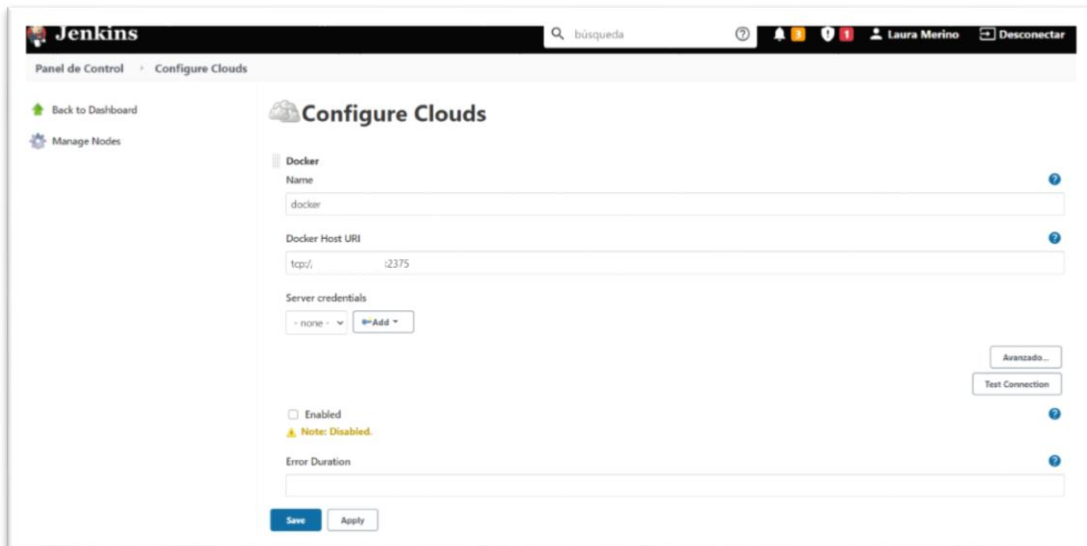


ILUSTRACIÓN 26. CONFIGURACIÓN DOCKER EN JENKINS

Creación de la canalización

Una vez cumplidas las condiciones previas, se está en disposición de crear la canalización para conseguir la validación automática de la aplicación.

Para crear un trabajo en Jenkins se selecciona *Panel de Control > Nueva Tarea.* En primer lugar se crea una tarea tipo *Folder.* De esta forma Jenkins permite agrupar las canalizaciones que tengan que ver con el mismo proyecto.

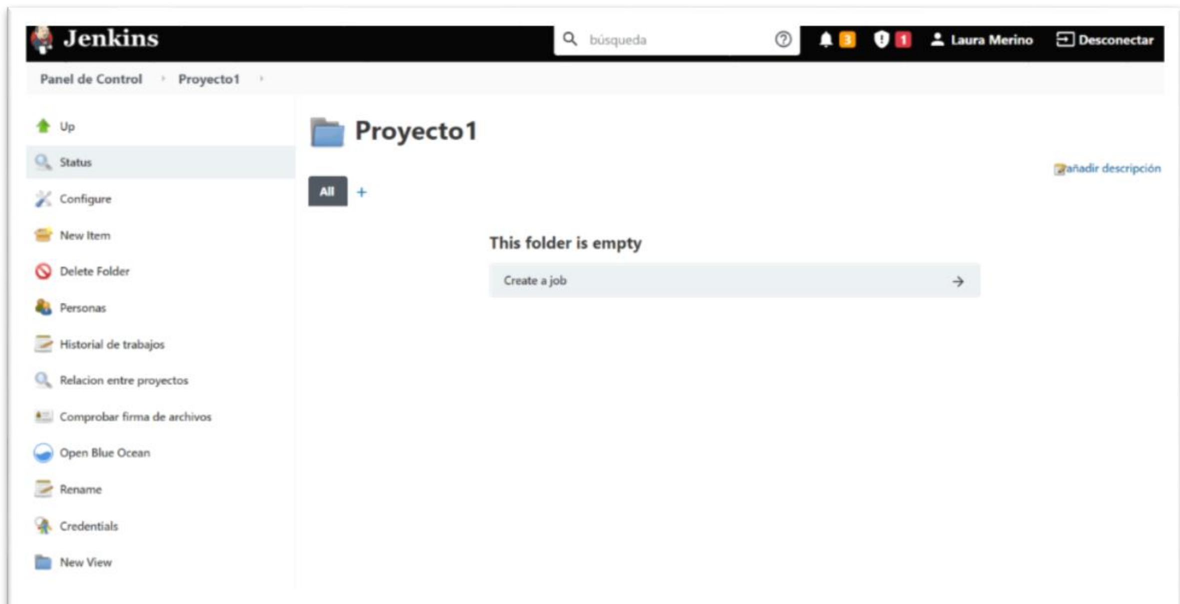


ILUSTRACIÓN 27 PROYECTO JENKINS.

Si ahora navegamos a *Nueva tarea* dentro de la carpeta *s*e accederá a una ventana donde se incluyen los tipos de trabajos que se pueden desarrollar. Los principales son: proyecto estilo libre y pipeline.

Proyecto estilo libre	Pipeline
Forma guida de crear un nuevo proyecto. Los pasos de la canalización se incluyen uno a uno mediante la interfaz y posteriormente es el propio Jenkins quien genera el código. Pensado para las primeras canalizaciones para familiarizarse con la herramienta.	El código se genera directamente. Esta forma de construcción otorga mayor precisión en la construcción de las tareas. Acaba convirtiéndose en la forma de construir proyectos más empleada por programadores. Las pipeline proporcionan un conjunto extensible de herramientas pata modelar canales de entrega complejos como código a través de la sintaxis del lenguaje específico de dominio (DSL) de Pipeline.

TABLA 10. COMPARACIÓN TIPOS DE PROYECTOS EN JENKINS.

1) Primeras configuraciones.

- General.

En este primera apartado se puede incluir una descripción de la tarea así como ciertas configuraciones iniciales. Se selecciona *Deshacer ejecuciones antiguas* que borrará los registros de compilaciones antiguas de este proyecto, desde salidas de consola hasta artefactos archivados.

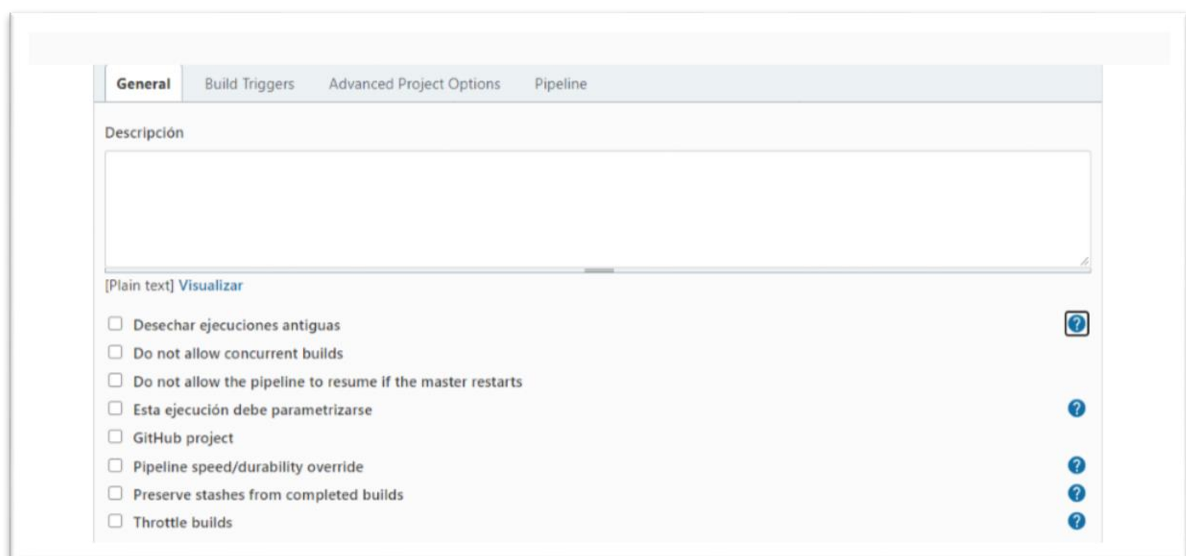


ILUSTRACIÓN 28. CONFIGURACIÓN GENERAL JENKINS.

- Build Trigger.

En esta etapa se configura el momento en que se lanza la ejecución de la pipeline. En este punto es donde se indicará a la herramienta que el lanzamiento se realice de forma automática.

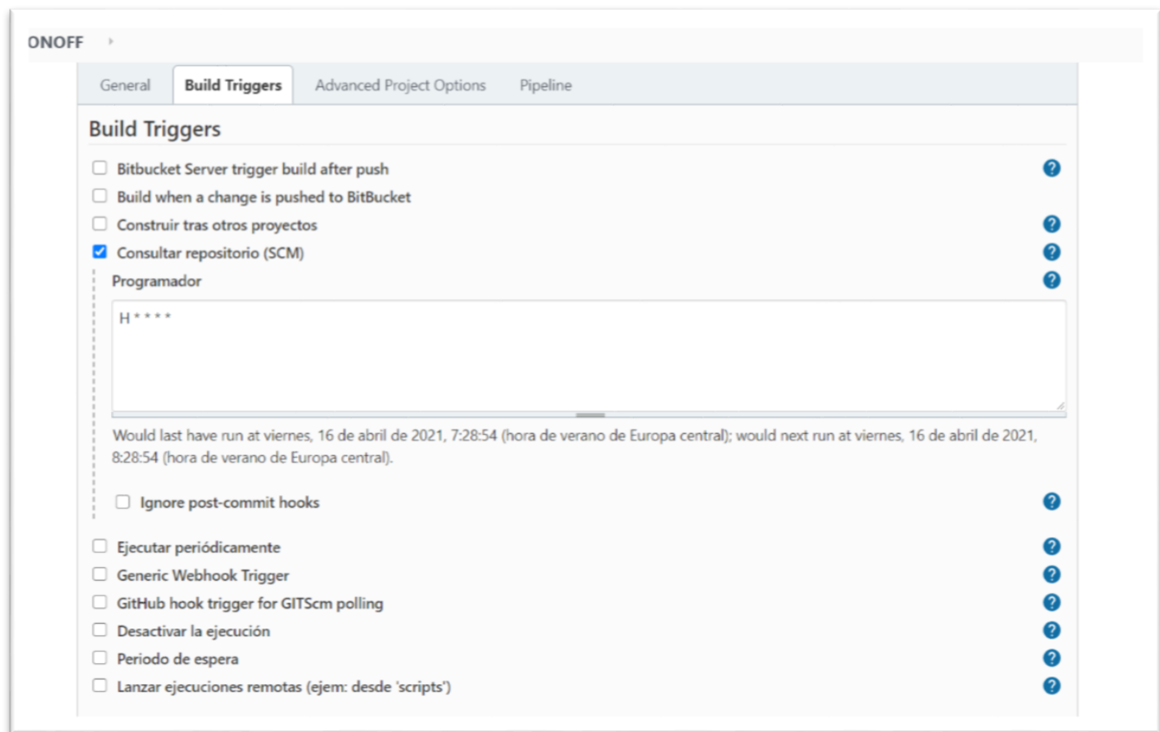


ILUSTRACIÓN 29. CONFIGURACIÓN TRIGGER JENKINS.

Entre ellas se selecciona *Consultar repositorio (SCM).* En el cuadro se precisa cada cuanto tiempo se consulta el repositorio. En este caso *H * * * ** significa cada hora y, por ejemplo, *H/15 * * * ** cada 15 minutos. Esto hará que Jenkins consulte el repositorio que se indiqué en el código de la pipeline y ejecute la tarea si ha habido nuevos cambios.

2) Pipeline.

De forma muy general, las pipelines cumplen la siguiente estructura.

```
pipeline {
   agent any
      stages{
            stage ('1'){
                  steps{

            }
      }
            stage ('2'){
                  steps{
                        script{

                  }
            }
      }
}
```

Stages	Indica a Jenkins una agrupación de estados.
Stage	Son una separación lógica de los steps. Son grupos de secuencias conceptuales distintas. Pueden incluir un nombre como, por ejemplo: repositorio, ejecutar_test… Estos nombre no deben contener espacios.
Steps	Indican a Jenkins una sola operación.

Script	Ejecutan un bloque de código escrito en *groovy* que pueden ser usados para escenarios no comunes en donde se necesita algún tipo de control de flujos.
Nodos	Los nodos son agrupaciones de tareas o steps que comparten un workspace. Los conjuntos de steps son añadidos a la cola de Jenkins para ser ejecutados cuando haya algún espacio libre entre los agentes de ejecución. Los agentes de ejecución pueden ser la misma máquina del servidor maestro de Jenkins o un Jenkins en otra máquina en modo esclavo dedicado a este propósito. Es importante reseñar que el directorio de trabajo (workspace) es compartido por los steps del nodo, de forma que steps de un nodo pueden acceder a ficheros/directorios generados por steps de ese mismo nodo. Por el contrario, un step de un nodo no puede acceder al workspace de otro nodo.
Agent	Indican donde se ejecutará la canalización que puede ser la misma máquina virtual. *Agent any* indica que la pipeline se ejecutará en cualquiera de los nodos.

TABLA 11. FORMATO PARA ESCRIBIR PIPELINES.

3) Desarrollo de la canalización.

En este apartado se realizará una explicación más detallada de los pasos a seguir para conseguir que una aplicación alojada en un repositorio sea testeada reportando los datos del test al propio Jenkins.

1º) Clonar repositorio

La propia herramienta Jenkins genera un workspace donde se clonan los archivos que se encuentren en la rama del repositorio indicado en las credenciales. Si no se indica rama, se selecciona por defecto la master.

El workspace con directorio */var/lib/Jenkins/<nombretarea>*tendrá un aspecto igual al repositorio remoto.

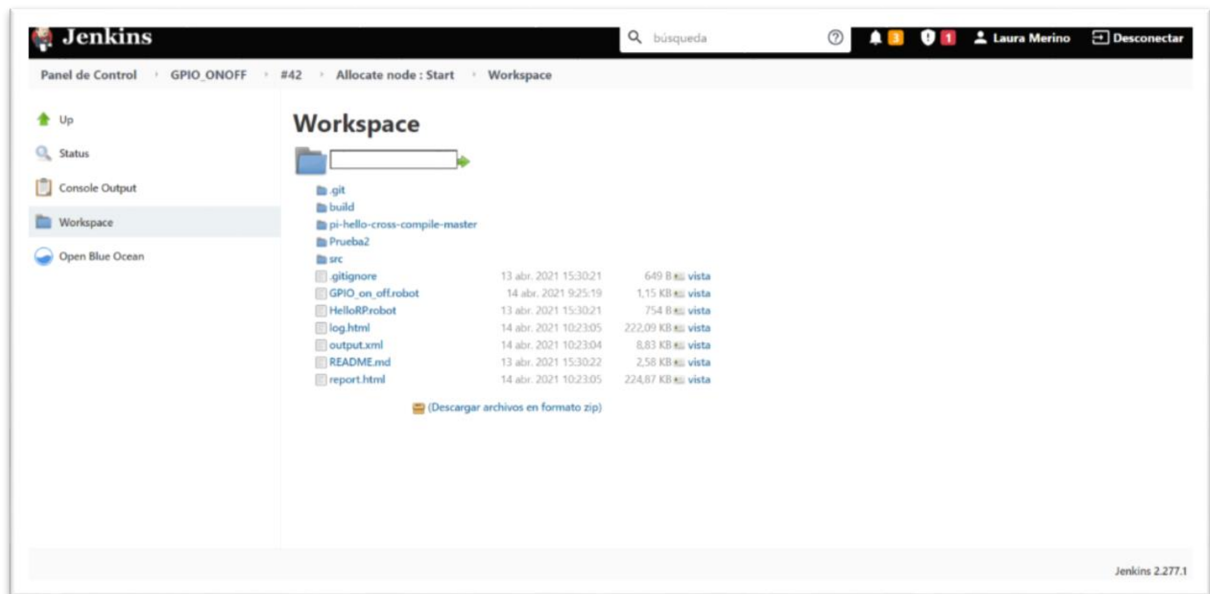


ILUSTRACIÓN 30. WORKSPACE DE JENKINS.

Para conseguir esto basta con generar un stage llamado Repositorio y en *Pipeline Syntax* incluir el enlace del repositorio y las credenciales creadas anteriormente.

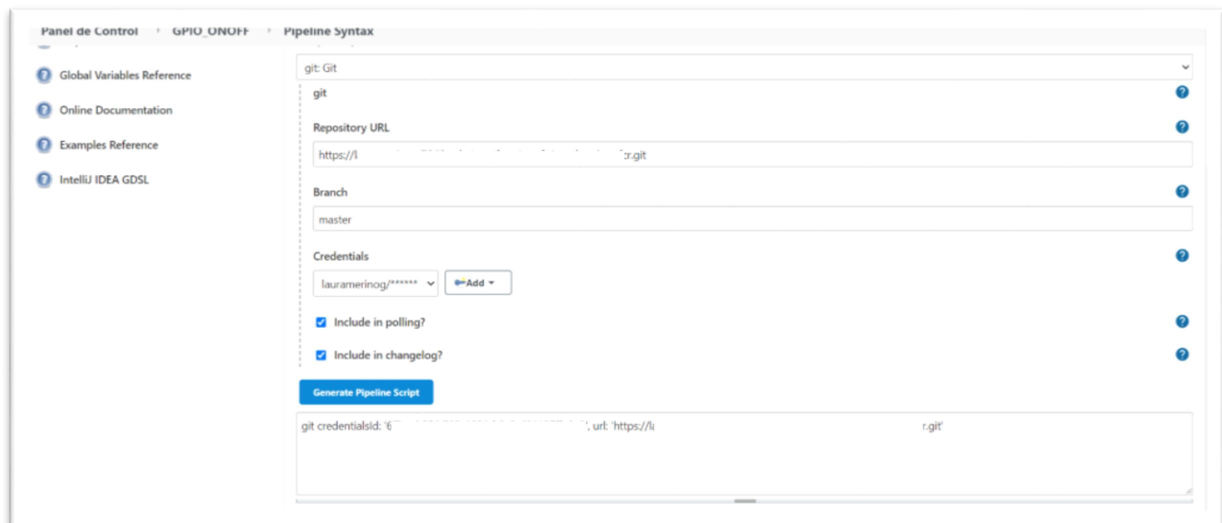


ILUSTRACIÓN 31. SELECCIONAR REPOSITORIO JENKINS.

De tal forma que el código creado para incluir en la pipeline sea el siguiente.

```
stage (Repositorio){
      steps{
            git
         credentialsId:'
            url:'https://                @bitbucket.org/            /
         repolmerinopfcr.git'
}
}
```

2º) Ejecutar Docker

Jenkins pipeline puede ejecutar comandos de shell donde podemos escribir el comendo Docker run y generar el archivo app necesario. Este archivo se almacena en el workspace.

```
stage (Docker){
steps{
        script{
   sh ' docker run -v /opt/sdks/sdk8p:/opt/sdks/sdk -v
   /var/lib/jenkins/workspace/GPIO_ONOFF:/home/proy sdk8p:last'

                        }

                }

                }
```

El comando *sh ' '* ejecuta un script de Shell Bourne, generalmente en un nodo Unix aceptando varias líneas.

Ejecutando el comando se le indica a Docker que el archivo llamado sdk8p debe ser tratado dentro de la imagen como sdk y que la dirección del workspace del proyecto se interpretará como la dirección /home/proy. Sdk8p:last indica que imagen se ejecuta.

3º) Enviar por ssh.

```
stage (Enviar_por_ssh){
  steps{
  script{
   sshPublisher(publishers:[sshPublisherDesc(configName:'root',
transfers:[sshTransfer(cleanRemote:false, excludes:'', execCommand:'',
execTimeout:120000, flatten:false, makeEmptyDirs:false,
noDefaultExcludes:false, patternSeparator:'[, ]+',
remoteDirectory:'.', remoteDirectorySDF:false, removePrefix:'',
sourceFiles:'build/app')], usePromotionTimestamp:false,
useWorkspaceInPromotion:false, verbose:false)])

   sshPublisher(publishers:[sshPublisherDesc(configName:'robopi',
transfers:[sshTransfer(cleanRemote:false, excludes:'', execCommand:'',
execTimeout:120000, flatten:false, makeEmptyDirs:false,
noDefaultExcludes:false, patternSeparator:'[, ]+',
remoteDirectory:'.', remoteDirectorySDF:false, removePrefix:'',
sourceFiles:'GPIO_on_off.robot')], usePromotionTimestamp:false,
useWorkspaceInPromotion:false, verbose:false)])


                }
          }
}
```

4º) Ejecutar comandos por ssh

Para esto se emplea el comando sshCommand. A diferencia del comando anterior, que funcionaba mediante unas credenciales creadas, aquí se debe definir una variable llamada remote donde se indican las características del dispositivo a conectar: name, username, host y password.

Estos comandos deben ser ejecutados dentro de un nodo fuera de la pipeline.

```
node {
  def remote=[:]
  remote.name ='pi'
  remote.host ='192.          '
  remote.allowAnyHosts =true
  remote.password ='          '
  remote.user ='pi'


  stage('test'){
     sshCommand remote: remote, command:"robot
   /home/pi/GPIO_on_off.robot"
     robot archiveDirName:'robot-plugin', outputPath:'.',
   overwriteXAxisLabel:''
     sshCommand remote: remote, command:"rm /home/pi/output.xml
   /home/pi/log.html /home/pi/report.html"
     sshCommand remote: remote, command:"rm
   /home/pi/GPIO_on_off.robot"

}
}
```

5º) Resultados Robot Framework.

El plugin instalado de Robot Framework es capaz de transferir, tanto los resultados, como los archivos generados por Robot Framework a Jenkins para que posteriormente puedan ser eliminados de la Raspberry sin perder los resultados.

```
robot archiveDirName:'robot-plugin', outputPath:'.',overwriteXAxisLabel:''
```

Para que puedan ser abiertos es necesario hacer una serie de ajuste y ejecutar el siguiente comando en la consola de Jenkins:

```
System.setProperty("hudson.model.DirectoryBrowserSupport.CSP","sandbo
x allow-scripts; default-src 'none'; img-src 'self' data: ; style-src
'self''unsafe-inline' data: ; script-src 'self''unsafe-inline''unsafe-
eval' ;")
```

6º) Borrar aplicación de la placa.

En este paso se eliminará de la placa el archivo binario ejecutado para que esta vuelva al estado inicial de la canalización.

Para ello se crea un nuevo nodo y, al igual que con la Raspberry, se configura la variable remote para conectarse a la placa de prueba y ejecutar el comando que borrará la aplicación.

```
node {
    def remote=[:]
    remote.name ='root'
    remote.host ='192.           '
    remote.allowAnyHosts =true
    remote.user ='root'


stage('Borrar_app'){
sshCommand remote: remote, command:"rm -rf /home/root/build"

 }

 }
```

7º) Enviar resultados por correo.

El último paso que debe realizar la canalización es enviar los reportes de las pruebas por correo electrónico a los desarrolladores. De esta forma, Jenkins podrá ejecutarse de manera independiente llegando a las personas solo los resultados de la canalización.

Para ello se emplea el siguiente paso en la canalización con la cual, dependiendo del resultado, se enviará al correo señalado un mensaje indicando el resultado de la ejecución y el archivo de reportes que genera el propio Robot Framework.

```
post{

success {

      emailext to:'correo',
      subjet:"Build: ${env.JOB_NAME} - Sucess",
      attachlog:true,
      attachmentsPattern:'**/report.html'
      body:"Job Sucess - \"${env.JOB_NAME}\"build:
      ${env.BUILD_NUMBER}\n\n"
}
failure {

      emailext to:'correo',
      subjet:"Build: ${env.JOB_NAME} - Failed",
      attachlog:true,
      attachmentsPattern:'**/report.html'
      body:"Job Failed - \"${env.JOB_NAME}\"build:
        ${env.BUILD_NUMBER}\n\n"


}
```

6. Ejemplo de prueba y Resultados.

En este aparatado se va a realizar un estudio de los resultados conseguidos así como demostrar el correcto funcionamiento del mismo. En primer lugar se definirá la aplicación a probar.

6.1. Requisito de la aplicación.

Para validar el diseño creado se planteó generar una aplicación simple capaz de encender y apagar un pin de una placa.

En vista de esto, los requisitos que se crearon y debieron validar fueron los siguientes:

ID	Sumario	Cuerpo del Requisito	Pr	Caso de Uso	Test
RSF-1.0.0	Empieza gpio off	El pin debe empezar apagado.	Alta	Cuando el usuario ejecuta la aplicación el pin deberá estar apagado y aparecerá en la pantalla de comandos *GPIO cambia a off.*	T.0001
RSF-2.0.0	Cambia gpio on	El pin debe cambiar a on cuando se ejecute un comando	Alta	Cuando el usuario ejecuta *gpio on* el pin deberá encenderse y aparecerá en la pantalla de comandos *GPIO cambia a on.*	T.0001
RSF-3.0.0	Cambia gpio off	En pin debe cambiar a off cuando se ejecuta un comando.	Alta	Cuando el usuario ejecuta *gpio off* el pin deberá apagarse y aparecerá en la pantalla de comandos *GPIO cambia a off.*	T.0001

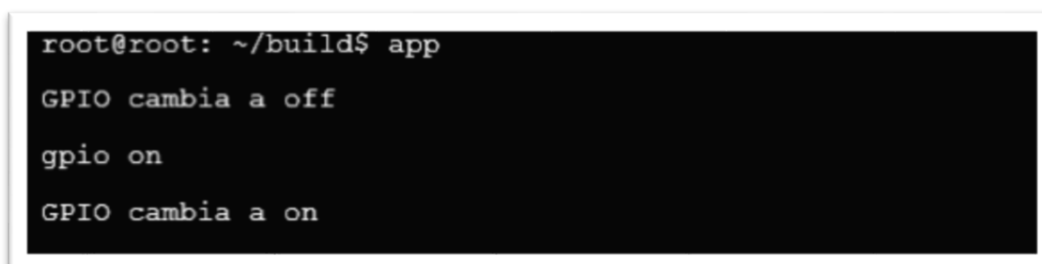
TABLA 12. REQUISITOS DE LA APLICACIÓN.

Aunque sea una aplicación muy simple, esta funcionalidad podría darse en un sistema más grande que contase con un sistema de alarmas led cuyo requisito de funcionamiento debería darse en la primera versión de código, es decir, se debería comprobar cómo cada vez que se produce un nuevo código (podría ser diariamente) cómo esta funcionalidad sigue funcionando según lo marcado.

6.1.1. Ejemplo de prueba creado

El programa a testear se trata de una aplicación que se ejecutará en una placa donde se instala un SDK (Kit de Desarrollo Software) con sistema operativo Linux dedicada a encender y apagar un pin GPIO.

Cuando se escribe en la consola de comando *gpio on* el pin x de la placa se enciende y posteriormente tras escribir *gpio off* se apagará escribiendo en la consola GPIO cambia a off.



6.1.2. Compilación de aplicación

Como se ha explicado, la función de Docker es compilar para así generar un binario de la aplicación que, posteriormente la placa pueda ejecutar.

Para ello se emplean las imágenes de Docker. Las imágenes son capturas instantáneas de un contenedor, donde un contenedor se entiende como un entorno donde una aplicación puede ser ejecutada ya que en ella se encuentran todas aquellas librerías, sistema operativo y demás artefactos que necesitará la aplicación. Los contenedores de Docker se pueden asemejar a una máquina virtual donde poder ejecutar aplicaciones comandos, etc.

Las imágenes de Docker se crean a través de un archivo llamado Dockerfile. Para conseguir compilar la aplicación deseada se redacta lo siguiente en el archivo:

```
FROM amd64/ubuntu

RUN apt update
RUN apt install make
CMD /home/proy/src/MakeEntrega /home/proy/build
/opt/sdks/sdk/activate /home/proy/src
```

Donde partiendo de una imagen del sistema operativo amd64/Ubuntu, que tiene la placa, se genera la imagen que se desea.

Los pasos RUN se usan para correr un comando dentro del dockefile. Se utilizan cuando se está construyendo una imagen personalizada para realizar una acción, creando una capa nueva

El comando CMD se encarga de pasar valores por defecto a un contenedor. Entre estos valores se pueden introducir ejecutables.

El comando a ejecutar es el archivo MakeEntrega con el que se compila la aplicación de c++:

```
BUILDDIR=$1
SDK=$2
SRCDIR=$3

echo BUILDDIR = $BUILDDIR
echo SDK = $SDK
echo SRCDIR = $SRCDIR


rm -rf ${BUILDDIR}
mkdir ${BUILDDIR}
. ${SDK}
cd ${BUILDDIR}
cmake ${SRCDIR}/applinux/app
make
```

De esta forma se consigue compilar el archivo que se encuentra en el segundo argumento del comando de docker creando el archivo binario en el directorio que se encuentra en el primer argumento.

6.1.3. Pruebas.

La prueba a realizar, que valida los tres requisitos es la siguiente:

<table>
<tr><th>T.0001</th><th>GPIO_on_off</th><th>V 01</th></tr>
<tr><td colspan="3">Requisitos validados:</td></tr>
<tr><td colspan="3">• RSF-1.0.0
• RSF-2.0.0
• RSF-3.0.0</td></tr>
<tr><td colspan="3">Condiciones iniciales:</td></tr>
<tr><td colspan="3">• La placa y la Raspberry deben estar encendidas y conectadas a la red.
• La placa no debe contener el programa a probar.
• La Raspberry no debe contener el test de Robot Framework.
• El pin GPIO 6 de Raspberry debe estar configurado como entrada.
• Conectar el pin de salida y la tierra de la placa al pin GPIO 6 y la tierra de las Raspberry.</td></tr>
<tr><td colspan="3">Secuencia.</td></tr>
<tr><td colspan="2">Paso 1: Subir versión a validar a la rama master del repositorio para dar comienzo al proceso automático de test.</td><td>OK/NOK</td></tr>
<tr><td colspan="3">Descripción del resultado:</td></tr>
<tr><td colspan="3">Se comprueba los reportes de Robot Framework observando si han sido pasados correctamente.</td></tr>
</table>

TABLA 13. TEST DE REQUISITOS.

6.1.4. Creación de pruebas en Robot Framework.

Para la escritura de test de Robot Framework se emplea el editor RIDE. Mediante este editor, la redacción de proyectos de tests se convierte en algo sencillo e intuitivo.

Robot Framework funciona en base a librerías de keywords y variables. En el caso del presente diseño, se necesitaron dos librerías para cumplir con ciertas funcionalidades:

- SSHLibrary. En este caso, Rasberry necesita comunicarse con la placa a testear. Al tener que ejecutar comandos en ella, la solución planteada fue seguir conexiones SSH.
- OperatingSystem. Esta librería aporta el comando Grep File el cual es muy útil cuando se desea leer frases de consola y archivos.

También el propio usuario puede generar sus propias palabras clave si lo ve oportuno. En este caso se crea una keyword llamada *Open Conecction And Log In.*

```
*** Keywords ***
Open Connection And Log In

       Open Connection     ${HOST}
           Login     ${USERNAME}
```

Dentro de un proyecto se pueden incluir todos los test que sean necesarios. Estos test comparten las librerías, keywords y variables. Cada test cuenta con una ventana como la siguiente donde se puede dar comienzo a la redacción de la prueba.

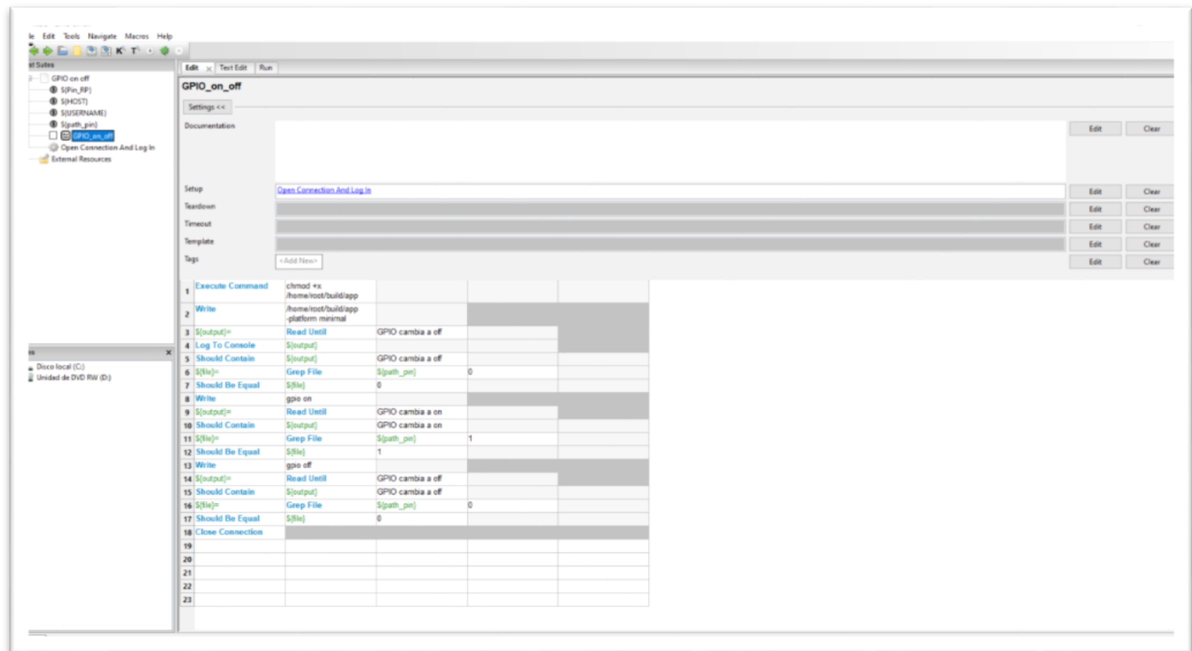


ILUSTRACIÓN 32. INTERFAZ RIDE

Donde:

Documentation.	Apartado para escribir aclaraciones o todo lo que sea conveniente.
Setup	Primera acción que se realiza nada más comenzar el test
Teardown	Última acción que se realiza al finalizar el test.
Tags	Etiquetas que se incluyen en el test que luego se pueden emplear para ejecutar los test que solo presenten dicha etiqueta.
Test Case	En esta plantilla de celdas se van incluyendo los pasos para ejecutar el test.

TABLA 14. CARACTERÍSTICAS TEST.

La forma de programar que presenta Robot Framework permite seguir el orden lógico que realizaría un testar para validar la funcionalidad. Estos pasos son los siguientes:

1. Ejecutar un comando de la aplicación.
2. Comprobar si lo que devuelve la consola de la aplicación encaja con lo deseado.
3. Comprobar el estado del pin de la Raspberry para comprobar si el comando de salida en consola es el correcto gracias a una función Grep.

De esta forma, el código del test de prueba es el siguiente:

```
*** Settings ***

Library             SSHLibrary
Library             OperatingSystem

*** Variables ***

${Pin_RP}     6
${HOST}       192.███.███.███
${USERNAME}   ████████████████
${path_pin}   /sys/class/gpio/gpio6/value

*** Test Cases ***

GPIO_on_off

[Setup]     Open Connection And Log In

    Execute Command     chmod +x /home/root/build/app

    Write     /home/root/build/app -platform minimal

    ${output}=     Read Until     GPIO cambia a off

    Should Contain     ${output}     GPIO cambia a off

    ${file}=     Grep File     ${path_pin}    0

    Should Be Equal     ${file} 0

    Write     gpio on

    ${output}=     Read Until     GPIO cambia a on

    Should Contain     ${output}     GPIO cambia a on

    ${file}=     Grep File     ${path_pin}    1

    Should Be Equal     ${file}          1

    Write     gpio off

    ${output}=     Read Until     GPIO cambia a off

    Should Contain     ${output}     GPIO cambia a off

    ${file}=     Grep File     ${path_pin}    0

    Should Be Equal     ${file}          0

  Close Connection


*** Keywords ***
Open Connection And Log In

    Open Connection     ${HOST}
    Login     ${USERNAME}
```

6.2. Resultados del diseño.

El resultado del diseño planteado en este trabajo será entonces visualizar el flujo de trabajo de la pipeline creada así como demostrar que ciertamente supone una ventaja respecto a un planteamiento maual.

Una vez subido un cambio a la rama master del repositorio, la canalización es lanzada llegando al desarrollador los siguientes reportes desde Jenkins:

ILUSTRACIÓN 33. LOG DE PRUEBA ROBOT FRAMEWORK

ILUSTRACIÓN 34. REPORTE PRUEBA ROBOT FRAMEWORK

Lo que confirma que los test han sido pasados con éxito. Si se fuese al menú de ejecución en Jenkins, se observaría algo así:

ILUSTRACIÓN 35. MENÚ PRINCIPAL DE UNA EJECUCIÓN DE CANALIZACIÓN.

Donde aparece un resumen con: el trigger que lanzó la canalización, el repositorio del que se clona el código y los resultados de los test, así como fecha y hora de la ejecución.

El proceso se realiza de forma completamente automática, lo único que debe realizar el desarrollador es lanzar la canalización y esperar los resultados de las pruebas con informes perfectamente detallados.

Esto lleva consigo una serie de ventajas muy importantes con respecto al sistema de validación que será presentado a continuación.

6.3. Validación manual y comparación.

Con el fin de poner de manifiesto la mejora que supone este nuevo proceso, se explicará el diseño que se obtendría obviando todos los conceptos presentados en este trabajo.

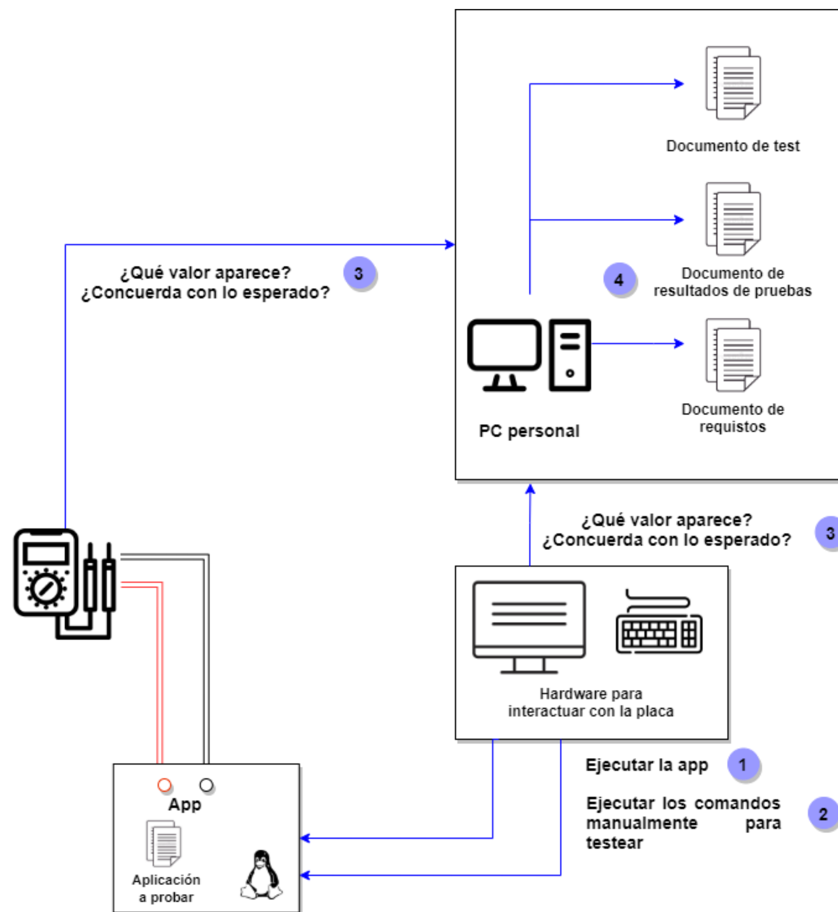


ILUSTRACIÓN 36. FLUJO DE VALIDACIÓN TEST DE PARTIDA.

Destacar que las flechas no tienen por qué representar conexiones directas entre dispositivos sino flujo de información que sí puede ir complementado por una conexión real.

Los pasos de este planteamiento son los siguientes:

1. Ejecutar la app de forma manual en la placa de prueba.
2. Gracias a los periféricos conectados a la placa se ejecutan los comandos. Aunque se pueda ejecutar esto desde el mismo PC, no tendría porque ya que el hardware donde se encuentra la aplicación podría estar completamente separado del ordenador.
3. Tomar los resultados de forma manual según los casos de prueba planteados.
4. Redactar los resultados observados los cuales solo se encuentran en el PC del que testea la aplicación.

Este diseño sería, entonces, el que se emplearía de forma tradicional para testear un requisito de aplicación.

Aunque aparentemente parezca una solución sencilla y razonable para testear los requisitos planteados, esta idea presenta ciertas desventajas que hacen del planteamiento algo completamente ineficiente:

- El diseño solo sirve para un test, igual siquiera para un requisito. La aplicación del proyecto a validar es muy sencilla, por lo que esta desventaja no parece algo crítico, pero en diseños más complejos con multitud de requisitos y pruebas, con este planteamiento, se necesitaría montar un nuevo hardware y software para ejecutar un único test.

- Es ineficiente en tiempo. El programador que realiza el test debe ejecutar todo él mismo en persona, tomar nota, pasar los reportes a su ordenador… Nada está automatizado.

- El código solo se encuentra en la placa y el PC. No hay control de versiones ni repositorio. Todo el desarrollo borrado desaparece por completo sin llegar a conocer que cambios se han ido realizando a lo largo de su creación.

- Los errores humanos pueden aparecer en cualquier momento. Por la propia naturaleza de las pruebas manuales, cada prueba se realiza de diferente forma pudiendo llegar a cometes errores en la toma de resultados.

- Los documentos redactados están únicamente en el ordenador de la persona que ejecuta el test. Por lo general, los proyectos software se realizan entre varios desarrolladores por lo que esta acción corta la comunicación por completo entre ellos.

- No hay comunicación entre diferentes desarrolladores en cuanto al estado del proyecto. La información no se comparte de forma constante ni metódica.

- Una vez que se ha montado todo aquello que es necesario para realizar el test, surge una problemática en cuanto al número y tipo de pruebas que pueden realizarse.

¿Cómo se pueden probar casos "extremos"? ¿Cómo se podría ejecutar ciertos test como , por ejemplo, aquellos que requiera periodos de encendido y apagado de ms? ¿Y si las operaciones deben realizarse durante largos periodos de tiempo? Estos casos de prueba, los cuales son en gran medida los que sacan a la luz los fallos del sistema, en muchas ocasiones no pueden llegar a ser realizados de forma manual.

- La aplicación se debe encontrar en el placa. Aunque esto, en principio, no parezca una desventaja, contar con un único archivo binario para ejecutar en cualquier placa "limpia" es algo que podría sonar más conveniente si, por ejemplo, se quieren testear varias placas sin preocuparse de si en ella se encuentran los archivos y programas necesarios para ello.

En vita de este proceso, se pueden remarcar las grandes soluciones que aporta el nuevo modelo planteado de Integración Continua.

Soluciones	
Diseño de partida.	**Nuevo diseño.**
Diseño único	Este nuevo planteamiento permite recrear varios escenarios de test con una Raspberry donde RF es el ejecutor de multitud de test al contrario que en el caso anterior, donde todo el hardware incluido solo serviría para testear el valor de un pin. El uso de una Raspberry permite tener un hardware con un software especializado para testeo portátil y todo lo versátil que una aplicación de test permite.
Ineficiencia	Como se ha recalcado al principio, realizar test a mano conlleva un tiempo exageradamente grande si se compara con las capacidades que la automatización de todo este nuevo proceso permite. En esta demo, proceso de test era capaz de realizarse en **1,6 segundos** y la canalización completa en aproximadamente **10 segundos**.

	Esto se compara con todo el tiempo que emplearía un desarrollador en compilar la aplicación, ejecutarla, saber qué y cómo debe testear, redactar los reportes, etc. Una vez programada la canalización, esta será capaz de realizar todas esas tareas automáticamente en el menor tiempo posible.
Errores	Los test los realiza una máquina. Es por esto por lo que todos los test se realizan y comprueban de la misma forma desapareciendo a su vez los errores humanos.
Documentos no compartidos	Con este nuevo planteamiento todos los documentos que necesiten ser compartidos deberá ser subirlos a Confluence para llevar un seguimiento de los requerimientos de forma compartida con todos los desarrolladores que participan en el proyecto.
Limitaciones de las pruebas.	El poder de las herramientas de prueba es su versatilidad. Estas herramientas pueden llegar a testear casos extremos, como por ejemplo trabajar durante horas realizando test a ciertas funcionalidades que necesitan probarse a lo largo del tiempo.
Aplicaciones en la placa.	Contar con un archivo binario que pueda ser ejecutado en cualquier placa permite testear grandes cantidades de hardware sin comprobar si tiene las dependencias necesarias para ello. Además de ello, cada vez que se quiera testear se realizará con el código alojado en Bitbucket el cual no entrará en conflicto con demás código que pudiese haber permanecido en el hardware.

TABLA 15. COMPARACIÓN DE RESULTADO

7. Conclusiones.

El método presentado en este trabajo para conseguir lidiar con una de las problemáticas más grandes que presentan los proyectos software pretende ser la base para conseguir una mejora sustancial en el desarrollo eficiente y automático de software correctamente validado.

En vista de los objetivos, tanto del proyecto como de la empresa, se realizaron diferentes estudios para comprender como se podían mejorar estos procesos asimilando nuevos conceptos teóricos del mundo de las metodologías ágiles (DevOps), marcando pautas para gestionar requisitos y casos de prueba y finalmente generando un diseño para conseguir así la automatización total de las pruebas de validación.

Tras los análisis y resultados se llegaron a una serie de importantes conclusiones principales:

- Las nuevas metodologías del mundo de la validación software deben ser conocidas y asimiladas por todos los desarrolladores de un proyecto software. Comprender los métodos y las ventajas que estos ofrecen ayudará a impulsar el cambio hacia formas de trabajo más eficientes.
  Las metodologías ágiles han cogido una gran fuerza en los últimos años y no se puede obviar como su implementación en las empresas comienza a ser una obligación. Empezar a replantearse el modo de trabajo en empresas del sector es una tarea que debe darse cuanto antes.
  Pero este proceso no debe ocurrir de la noche a la mañana, plantearse un objetivo futuro claro y conseguir dar pasos cortos pero firmes hará que el desarrollo se dirija de forma natural hacia nuevos procesos adaptados a la empresa y proyecto en particular para así mejorar la eficiencia y calidad global de los desarrollos software.

- La gestión de requisitos es una tarea compleja, especialmente para empresas que trabaja con la creación de diseños complejos y no tiene aún capacidad para integrar programas de gestión de requisitos tan complejos como puede ser la famosa herramienta de gestión DOORS®, herramienta que emplean las grandes empresas como American Airlines, donde los requisitos de los diseños se salen por completo de las capacidades humanas para la gestión. Es por esto por lo que comenzar a plantear

la gestión de los requisitos desde la correcta redacción, formato y validación parece algo más razonable.

Tras esto la gestión de los requisitos puede llevarse a cabo por herramientas secundarias como Jira, que sin ser este se propósito principal, esta pueda ayudar a una mejor comunicación y gestión de requisitos.

- La creación de un diseño capaz de automatizar el proceso de prueba puede ser el comienzo para introducir los conceptos de DevOps e implementar un proceso completo de Integración Continua. Mediante este diseño se colocan los primeros cimientos hacia una nuevo proceso de validación. La propia estructura del diseño obliga a los desarrolladores a testear su código y mantener activo los repositorios de código.

Finalmente señalar como los resultados y los estudios realizados en el tema, así como las ventajas que las metodologías planteadas proporcionan a los desarrollos software, hicieron posible la adopción de estas prácticas en proyectos reales en la empresa donde se trabajó, mejorando especialmente en temas de:

- Validación automática con Robot Framework.
- Creación de canalizaciones para validar el código del repositorio mediante Jenkins.

8. Líneas futuras.

La principal línea futura del presente trabajo es conseguir que acabe siendo la base para que, en proyectos futuros, se consiga desarrollar una validación de software automática lo más eficiente posible.

La versatilidad de las herramientas seleccionadas permite que los diseños se amplíen todo lo que se vea conveniente.

Pero, siendo más concretos, se podría ampliar el trabajo en los siguientes apartados:

- Estudiar los estándares y normas que hacen referencia a estos temas y plantearse adoptar alguno en la empresa.
- Estudiar que herramientas de gestión de requisitos cumple con el compromiso capacidades/facilidad de uso para así dar un paso más en el proceso de gestión de requerimientos software.
- Existen programas capaces de testear código como son SonarQube. Esta herramienta es capaz de testear código fuente con el fin de mejorar la calidad de un programa. Integrar en la canalización de Jenkins un apartado dedicado a testear el código y así desenmascara: código duplicado, código muerto, bugs, complejidad ciclomática, cobertura de código... sería el primer paso a dar para implementar el proceso de pruebas unitarias en el ciclo de vida del proyecto.

9. Referencias.

[1] Brady Hill. RobotFramework for Embedded Software Testing. *Robot Framework* (2019) [Online] https://www.youtube.com/watch?v=0q4-AjqpO9M

[2] Eficode. Developing embedded software with DevOps. [Online] https://www.eficode.com/guides/devops-embedded

[3] Quality management and quality assurance, ISO 8402 (1986)

[4] Quality management systems. ISO/IEC 9000 (2015)

[5] SOMMERVILLE, IAN, Software Engineering 07 Edition, 2005

[6] Information Technology / Software Life Cycle Processes. ISO/IEC 12207 (2017).

[7] Luisan (2019) ¿Qué son las metodologías ágiles? [Online] https://www.luisan.net/blog/transformacion-digital/que-son-las-metodologias-agiles

[8] Wikipedia. DevOps. [Online] https://es.wikipedia.org/wiki/DevOps

[9]Jesús Angulo (2018). Entendiendo DevOps en 5 minutos. [Online] https://www.autentia.com/2018/08/17/entendiendo-devops-en-5-minutos/

[10] Martin Fowler (2006). Continuous Integration. [Online] http://www.martinfowler.com/articles/continuousIntegration.html (abril 2021)

[11] Justin Ellingwood (2020). Introducción a practicas recomendadas de CI/CD. [Online] https://www.digitalocean.com/community/tutorials/an-introduction-to-ci-cd-best- practices-es

[12] Herramienta para Implementar LEL y Escenar. [Online]

http://sedici.unlp.edu.ar/bitstream/handle/10915/4057/2_-
_Ingenier%C3%ADa_de_requerimientos.pdf?sequence=4

[13] Recommended Practice for Software Requirements Specifications. IEEE 830 (1998)

[14] Ivar Jacobson, Ian Spence, Kurt Bittner (2013). Use Cases 2.0.

[15] Manuel Cillero (2020). Pruebas de integración. [Online]
https://www.autentia.com/2018/08/17/entendiendo-devops-en-5-minutos/

[16] Standard to Software Test Documentation. IEEE 29119-3 (2013)

[17] Atlassian Tutorials. Qué es el control de versions. [Online]
https://www.atlassian.com/es/git/tutorials/what-is-version-control

[18] Atlassian Tutorials. Flujo de trabajo de Gitflow. [Online]
https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow