Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingeniería de Software

Aplicación de metodologías ágiles en un entorno de trabajo distribuido y el rol del asegurador de calidad

Autor:
D. Luis Miguel Gutiérrez Mozo

Tutor:
Dr. César Llamas Bello

Resumen

Este documento recopila la información que he podido obtener acerca del flujo de trabajo de los equipos de desarrollo de software en la actualidad durante mi estancia, hasta ahora de un año y cuatro meses de duración, en una consultora multinacional. El motor principal detrás de estos equipos son las metodologías ágiles. Se expondrá una introducción a las mismas y los principios que estas representan, seguido de una mirada al pasado a otras técnicas y modelos de desarrollo de software en empresas que ya no se aplican con la misma frecuencia, sobre todo en proyectos de gran calibre a largo plazo. Tras esto, se mostrarán los mecanismos y herramientas con los que se ha conseguido agilizar los proyectos software alrededor del mundo, junto con el porqué de esta necesidad.

Posteriormente, se desarrolla la experiencia adquirida en uno de los campos que más ha crecido en la industria en los últimos años, las pruebas de software. Se explicarán los distintos tipos de pruebas existentes y la forma en la que se realizan dentro del marco de los proyectos ágiles.

Abstract

This document contains the information that I managed to obtain regarding the workflow of the software development teams nowadays during my stay at a international consulting corporation for the last sixteen months. The main engine that moves these teams is Agile. An introduction to Agile methodologies and the principles that they stand for will be explained, followed by a glimpse to the models and techniques used in the past by the corporations, no longer used with the same frequency, specially in long-term projects. After that, the mechanisms and tools used to increase the agility of the software projects all around the world will be shown, together with the need of this change.

Afterwards, the experience I got about one of the subjects that has grew the most over the last year in the industry, software testing, will be described. The different types of tests and the way those are crafted inside of the scope of Agile projects.

Contenido

1. Contexto y motivación

La idea de realizar este proyecto viene de tres grandes influencias. La primera, la utilización de manera implícita de las herramientas y metodologías ágiles que he podido aprender durante las prácticas en empresa y que aún se hace necesaria en la actualidad, además de la intención de conocer mejor estas técnicas debido a su auge y a la productividad que traen consigo. La segunda, la asignatura Planificación y Gestión de Proyectos, la cual fue impartida entre septiembre y enero del curso 19-20, que sentaba las bases del correcto desempeño de las funciones de un gestor, y que también dedicaba parte de su temario a introducir a las metodologías ágiles y algunas herramientas utilizadas para este propósito. Finalmente, la tercera, la oferta del profesor César Llamas Bello de un trabajo de fin de grado similar en el portal web que gestiona la facultad de Ingeniería Informática de Valladolid para este fin. Conversé con el señor Llamas acerca del posible enfoque que podría darle desde la perspectiva del aseguramiento de la calidad (QA) que, como la gran mayoría de trabajadores del sector de la informática, se ve igualmente bajo la necesidad de aplicar estas metodologías que rigen el día a día de cualquier proyecto de tamaño considerable, ya sea realizado de manera local o de manera remota y distribuida, llevado a cabo por integrantes de distintos lugares, que deben coordinarse con la misma eficiencia que lo harían si compartieran oficina.

1.1. Objetivos y alcance

El objetivo principal de este documento es contener la información relativa a las metodologías ágiles, sus posibles usos y algunas de las formas más habituales de aplicarlas a cualquier proyecto, así como mostrar varias de las herramientas más comúnmente utilizadas actualmente para conseguir poder hacerlo correctamente y de la manera más efectiva posible.
Dado que estas herramientas son utilizadas ampliamente en todo tipo de proyectos por todas las empresas que buscan proporcionar un producto de manera continuada, o cuyo desarrollo no se contemple en un tiempo reducido, este documento puede ser útil para cualquier persona relacionada con el mundo de la informática, más aún si su fin es adentrarse en la realización de proyectos de cualquier tipo, ya sea por su propia cuenta, dado que las metodologías ágiles también pueden aplicarse en equipos pequeños (o incluso equipos formados por una persona) o ya sea bajo el foco de otra empresa para la cual participe en el desarrollo de dicho producto, por el conocimiento que obtendrá de la forma que estas tienen de trabajar.

2. Las metodologías ágiles

2.1. Definición

Lo primero que debemos abordar es la definición de metodologías ágiles. Son aquellas pautas que cambiaron el paradigma sobre las prioridades a la hora de desarrollar software. Actualmente, para aplicar estas pautas, se emplean protocolos que nos permiten que un volumen de trabajo definido sea repartido entre los integrantes del proyecto de modo que se cumplan unos plazos en los que se produzca una entrega, siendo cada una de ellas completa en sí misma y de un valor calculable, pero a la vez pudiendo no representar el producto final deseado, debido a la posibilidad de que el plan a seguir sea mayor o de que se encuentren nuevos objetivos durante el proceso, creando la necesidad de una integración continua. La base del desarrollo ágil de software es el modelo iterativo e incremental, caracterizado por lo que fue denominado entrega en la definición anterior.

En 2001 fue confeccionado el Manifiesto del Desarrollo Ágil de Software, el cual puede consultarse en su página web (agilemanifesto.org), en el que presentan sus puntos clave, que contienen cuatro ideas principales sobre las prioridades a la hora de valorar:

"Estamos descubriendo mejores formas de desarrollar software al hacerlo y al ayudar a otros a hacerlo. Valoramos:

- Individuos e interacciones sobre procesos y herramientas.
- Software funcionando sobre documentación extensiva.
- Colaboración con el cliente sobre negociación contractual.
- Respuesta ante el cambio sobre seguir un plan."[1]

Después, cierran con esta frase: *mientras haya valor en los elementos de la derecha, valoramos más los de la izquierda.*

El Manifiesto Ágil tiene varios aspectos reseñables, sin olvidar que se consiguió que 17 personas aceptasen firmarlo. Primero, la palabra *descubriendo*. Aun siendo un grupo de experimentados y reconocidos gurús del desarrollo de software, la palabra *descubriendo* fue elegida para asegurar a la audiencia de que los miembros de la Alianza no tienen todas las respuestas y no creen que su teoría deba ser considerada como si así fuera.

Segundo, la frase *al hacerlo* indica que los miembros realmente practican estos métodos en su puesto de trabajo.

Tercero, este grupo pretende ayudar, no contar. Los miembros de la Alianza quieren ayudar a otros con los métodos ágiles, e incrementar su propio conocimiento al aprender de aquellos a los que pretenden ayudar.

Los cuatro puntos clave anteriores indican una preferencia. Esta yace en el núcleo de la definición de agilidad. Se valora la planificación, la documentación comprensible, los procesos y las herramientas. La diferencia para las metodologías ágiles viene dada por lo que se valora más.

La negociación del contrato no es una mala práctica, pero puede resultar ser insuficiente. Los contratos y las guías de proyecto pueden proporcionar algunas condiciones dentro de las cuales las partes

[1] Traducido y adaptado de: Fowler, M., & Highsmith, J. (2001): The agile manifesto. *Software Development*, *9*(8), 28-35.

pueden trabajar, pero solo mediante la colaboración a lo largo del tiempo el equipo de desarrollo puede aspirar a entender y proporcionar lo que el cliente verdaderamente quiere.

En el turbulento mundo de los negocios y la tecnología, seguir un plan de manera escrupulosa puede tener malas consecuencias, ya que se convierte en algo peligroso si llega a eclipsar el cambio. Pocos proyectos exitosos terminan por entregar lo que se planificó en el principio, y aún así tuvieron éxito porque el equipo de desarrollo fue lo suficientemente ágil como para responder contínuamente a los cambios externos.

El Manifiesto Ágil fue firmado por Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas.

2.2. Los doce principios[2]

A continuación de las cuatro pautas citadas en el punto anterior, los miembros de la Agile Alliance acordaron doce principios:

1. Nuestra mayor prioridad es satisfacer al cliente mediante la pronta y continua entrega de software de valor.

   Los clientes se preocupan de si les estás entregando software que funcione cada ciclo del desarrollo, alguna parte de la funcionalidad del sistema que les demuestre que la aplicación en evolución satisface sus necesidades, más que preocuparse de especificaciones de requisitos y documentos arquitectónicos.

   El principio de implementar un valor de cliente es una de esas actividades que se dicen más fácil de lo que se hacen. Los planes de gestión de proyectos tradicionales asumen que desarrollar un plan equivale a éxito del proyecto, lo que equivale a valor demostrado al cliente. La volatilidad asociada a los proyectos de la actualidad requiere la reevaluación frecuente del valor del cliente, y alcanzar los objetivos originales del proyecto pueden no influir de manera significativa en el éxito final del proyecto.

2. Acoger y aceptar cambios en los requerimientos, incluso en etapas tardías del desarrollo. Los procesos Ágiles se aprovechan del cambio como fuente de ventaja competitiva para el cliente.

   Las turbulencias, tanto en los negocios como en la tecnología, causan cambios que pueden verse como una amenaza de la que protegerse o como una oportunidad que tomar. En vez de resistirse al cambio, el enfoque Ágil se esfuerza en adaptarse a este de la manera más fácil y eficiente posible, logrando mantener la atención sobre las posibles consecuencias. Aunque la mayoría de las personas están de acuerdo en que la retroalimentación es importante, normalmente tienden a ignorar el hecho de que el resultado de aceptar dicha retroalimentación es el cambio. Las metodologías Ágiles aprovechan este principio, ya que sus partidarios entienden que facilitar el cambio es más sencillo que intentar prevenirlo.

[2] Traducido y adaptado de: Fowler, M., & Highsmith, J. (2001). The agile manifesto. *Software Development*, *9*(8), 28-35

3. Entregar software funcional frecuentemente, entre cada par de semanas o cada par de meses, con preferencia por los intervalos más pequeños posibles.

   Durante muchos años, entendidos de los procesos de producción promulgaron el uso del estilo de desarrollo iterativo o incremental, con múltiples entregas de una funcionalidad continuamente creciente, lo que es esencial para los proyectos ágiles. Sin embargo, hay que recordarla diferencia entre entrega y liberación. La gente de negocio puede tener razones para no poner código en producción cada dos o tres semanas. Se sabe de proyectos que no han alcanzado funcionalidades liberables durante un año, incluso más, pero eso no los exime de integrar ciclos rápidos de entrega que permitan evaluar el producto en crecimiento y aprender de él.

4. La gente de negocio y los desarrolladores trabajan de manera conjunta a lo largo de todo el proyecto.

   En cierta medida, existen personas que quieren "comprar software" como si estuvieran comprando un coche. Tienen una lista de atributos en mente, negocian un precio y pagan por lo que acordaron. Este sencillo ejemplo de modelo de compra es atractivo, pero para la mayoría de los proyectos de software, no funciona así. Así que los desarrolladores ágiles responden con un cambio radical en el concepto del proceso de elicitación y realización de requisitos.

   No se espera que se firme un conjunto detallado de requisitos desde el principio del proyecto. En su lugar, se tiene una vista a alto nivel de los requisitos, sujeta a cambios frecuentes. Esto no es suficiente para diseñar y programar, así que el trecho se cierra con una interacción frecuente entre la gente de negocio y los desarrolladores.

5. Construir proyectos en torno a individuos motivados. Proporcionarles el entorno y la ayuda que puedan necesitar, y confiar en que conseguirán realizar su trabajo.

   Quien realmente marca la diferencia entre el éxito y el fracaso son las personas. Las decisiones deben ser tomadas por las personas con mayor conocimiento de la situación, pero los gestores deben confiar en que esos miembros de su equipo tomen las decisiones correctas.

6. El método más eficiente y efectivo para expresar información hacia y en el interior de un equipo de desarrollo es una conversación cara a cara.

   La documentación física tiene peso y sustancia, pero la fuente del éxito es abstracta. Lo necesario es que la gente involucrada en el proyecto obtenga el entendimiento que necesita. Los redactores del Manifiesto Ágil comentan que, a pesar de ser escritores, saben que la escritura es un medio de comunicación difícil y a veces ineficiente. Añaden que lo utilizan porque tienen que hacerlo, pero que muchos de los equipos que conforman los proyectos pueden y deben utilizar unos medios de comunicación más directos. "El conocimiento tácito no puede ser transferido sacándolo de las cabezas de las personas y plasmándolo en papel", escribe Nancy Dixon en Conocimiento General, "el conocimiento tácito puede ser transferido manteniendo activas a las personas que tienen dicho conocimiento. La razón es que el conocimiento tácito no consta solo de hechos, sino también de las relaciones entre los hechos. Esto quiere decir, cómo la gente puede combinar ciertos hechos para lidiar con una situación específica."[3] Así, la distinción entre las metodologías ágiles y las centradas en la documentación radica en el concepto de la mezcla entre documentación y conversaciones requeridas para obtener el entendimiento deseado.

[3] Dixon, N. M. (2000). *Common knowledge: How companies thrive by sharing what they know*. Harvard Business School Press.

7. El software funcional es la medida principal del éxito.

   Muy a menudo se han visto equipos que no se dan cuenta de que tienen problemas hasta que les queda poco tiempo para realizar la entrega. Completaron los requisitos a tiempo, el diseño, incluso el código, pero las pruebas y la integración llevan más tiempo de lo esperado. Se incita el desarrollo iterativo porque proporciona hitos que no pueden ser ignorados, han de ser concretados, lo que proporciona una medida precisa del progreso y un entendimiento más profundo de los riesgos relacionados en cualquier proyecto dado. Como Chet Hendrickson, coautor de Extreme Programming Installed, señala, "si un proyecto va a fracasar, prefiero saberlo después de un mes que después de quince."[4]
   "El software funcional es la medida del progreso porque no hay otra manera de capturar las sutilezas de los requisitos"[5] dice Dave Thomas, coautor de El Programador Pragmático.

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

   La agilidad se sustenta en gente que permanece alerta, que es creativa y que puede mantener ambos atributos durante el transcurso del desarrollo del proyecto. Desarrollo sostenible significa encontrar un ritmo de trabajo (alrededor de las 40 horas por semana) que el equipo pueda mantener a lo largo del tiempo y mantenerse así, dado que las horas de fuera del trabajo no suelen llevar a mayor productividad y a veces son utilizadas para intentar solventar las planificaciones mal estimadas.

9. La atención continua a la excelencia técnica y el buen diseño incrementa la agilidad.

   Los enfoques ágiles hacen énfasis en la calidad del diseño, porque esta es esencial para mantener la propia agilidad.

   Uno de los aspectos más delicados, sin embargo, es el hecho de que los procesos ágiles asumen e incitan a la alteración de los requisitos mientras el código está siendo escrito. Así, el diseño no puede ser una actividad a completar antes de la construcción. En cambio, el diseño es una actividad continua que se realiza durante todo el proyecto. Cada una de las iteraciones tendrá parte de trabajo de diseño.

   Los distintos procesos ágiles hacen énfasis en diferentes estilos de diseño. FDD (Feature Driven Development, desarrollo conducido por las características) tiene un paso explícito al principio de cada iteración en el que se analiza el diseño, frecuentemente mediante UML. XP (Extreme Programming) pone mucho hincapié en refactorizar y permitir la evolución del diseño a la vez que avanza el desarrollo. Pero todos estos procesos se toman prestadas cosas de otros: FDD usa refactorización cuando los desarrolladores revisan decisiones de diseño anteriores, y XP pone atención en sesiones cortas de diseño antes de llevar a cabo las tareas de programación. En todos los casos, el diseño del proyecto es mejorado contínuamente a lo largo del proyecto.

[4] Jeffries, R., Anderson, A., & Hendrickson, C. (2001). *Extreme programming installed*. Addison-Wesley Professional
[5] Thomas, D. (2005). *Programming Ruby The Pragmatic Programmers' Guide*. he Pragmatic Bookshelf

10. Simplicidad, el arte de maximizar la cantidad de trabajo no realizado, es esencial.

    Cualquier tarea de desarrollo de software puede ser abordada con multitud de métodos. En un proyecto ágil, es particularmente importante utilizar enfoques sencillos, porque son más susceptibles a ser cambiados. Es más fácil añadir algo a un proceso muy sencillo que extraer algo de un proceso demasiado complicado. Por eso, hay un contundente toque de minimalismo en todos los métodos ágiles. Incluir solo lo que todos necesitan en vez de lo que cualquiera pueda necesitar, para simplificar a los equipos la posibilidad de añadir algo que les sirva para sus necesidades particulares.

11. Las mejores arquitecturas, requisitos y diseños emergen de los equipos auto-organizados.

    Cuando un equipo se organiza a sí mismo, con el paso del tiempo y las distintas iteraciones se acerca cada vez más a comprender qué es lo que su proyecto realmente necesita. Los miembros ganan conocimiento del marco de trabajo y los objetivos parecen más claros a cada intervalo. No deben dejarse llevar por la idea de establecer unas guías robustas desde el primer momento, mucho menos si estas pueden venir de un agente externo.

12. A intervalos regulares, el equipo se preocupa de cómo ser más efectivos y ajustan su comportamiento de manera acorde.

    Los métodos ágiles no están hechos para integrarlos y seguirlos a rajatabla. Se puede empezar por incorporar alguno de estos procesos, teniendo en cuenta que no siempre será elegido el adecuado para cada situación. Así, un equipo ágil debe refinarse e inspeccionarse a lo largo del tiempo, mejorando constantemente sus prácticas en el ámbito local.

    Confiar en la gente, creer en sus capacidades individuales y la interacción dentro del grupo son la clave para incrementar exitosamente la confianza de dichos equipos para monitorizarse a sí mismos, aprender y mejorar sus propios procesos de desarrollo.

Los miembros que componen el grupo de trabajo tienen distintos roles, siguiendo una jerarquía, pero en la que todos pueden aportar y dar su opinión, de modo que el producto que se genera es satisfactorio para todos, siempre respetando los criterios de finalización previamente establecidos, en los que se considera el significado de “terminado”. Es característico de estas metodologías que dicho producto se genera de forma continuada en el tiempo, pudiendo medir la completud del mismo en diversas entregas producidas con una frecuencia también establecida dentro del ámbito de la gestión del proyecto.

2.3. Modelos de desarrollo de software anteriores a la agilidad

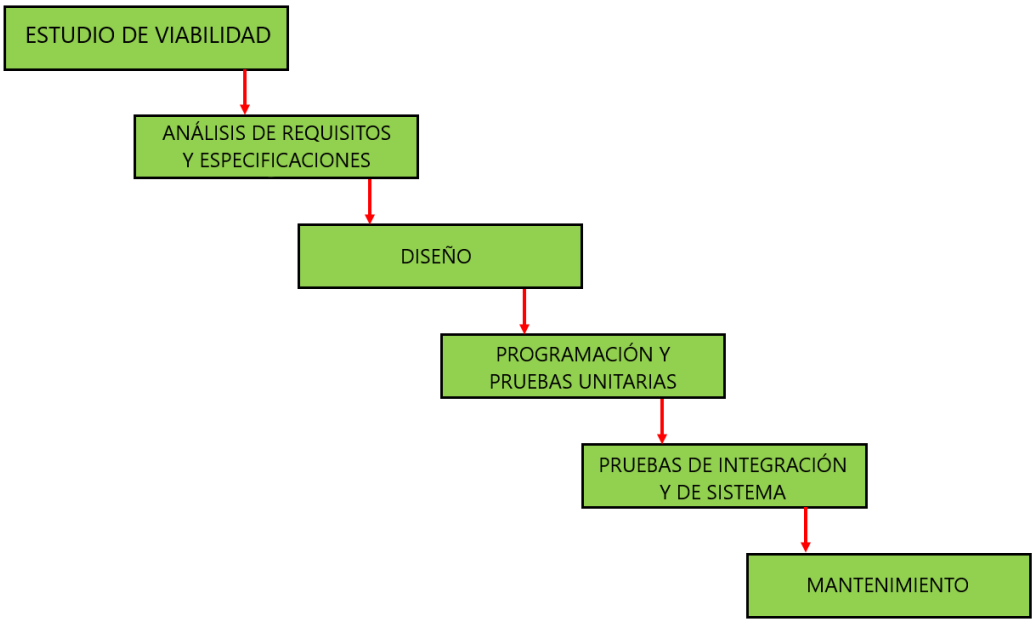
A finales del siglo XX varios modelos de desarrollo de productos software han sido utilizados. Para entender por qué se utilizan las metodologías ágiles a día de hoy es importante conocer estos modelos y entender por qué han no se mantienen en uso en la actualidad.

2.3.1. Modelo en cascada clásico:[6]

Es el ciclo de vida de desarrollo software más básico. Es muy simple, pero bajo un supuesto ideal. Antiguamente fue un modelo de desarrollo muy popular, pero a día de hoy está en desuso. Sin embargo, es de gran importancia, ya que todos los demás modelos de ciclo de vida del desarrollo de software están basados en el modelo en cascada clásico.

Divide el ciclo de vida en un conjunto de fases. Este modelo considera que una fase puede empezar tras la finalización de la fase anterior. La salida provocada por dicha fase estará presente como entrada de la siguiente fase. Por lo tanto, el proceso del desarrollo puede considerarse como un flujo secuencial.

Las fases no se solapan con otras. Estas fases secuenciales se pueden ver en la siguiente figura:



1. Estudio de viabilidad: Se pretende determinar si el proyecto es abarcable tanto de manera económica como de manera técnica. Este estudio requiere un profundo entendimiento del problema y posteriormente la determinación de las posibles estrategias para resolverlo.

2. Análisis de requisitos y especificaciones: El objetivo de esta fase es entender los requerimientos exactos que solicita el cliente y documentarlos convenientemente. Esta fase se compone de dos actividades diferentes:

- Elicitación de requisitos y análisis: Se pretende eliminar la falta de compleción (un requisito no completo es aquel en el que algunas partes del mismo se han omitido) y las inconsistencias (un requisito inconsistente es aquel en el que se encuentran contradicciones).
- Especificación de los requisitos: Los requisitos analizados se guardan en un documento de especificación de software (SRS, Software Requirement Specification). Este documento sirve como contrato entre el equipo de desarrollo y el cliente. Cualquier disputa futura entre los miembros puede ser resuelta examinando el SRS.

[6] Geeksforgeeks: *Software Engineering | Classical Waterfall Model*: https://www.geeksforgeeks.org/software-engineering-iterative-waterfall-model/?ref=lbp

3. Diseño: El objetivo de la fase de diseño es transformar el documento especificado en la fase anterior, el SRS, a una estructura que sea implementable en algún lenguaje de programación.

4. Programación y pruebas unitarias: El diseño de software se traduce en código fuente utilizando el lenguaje de programación aceptado en la fase anterior. Así, se programa cada módulo diseñado. Las pruebas unitarias se realizan para comprobar si cada módulo está funcionando correctamente o no.

5. Pruebas de integración y de sistema: La integración de los diferentes módulos se lleva a cabo poco después de haber sido probados de manera unitaria. Esta integración se realiza mediante de manera incrementa. Módulos previamente elegidos se añaden al sistema parcialmente integrado y se prueba cada iteración de este nuevo sistema en agregación. Finalmente, cuando todos los módulos han sido integrados y probados exitosamente, se obtiene el sistema funcional completo, y se realizan las pruebas de sistema.

6. Mantenimiento: Es una de las fases más importantes del ciclo de vida del desarrollo de software. El esfuerzo empleado en esta es aproximadamente un 60% del esfuerzo total realizado durante todo el proyecto. Hay tres tipos de mantenimiento de software:

- Mantenimiento correctivo: Se realiza para corregir errores no descubiertos durante la fase de desarrollo del producto.
- Mantenimiento perfectivo: Para mejorar las funcionalidades del sistema basándose en peticiones del cliente.
- Mantenimiento adaptativo: Requerido si es necesario transferir el software en funcionamiento a un nuevo entorno, como por ejemplo, un nuevo sistema operativo.

Ventajas del modelo en cascada clásico:

- Simple y fácil de entender.
- Las fases son procesadas de una en una.
- Cada fase está claramente definida en el modelo.
- Considera hitos definidos y entendibles.
- Refuerza buenos hábitos: define antes de diseñar, diseña antes de programar.
- Funciona bien para proyectos pequeños donde los requisitos sean entendidos correctamente.

Desventajas del modelo en cascada clásico:

- No hay posibilidad de retroalimentación: El modelo asume que no se cometen errores durante ninguna de las fases. Por lo tanto, no incorpora ningún mecanismo de corrección.
- Dificultad para adaptarse a cambios en las solicitudes: Este modelo supone que todos los requisitos propuestos por el cliente pueden ser definidos completa y correctamente, pero los requisitos de los clientes reales no dejan de cambiar con el tiempo. Dada la rígida naturaleza de este modelo, es difícil poder tramitar e incorporar dichos cambios.
- Las fases no se solapan: Se recomienda que una fase solo puede empezar cuando la anterior ha sido completada, pero en los proyectos reales esto no puede mantenerse. Para incrementar la eficiencia y reducir el coste, las fases pueden solaparse, pero esto puede generar incongruencias entre cualquiera de ellas.

Para proyectos grandes, la suma de las desventajas es claramente mayor que la suma de las ventajas, es por esto que este modelo no es utilizado actualmente.

2.3.2. Modelo en cascada iterativo:[7]

Dado que en la práctica del desarrollo de software el modelo en cascada clásico es difícil de usar, surge el modelo en cascada iterativo como una incorporación de los cambios necesarios para poder ser utilizado en proyectos de desarrollo de software. Es casi idéntico al clásico, salvo por algunos cambios hechos para incrementar la eficiencia.

Este modelo proporciona caminos de retroalimentación desde cada fase hacia su fase predecesora.
Cuando se detectan errores durante alguna fase tardía, estos caminos permiten que se vuelva a trabajar en la que contenga dichos errores para corregirlos, y poder reflejar estas correcciones en las siguientes fases.

Es bueno detectar errores en la misma fase en la que se cometen, reduce el esfuerzo y el tiempo requerido para corregirlos.

Ventajas del modelo en cascada iterativo:
- Camino de retroalimentación: Permite corregir los errores cometidos.
- Simple: Sigue manteniendo esa simplicidad característica. Es fácil de utilizar y de entender.

Desventajas del modelo en cascada iterativo:
- Dificultad para incorporar solicitudes de cambio: Dado que los requisitos deben ser claramente establecidos antes de comenzar la fase de desarrollo. El cliente puede querer cambiar los requisitos tras cierto tiempo, pero el modelo iterativo en cascada no permite esto una vez que ha comenzado la fase mencionada.
- Entrega incremental no soportada: El software al completo es desarrollado y probado antes de ser entregado al cliente. No hay lugar para una entrega intermedia, así que los clientes deben esperar lo que sea necesario para obtener su software.
- No se permite el solapamiento de fases: Al igual que el anterior modelo, se asume que solo se puede comenzar una fase si ha terminado la anterior, pero en la realidad las fases terminan por solaparse para reducir costes de producción.
- No se contempla el manejo de los riesgos: Los proyectos pueden sufrir varios tipos de riesgos, pero el modelo en cascada no implementa ningún mecanismo para lidiar con ello.
- Interacciones con el cliente limitadas: Pueden ocurrir al principio del proyecto en el momento de recoger los requisitos y también al finalizar el proyecto, a la hora de la entrega. Estas escasas interacciones pueden llevar a problemas si el producto desarrollado difiere de los requisitos reales del cliente.

2.3.3. Modelo en espiral:[8]

La representación de este modelo en un diagrama tiene forma de espiral con varias vueltas. Cada una de ellas es llamada fase del proceso de desarrollo de software. El número de fases necesitado para desarrollar el producto puede ser modificado por el gestor del proyecto en función de los riesgos del mismo, ya que este modelo sí proporciona posibilidad de manejar dichos riesgos. Nótese la importancia del rol que desempeña este gestor de proyecto.

[7] Geeksforgeeks: *Software Engineering | Iterative Waterfall Model*
https://www.geeksforgeeks.org/software-engineering-iterative-waterfall-model/?ref=lbp
[8] "Software Engineering | Spiral Model" at geeksforgeeks.org:
https://www.geeksforgeeks.org/software-engineering-spiral-model/?ref=lbp

El radio de la espiral en cualquier punto de la misma determina los costes en ese momento, y la dimensión angular representa el progreso realizado hasta esa fase.

La siguiente figura muestra las distintas fases del modelo en espiral:

Cada fase se divide en cuatro cuadrantes. Las funciones de estos cuatro cuadrantes son:

- Análisis: Determinación de objetivos e identificación de soluciones. Se recogen los requisitos de los clientes y se identifican los objetivos. Se elaboran y se analizan al principio de cada fase. Posteriormente, se proponen las soluciones alternativas posibles, que podrán ser revisadas en posteriores iteraciones.
- Evaluación: Identificar riesgos, si los hubiera, y presentar posibles soluciones para ellos. Al final de este cuadrante, se suele construir un prototipo para desarrollar en el siguiente.
- Desarrollo: Las características identificadas y deseadas son desarrolladas y probadas. Al final de este cuadrante, se obtiene una nueva versión del software (o la primera, si estamos en la primera fase.)
- Planificación: Se efectúa la revisión del producto y la planificación para la siguiente fase. Los clientes evalúan el producto y dictaminan si se continúa con otra iteración, pues nuevos riesgos o requisitos pueden surgir.

Gestión de los riesgos en el modelo en espiral:

Un riesgo es cualquier situación adversa que pueda afectar a la finalización exitosa de un proyecto de software. La característica más importante del modelo en espiral es la gestión de esos riesgos desconocidos después del comienzo del proyecto. El modelo permite lidiar con los riesgos al proporcionar la posibilidad de construir un prototipo a cada fase del proyecto.

El modelo de prototipado también proporciona gestión de los riesgos, pero estos deben ser identificados completamente antes del comienzo del desarrollo del proyecto. Pero en la vida real, el riesgo puede ocurrir después de comenzar el desarrollo, por lo tanto, no podemos utilizarlo. En cada fase del modelo en espiral, las características del producto analizado y los riesgos encontrados son entendidos, plasmados y supuestamente evitados mediante el prototipado.

El modelo en espiral es conocido como el meta-modelo ya que subsume el resto de modelos SDLC (Software Development Life Cycle). Por ejemplo, una vuelta de la espiral representa el desarrollo iterativo en cascada. También, el modelo en espiral incluye el enfoque paso a paso del modelo en cascada clásico. Además, incorpora la forma de trabajar del modelo de prototipado como se acaba de comentar.

Las iteraciones a lo largo de la espiral pueden ser consideradas como niveles evolucionarios a través de los cuales el sistema completo es construido.

Ventajas del modelo en espiral:

- Gestión de los riesgos: Este modelo era el adecuado para aquellos proyectos con muchos posibles riesgos desconocidos que puedan ocurrir durante el desarrollo.
- Adecuado para proyectos grandes.
- Flexibilidad en los requisitos: Se pueden incorporar fácilmente solicitudes de cambio en los requisitos en fases posteriores.
- Satisfacción del cliente: Se le puede mostrar el desarrollo del producto en una fase temprana y conseguir que se adapten al sistema incluso antes de la entrega del proyecto terminado.

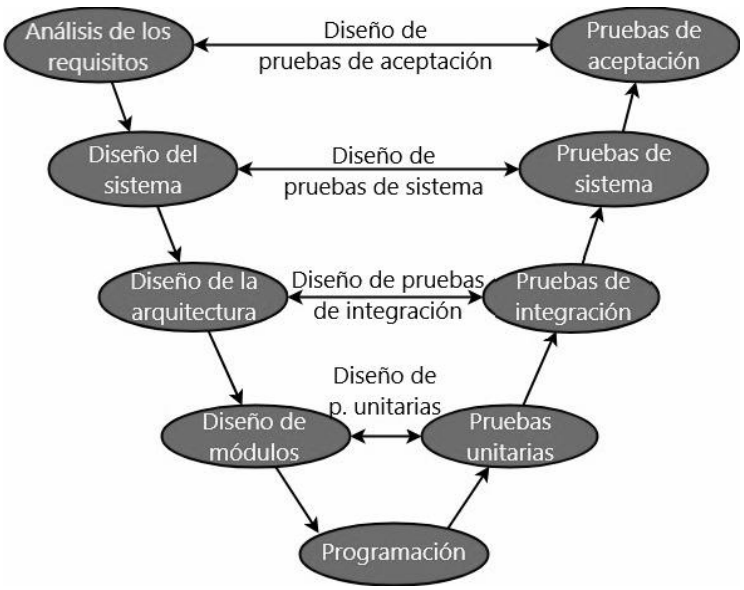
Desventajas del modelo en espiral:

- Complejo: El modelo en espiral es mucho más complejo que otros modelos SDLC.
- Caro: No es abarcable para pequeños proyectos debido a su naturaleza iterativa y prolongada en el tiempo.
- Demasiado dependiente del análisis de riesgos: La finalización exitosa del proyecto depende en exceso de este análisis. Sin un experto muy experimentado en riesgos, la utilización de este modelo sería un error.
- Dificultad en la gestión del tiempo: Dado que el número de fases necesarias es desconocido al principio del proyecto, también lo es estimar el tiempo que llevará desarrollarlo.

El modelo en espiral fue clasificado por la IEEE como modelo no operativo. (IEEE 1074, 30 de marzo de 2006)

2.3.4. Modelo en V:[9]

El modelo en V es un tipo de modelo SDLC donde los procesos se ejecutan de manera secuencial en forma de V, como se puede ver en la siguiente figura.
También se le conoce como modelo de verificación y validación. Está basado en la asociación de una fase de pruebas por cada escenario de desarrollo correspondiente. La siguiente fase solo empieza tras completar la fase anterior.



[9] "Software Engineering | SDLC V-Model" at geeksforgeeks.org
https://www.geeksforgeeks.org/software-engineering-sdlc-v-model/?ref=lbp

Verificación: Es el proceso de la evaluación del desarrollo del producto para encontrar si los requisitos especificados se cumplen. Se emplean técnicas de análisis estático sin ejecutar código.

Validación: Es el proceso de la evaluación del software tras la finalización de la fase de desarrollo para determinar si el producto satisface las expectativas del cliente y los requisitos. Se emplean técnicas de análisis dinámico y se realizan pruebas mediante la ejecución de código.

Así, el modelo en V contiene fases de verificación en un lado y fases de validación en el otro. Ambas fases se conectan mediante la fase de programación, en la punta de la V.

Principios del modelo en V:
- De grande a pequeño: Las pruebas se realizan desde una perspectiva jerárquica.
- Integridad de los datos y de los procesos: El diseño exitoso de cualquier proyecto requiere la incorporación y la cohesión de ambos elementos.
- Escalabilidad: Se puede aplicar a cualquier proyecto independientemente de su tamaño, complejidad o duración.
- Referencias cruzadas: Así se conoce a la correlación directa entre requisitos y la actividad de pruebas correspondiente.
- Documentación tangible.

Ventajas del modelo en V:
- Las fases se completan de una en una.
- Alta utilidad cuando los requisitos están claros. Útil para proyectos pequeños.
- Simple y fácil de entender.
- Se centra en la validación y la verificación al comienzo del ciclo de vida, lo que incrementa la posibilidad de construir un producto de calidad, sin errores.
- Permite conocer el progreso de manera precisa.

Desventajas del modelo en V:
- Alto riesgo y desconocimiento.
- No es bueno para proyectos complejos y orientados a objetos.
- No es válido para proyectos donde los requisitos no están claros.
- No permite la iteración de fases.
- No maneja eventos concurrentes fácilmente.

Por sus desventajas, el modelo en V puede ser peligroso y difícilmente aplicable a los proyectos reales de grandes empresas. Sin embargo, para aprovechar sus ventajas y paliar sus inconvenientes, ha surgido otro modelo: el modelo en W.

2.4. La actualidad del desarrollo de software

Aunque no todos los equipos tienen presupuesto suficiente como para incorporar y mantener *testers* a tiempo completo, las pruebas son muy importantes y cada vez se les presta más atención, destinando a estas parte de los recursos que antes posiblemente fuesen invertidos en más miembros de desarrollo.

El crecimiento y la generalización de la necesidad de las pruebas viene del cambio en la naturaleza de la gran mayoría de proyectos a lo largo del tiempo. Anteriormente, en el desarrollo de un producto del cual no dependieran vidas u otros casos donde las pruebas no fuesen tan importantes, se utilizaban prácticas que buscaban la finalización en un plazo de tiempo determinado, tras el cual no se daría soporte o actualizaciones al servicio. Sin embargo, el tipo de mercado del desarrollo de Software ha cambiado, dando lugar a productos que no se consideran finales, pero sí terminados. Este cambio es el que ha dado lugar a la necesidad de la integración continua y de la distribución continua. Algunos de los ejemplos

más populares son los sistemas operativos, como de ordenadores o de móviles, videojuegos, sobre todo los que se basan su jugabilidad en los servicios en línea, los programas y las aplicaciones móviles.

El modelo en V no incorpora mecanismos que permitan ver las pruebas como una actividad a realizar de manera continua durante todo el desarrollo del proyecto.

El modelo en W fue propuesto por Paul Herzlich[10] como una modificación y refinamiento sobre el modelo V. Superpone dos V, una en la que se recogen los requisitos, se diseña, se especifica la arquitectura y se programa y otra en la que se comprueban los requisitos, la documentación sobre el diseño y la arquitectura y se prueban todas las fases al ritmo al que se recorren en la V de desarrollo. Estas dos V se recorren a la vez, mientras un equipo lleva a cabo una, el otro equipo recorre la otra. Así, programación y pruebas transcurren casi a la vez.

Este modelo sienta la base del desarrollo actual de grandes sistemas que requieren una cantidad exhaustiva de pruebas.

Sin embargo, no solo se requiere de un modelo para realizar un proyecto. Un modelo proporciona unas pautas y unos protocolos, pero una vez el trabajo ha empezado y las fases se suceden, se necesitan formas ágiles de gestionar el proyecto, el equipo y los avances realizados.

Una de las formas más populares y comúnmente extendidas y empleadas alrededor del mundo para llevar a cabo esta gestión de manera ágil es Scrum.

2.4.1. Scrum como aplicación de la agilidad

Scrum proviene del término melé que aparece en el rugby, en la que los jugadores del equipo se intercalan unos con otros para coordinar sus esfuerzos en una dirección. La idea más importante de Scrum es que el equipo se regula a sí mismo y que lo hace con el objetivo de conseguir una nueva versión del sistema, de manera incremental, pudiendo ser utilizable o representando en cierta parte el sistema completo a construir.

En el proceso de la aplicación de las técnicas contenidas en Scrum a un proyecto de desarrollo de software se ven envueltas muchas formas de almacenar el conocimiento.

Historias: Es una explicación de una función del software escrita desde la perspectiva del usuario final o cliente. Contendrá los siguientes campos:

- Identificador: Las historias poseen un identificador constituido como mínimo por un número, al cual normalmente le precede un código correspondiente a la abreviatura del nombre del proyecto en desarrollo. Este identificador incrementa su valor conforme nuevas historias se crean, de modo que independientemente del nombre siempre hay un valor único para referirse a una historia determinada.

- Nombre: Una cadena de texto que resume brevemente lo que se pretende conseguir. Una plantilla extendida entre el colectivo para esto sigue la siguiente estructura: Como [tipo de usuario], quiero [poder hacer esto], para [este fin]. Por ejemplo: “Como usuario, quiero poder ver mi lista de amigos para invitarlos a la partida”.

[10] Herzlich, P. (1993, October). *The politics of testing. In Proceedings of 1st EuroSTAR* conference, London

- Prioridad: Se puede establecer una prioridad para la historia, desde baja hasta muy alta. Será uno de los criterios clave para elegir la posición de cada historia en el backlog, la lista de tareas pendientes planificada para este sprint.

- Descripción: Lugar donde se puede profundizar más en el comportamiento esperado, notas adicionales y demás contenido.

- Story points: Un valor numérico que estima al principio de la tarea cuánto esfuerzo puede llevar implementar la historia. Este valor no refleja de manera exacta cuántos días puede llevar la tarea, ni cuántas horas, sino cuánta cantidad de esfuerzo requerirá comparado con otras historias. Lo más importante de este valor es que se mantenga la relación entre las distintas historias: si una tiene un valor de puntos de historia de 2 y otra tiene un valor de 1, la primera debería tardar el doble que la segunda en ser completada. Estimar este valor es un proceso que se perfecciona con el tiempo, basándose en estimaciones anteriores. Que la estimación se realice en equipo es clave, ya que una historia puede involucrar a varios equipos y lo que le parece fácil a uno puede complicarse para otro. Por ejemplo, cambiar el tipo de variable que puede almacenar un metadato puede parecer sencillo para el equipo de desarrollo, pero el equipo de pruebas tendrá que rediseñar todo un plan.

- Definición de "terminado": El usuario que crea la historia debe proporcionar una lista de elementos que, una vez el sistema consiga satisfacerlos, podrá considerarse como completada. Ejemplo del contenido que puede encontrarse en esta sección:
    - Los criterios de aceptación han sido verificados. Aceptados por el equipo de Control de Calidad y por los dueños del producto.
    - No introducen nuevos problemas en la regresión.
    - Los casos de prueba han sido creados y vinculados a la historia de usuario.
    - Pruebas unitarias completadas. Pruebas funcionales para las historias de usuario.
    - Resolución desplegada en producción y estado de la incidencia establecido como "Cerrado".

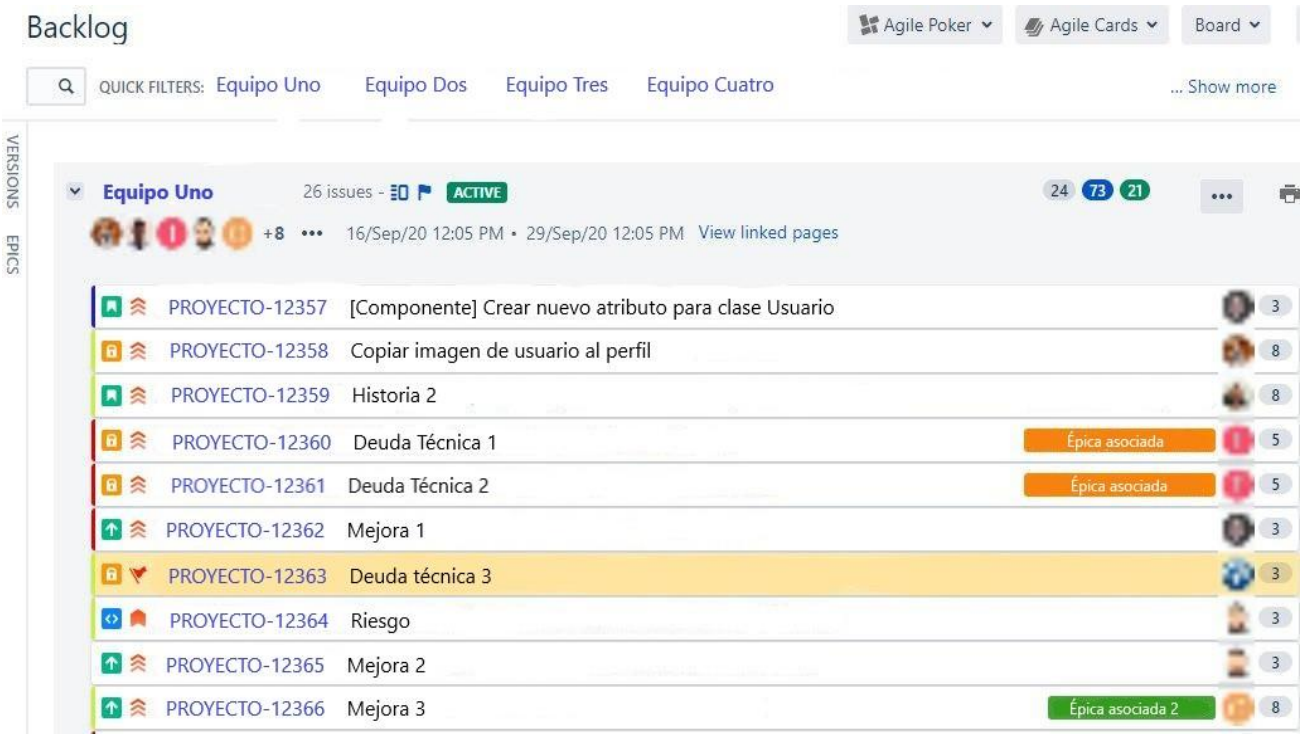
- Definición de "preparado": Otra lista que contendrá los criterios que deben contemplarse antes de empezar a trabajar en esta historia. Por ejemplo:
    - La historia ha sido discutida con el equipo y ha quedado clara.
    - Los requisitos funcionales están claros.
    - Los requisitos no funcionales están claros.
    - Los requisitos arquitectónicos están claros.
    - La historia de usuario tiene criterios de aceptación.
    - Las dependencias de la historia están identificadas y no se consideran cruciales.
    - El beneficio de la historia es de un valor claro para la organización.
    - La historia es lo suficientemente pequeña y el equipo puede hacer una estimación del esfuerzo necesario para completarla.
    - Los flujos de trabajo que utilizan la función han sido entendidos y están documentados.
    - La historia contiene la información necesaria para probar la salida, de cara a asegurar que se ha conseguido el resultado esperado.

- Criterios de aceptación: Lista de elementos que deben cumplirse para que se pueda aceptar la solución proporcionada. Puede parecer que pasa lo mismo que con la definición de terminado, pero no es así: los criterios de aceptación están contenidos dentro de la definición de terminado. Los criterios de aceptación pueden tomar la forma de los enunciados de las pruebas que se realizarán sobre la solución presentada, como si de una lista de verificación se tratase.

Otros campos que puede contener a mayores son:

- Componentes: Permite listar los componentes involucrados en esta historia.
- Informante: La persona que ha creado la historia.
- Asignado: La persona a la que se le encomienda esta tarea.
- Fechas: De creación, de actualización y de resolución, si la hubiera.
- Agile: Un campo en el que se puede ver una retrospectiva de los sprints durante los que la tarea ha estado activa.

Backlog: Se traduce como “trabajos en cola”, y precisamente es una lista de las tareas que deberían realizarse dentro de este sprint, ordenadas por prioridad. Esta ordenación se lleva a cabo al comienzo del sprint, durante la planificación, pero se puede modificar posteriormente a criterio del equipo. El backlog debe poder ser visible por todos los miembros del equipo, al que podrán acudir en caso de terminar su tarea y si no tuvieran otra asignada.



En la figura anterior podemos ver un ejemplo de un backlog de un proyecto real mostrado mediante la herramienta JIRA. Se han cambiado los nombres de las incidencias para no vulnerar la confidencialidad del proyecto pero sirve como ejemplo para comentar aspectos del mismo.

En la parte superior de la imagen podemos ver un apartado de filtros rápidos. Estos son confeccionados por los miembros del equipo, que pueden elegir cómo llamarlo y qué se debe mostrar cuando se seleccione. En el ejemplo anterior, se han creado cuatro filtros, los cuales mostrarán las incidencias asociadas al equipo elegido.

En la pantalla de la figura, se ha elegido el filtro correspondiente al Equipo Uno. Se puede ver la lista de tareas que están asociadas a miembros de este equipo. En la parte superior izquierda del recuadro gris, bajo el nombre del equipo seleccionado, se puede ver el número de integrantes del equipo. A la derecha, a la misma altura, se muestra la cantidad de incidencias que se encuentran en el backlog y la fecha durante la que el sprint actual transcurre, de lo que se puede deducir la duración del mismo (dos semanas). Más a la derecha, se muestran tres burbujas de colores que contienen números. La burbuja gris muestra que hay 24 puntos de historia sobre los cuales aún no se ha empezado a trabajar, la burbuja azul señala que hay 73 puntos de historia que están en progreso y 21 puntos de historia han sido completados, lo que nos deja con 118 puntos de historia activos para este sprint.

Continuando con el examen de la figura vemos la lista de incidencias. No se traduce como incidencias por ser algo negativo. El término que se emplea en la realidad es *issues*. Estas se pueden ser de varios tipos:

- Historias: Explicadas en el apartado anterior.
- Deuda técnica: En el momento en el que se descubre que un sistema quedó previamente desarrollado y/o probado por debajo de los estándares requeridos, tras la subida de dichos estándares por parte de los gestores del proyecto o tras desarrollar nuevos componentes que hagan necesario la actualización de algún otro componente, se generará una incidencia que corresponde a este tipo.
- Mejora: Cuando se abre una incidencia en la que se propone otra forma de realizar algo que ya está hecho que ahorre tiempo, incremente la velocidad de los procesos o mejore la cobertura (en el caso de una prueba) es una mejora. Por ejemplo, proponer un cambio sobre una parte del sistema en la que se amplíe el tiempo disponible para que el servidor mande una respuesta es una mejora, en este caso, al rendimiento.
- Riesgo: Señala un comportamiento del sistema que puede causar problemas.

Además de estos tipos de incidencias vistos en la figura, hay otros posibles tipos:

- Épica: Un documento que incluye tal cantidad de trabajo que debe ser dividido en historias para ser abordado correctamente. El tiempo que se tardará en resolver este volumen de trabajo debe venir en el orden de las semanas.
  Imaginemos, por ejemplo, la especificación en la que se detalla el comportamiento que debe tener nuestra aplicación en el apartado social. "Creación y gestión del apartado social". Esto sería una épica, y se debe dividir en historias para poder empezar a trabajar, por ejemplo "como usuario, quiero poder añadir un amigo", "como usuario, quiero poder compartir una publicación con mis amigos", "como usuario, quiero poder ver en un muro las actualizaciones que hacen mis amigos".
- Bug: Error encontrado en alguna parte del proyecto que debe ser arreglado. A partir de este, se pueden crear subtareas que dirijan este arreglo. También se puede vincular a una incidencia, para dejar constancia de que fue descubierto trabajando en esta.
- Tarea: Pequeña pieza de información que representa un trabajo que debe hacerse. Normalmente, de una historia se pueden deducir ciertas tareas que deben ser realizadas.
- Subtareas: Una parte de un trabajo necesaria para completar una tarea. Se pueden utilizar para descomponer otras incidencias (tareas, bugs, historias).

Otro aspecto a señalar de la figura del backlog es la tarea que aparece resaltada con un color amarillo. Esto ocurre porque está marcada con una bandera (*flag*). Se procede de esta manera cuando el progreso a realizar sobre una tarea queda impedido por una condición ajena al desarrollo de la misma, por ejemplo, el servidor que tramita las conexiones no funciona como de costumbre por un fallo en el mismo. La tarea queda pausada indefinidamente, a la espera de una solución para ese problema que bloquea su avance.

A la derecha del todo de cada una de las incidencias podemos ver la imagen de perfil de cada usuario que la tiene asignada y el número de puntos de historia con los que ha sido estimado el esfuerzo necesario para resolver dicha incidencia.

2.4.2. Sprint: definición y características

Llamamos sprint al período durante el cual se trabaja para conseguir integrar nuevas funcionalidades en el sistema para poder ofrecer una versión actualizada de este, como si de una nueva entrega se tratase. La duración de este período es fija y se establece dentro del marco del equipo, pero suele ser de breve duración. Lo más frecuente es que esta sea de dos semanas.

La cantidad de trabajo que se pretende desarrollar durante el sprint se decide en la planificación del sprint. Esta reunión ocurre cada vez que se termina uno, y deben asistir todos los miembros del equipo. Será dirigida por el agile coach y el dueño del producto o su representante, e irán añadiendo al backlog las tareas que crean abarcables por el equipo durante el sprint.

La presencia del dueño de producto es muy importante ya que, aunque desempeña un rol que no es el más participativo en cuanto a la programación o a las pruebas del sistema, es el integrante que mejor puede discernir la importancia de las tareas pendientes, ya que tiene la mayor capacidad para conocer el alcance de las soluciones a integrar, la importancia de estas y la posible estimación de tiempo que puede llevar su desarrollo (en este último apartado puede verse ayudado por el agile coach y demás miembros del equipo).

La reunión comienza con la intervención del dueño del producto. Este debe proporcionar las incidencias que cree que deben ser resueltas durante este sprint. Conforme la reunión avanza, el equipo intervendrá, por ejemplo, si considera que alguna tarea puede dividirse en subtareas o si dudan de la cantidad de trabajo que pueda incluir una historia. Esto lleva al equipo a estimar de nuevo la cantidad de puntos de esfuerzo de la tarea, pudiendo dejar a otras fuera del sprint actual.

Al igual que los puntos de historia, la cantidad de trabajo que se puede abarcar en un sprint es también una estimación, aunque se mide de la misma manera. Como hemos visto en otros puntos del Scrum, estas estimaciones ganan precisión conforme el equipo pasa más tiempo dentro del proyecto. Supongamos que estamos en el primer sprint, en un equipo formado por cinco personas y el dueño del producto. La duración de este se ha establecido en dos semanas, y se incluye en él todo el contenido de una épica, que será “Mercado de otoño”. En ella, se desglosan todas las historias que definen qué supone esta épica. Algunas de estas pueden ser “como usuario europeo, quiero poder ver el mercado para mi región” y “como usuario europeo, quiero poder añadir elementos a la lista de deseados de temporada”. Puesto que es el primer sprint, no tenemos ningún dato anterior sobre la productividad de nuestro equipo. Sabiendo que tenemos cinco personas listas para trabajar en el sprint, suponiendo que ninguno de ellos tiene vacaciones, y nuestro sprint dura dos semanas naturales (diez días laborables), la cantidad de recursos que manejamos es (5 personas * 10 días) = 50 días-hombre. Imaginemos que el equipo toma la decisión de equiparar sus puntos de historia con la cantidad de días-hombre que van a ser necesarios para terminar la tarea. Cabe la posibilidad de pensar que, con esta cantidad de recursos, se pueden añadir tareas cuya suma de puntos de historia llegue a 50. Un dueño de producto con cierta perspectiva comprenderá que las personas no son máquinas, y que su rendimiento nunca puede ser del 100% (y no hay nada de malo en ello). Por lo tanto, lo más precavido es poblar el sprint con historias cuya suma de puntos quede ligeramente por debajo de este umbral. Otra posibilidad es observar datos de otros equipos que hayan estado en una situación similar y su factor de dedicación, si es posible.

No obstante, una vez se termina este sprint, se pueden obtener datos reales sobre el factor de dedicación del equipo con esta fórmula:

Factor de dedicación = (Velocidad real / Días-hombre disponibles) %

de lo que se puede obtener un valor que expresa un porcentaje.

Para sprints futuros, en vez de tener que reducir el volumen de trabajo por intuición, se puede calcular la velocidad estimada del próximo sprint:

Velocidad estimada = Días-hombre disponibles * Factor de dedicación

Gracias a este cálculo, los sprints consecutivos pueden planificarse de una forma más cercana a la realidad de lo que se puede obtener en ese plazo y lo que no. Esto siempre ayudará al dueño del producto a valorar lo que priorizar. Aunque esta técnica es sencilla, siempre hay que tener en cuenta las condiciones en las que se desenvolvió el sprint y no solo los números. Imaginemos que el sprint anterior fue particularmente mal debido a un cambio globalizado en la forma de trabajar de los miembros del equipo, sin ir más lejos, el cambio de espacio físico de trabajo debido a la pandemia causada por el covid-19 en 2020, obligando a los equipos a conectarse desde sus casas. Para algunos, esto puede derivar en problemas: de compatibilidad de horarios, de cuidado de sus familiares, incluso de infraestructura, por una posible falta de material de oficina. Hasta que estos problemas no se solucionen, dicho sprint puede haber transcurrido peor de lo esperado. Para el próximo sprint, se puede asumir que el factor de dedicación no será tan malo como el dato que arroja el anterior, que fue tan volátil. Otro ejemplo puede ser el de estimar el factor de dedicación de un nuevo integrante del equipo por debajo de la media del equipo, para tener en cuenta su formación. Es más, en este último ejemplo puede haber en juego factores tan importantes como la motivación del recién incorporado y la afinidad que muestra con el equipo vigente, por lo que considero de especial importancia la capacidad del dueño del producto para valorar convenientemente estos factores, ya que cuando no se tiene en cuenta, se generan problemas de difícil resolución, como la frustración y la falta de confianza, algo que choca completamente con las ideas de las metodologías ágiles.

Las reuniones diarias son las más breves y frecuentes de todo el sprint. Como su nombre indica, se mantienen todos los días y tienen una duración estimada de quince minutos. En ellas, cada miembro del equipo comenta brevemente lo que hizo el día anterior y lo que hará en el día actual. Esto favorece la comunicación y ayuda a tener cierto control sobre lo que están haciendo tus compañeros, lo que puede ser de utilidad si vas a trabajar en una parte del sistema sobre la cual otro compañero ha trabajado recientemente. Por ejemplo, si se pretende probar la construcción de los metadatos del sistema, puede que la persona adecuada a la que preguntar en caso de duda sea la que ayer estuvo implementando el código de ese mecanismo. En estas reuniones se puede aprovechar para pedir ayuda o comentar alguna situación en la que se ha encontrado un impedimento, y es muy probable que otro miembro pueda comentar algo al respecto. Sin embargo, no se suele hacer en la misma reunión a fin de no ocupar el tiempo de los demás, a no ser que sea una intervención muy breve.

Otro de los eventos que tienen lugar en Scrum son las revisiones del sprint. Estas tienen lugar hacia el final del mismo, convocados por el Scrum master. Durante ellas, las cuales no presentan una duración mayor a una jornada para sprints que tienen una frecuencia de un mes, y alrededor de una hora para sprints de dos semanas, los miembros del equipo al completo atienden a la explicación del dueño de producto, el cual muestra los elementos del backlog que han sido conseguidos y los que no. Inmediatamente después, se realiza lo que se conoce como incremento, que se puede definir como la suma de todos los trabajos del backlog que han sido completados junto con los anteriores incrementos. El equipo de desarrollo responderá las posibles cuestiones sobre el incremento realizado, y también el de pruebas, si es aplicable al escenario. Se revisa si alguna tarea ha quedado fuera del sprint y se intenta comentar el motivo. Es muy necesario hablar sobre esto, ya que resulta muy importante evitar valorar como abarcables los esfuerzos que al final del sprint se demuestra que no lo fueron, ya que el tiempo invertido en estas tareas que quedaron sin terminar se convierte en tiempo malgastado de cara al entregable que se produce al final del sprint, ya que no solo el trabajo realizado no se incluye en dicho entregable debido a que no está completo, sino que el miembro (o miembros) del equipo involucrados en esta tarea no terminada no han participado de manera activa en este sprint, y esto podría haberse evitado con una planificación más consciente de los límites del esfuerzo a realizar.

Tras esto, el dueño del producto considera el backlog como está e imagina los posibles nuevos objetivos para el siguiente sprint. Debo comentar que el backlog mantiene constancia de las tareas que fueron terminadas, por eso el hincapié en "como está".
Para continuar, el grupo considera los posibles avances para el futuro, lo que proporciona datos para la planificación del sprint. Se revisarán presupuesto, capacidades, plazos y posibles cambios de prioridades debido al potencial de mercado u otras variables.

Desde fuera, parece haber muchos aspectos parecidos en la planificación y en la revisión del sprint. Sin embargo, son totalmente opuestos, pero complementarios. La planificación está pensada para sentar las bases de lo que se hará y cómo se hará. La revisión sirve para observar lo conseguido, sopesar si se puede continuar con el proyecto e intentar esbozar algunas pautas sobre los posibles objetivos futuros, los que se fijarán en la planificación del siguiente sprint.

La naturaleza iterativa de Scrum nos permite poder pulir los esfuerzos del colectivo de cara a obtener el mayor rendimiento posible. Esto nos conduce a otro tipo de reunión que puede darse al final de cada sprint: las *retrospective* (tr. retrospectiva).
Son organizadas por el agile coach (o, en su defecto, por el Scrum master o por el dueño del producto) y deben asistir todos los miembros del equipo. En estas reuniones se reflexiona sobre el sprint que finaliza, pero no se presta especial atención al progreso realizado, sino a los posibles impedimentos o dificultades técnicas o logísticas que han sido encontradas durante la realización de dicho trabajo. En mi experiencia, estas reuniones transcurren de la siguiente forma:

- Se abre una ronda de aportaciones en la que los miembros del equipo pueden comentar obstáculos que han encontrado y las dificultades que estos les han supuesto, en el ámbito de la colaboración del equipo (infraestructura, relaciones, herramientas) y formulando una posible solución. Una forma de llevar a cabo esta ronda es ofrecer a cada miembro del equipo alguna manera de apuntar su posible sugerencia y, tras un breve período de reflexión, recoger todos estos datos y compartirlos con el equipo. Esta parte de la reunión se puede realizar en cinco minutos o menos.

- Cada miembro podrá votar algunos de los pares problema-solución que le parecen más interesantes o que él mismo ha encontrado durante el sprint. Se hablará sobre las sugerencias más votadas, se expondrán posibles alternativas o ajustes y se intentará ofrecer un posible acercamiento para aplicar estas soluciones.
- Se presenta una sección en la que el equipo participe de la misma manera, pero para comentar aspectos en los que creen que han mejorado respecto otros sprints, como soluciones que probaron y consideran fructíferas.

Estas últimas reuniones pueden dejar de verse con tanta frecuencia con el tiempo, o dependiendo del dueño del producto y el Scrum master. Cuando esto ocurre, pueden llegar a crearse ciertas tiranteces que compliquen una fluida comunicación entre los miembros. En un caso que he vivido, mientras formaba parte de un pequeño equipo de QAs, el equipo de los desarrolladores nos ha visto como enemigos, pues anteriormente a nuestra incorporación eran ellos quienes hacían las pruebas y no llevaban a cabo los protocolos correctamente, lo que, según ellos, ralentizaba el progreso. Cuando el dueño del producto fue notificado de este problema, contrató un agile coach para mejorar el clima de trabajo entre ambos bandos mediante una sesión retrospectiva.

2.4.3. El organigrama, visto desde cerca:

La distribución de los miembros de los distintos equipos es clave para la correcta aplicación de las metodologías ágiles en un proyecto de grandes medidas. El organigrama, visto desde dentro de un equipo centrado en el desarrollo del sistema, se compone de:

- La parte del equipo dedicada al desarrollo: Se encarga de diseñar e implementar el código necesario para alcanzar la meta propuesta. A su vez se compone de:

    - El jefe del equipo de desarrollo, o Dev Lead: Dirige y coordina los esfuerzos de los integrantes de su equipo. Se encarga de la creación y asignación de tareas o al menos participa como supervisor en dicha creación y reparto. Suele ser el miembro más experimentado del equipo, que además posee la mayor capacidad de toma de decisiones y de arquitectura y diseño de software.

    - Los desarrolladores: Se pueden distinguir entre junior y senior, en función de su experiencia, aunque las tareas que terminan por desempeñar son similares. Siguiendo las órdenes del Dev Lead, implementarán el código del sistema, corregirán los errores detectados, prepararán y actualizarán la documentación.

- La parte del equipo dedicada al testing: Realizan las pruebas sobre el software desarrollado. Al igual que la parte de desarrollo, se sigue esta jerarquía:

    - El jefe del equipo de testing, o Test Lead: Planea y gestiona las pruebas, redacta las que no se han desarrollado ya y las documenta en el tablón de tareas. Elabora las posibles listas de tests, tales como pruebas de regresión o pruebas smoke, las cuales no son tipos de pruebas en sí mismos sino tipos de conjuntos de pruebas. Estas serán explicadas en la sección “pruebas”.

    - Los probadores, o testers: De nuevo, se puede hacer una distinción entre testers junior y testers senior, con las mismas diferencias que en el caso anterior, la experiencia previa y la cantidad de responsabilidades. Un tester, siguiendo la documentación del sistema que se desea construir, comprobará si las distintas versiones lanzadas a lo largo del ciclo de vida del proyecto cumplen los estándares de calidad definidos en dicha documentación. Esta comprobación podrá ser realizada mediante pruebas manuales o mediante automatización de pruebas y su lanzamiento.

- Jefe o dueño del producto: Es el máximo responsable de la implementación y del desarrollo. Se relacionará con los jefes de ambos equipos, el de desarrollo y el de pruebas. Podrá hacerlo desde el principio, decidiendo junto a ambos las rutas a seguir, los objetivos que se pretenderá alcanzar y la funcionalidad que se quiere implementar con el sistema, siguiendo lo que se conoce como el enfoque de los tres amigos[11]. Cabe destacar que esta forma de proceder puede aplicarse en cualquier momento del ciclo de vida del proyecto, no solo al principio. También puede darse el caso de no contar con un equipo tan amplio desde el principio y tener que documentar toda la información previa por sí mismo. Esto suele ocurrir cuando la idea de producto y las guías iniciales se crean antes para ser presentadas ante un consejo de empresa o un concurso de acreedores, donde se determinará si el producto será financiado.

[11] Dinwiddie, G. (2011). *The Three Amigos: All for One and One for All. Better Software.* https://www.stickyminds.com/sites/default/files/magazine/file/2013/3971888.pdf

- Agile Coach: Deberá asegurarse de que se apliquen las metodologías ágiles. Será quien convoque las reuniones y facilitará la comunicación entre miembros del equipo. Propondrá sesiones retrospectivas con el fin de mejorar como equipo.

- Scrum Master: Ayudará con la gestión de las tareas y sus distintos flujos de estados posibles. Proporcionará infraestructura y acceso a herramientas de desarrollo, integración continua, pruebas, o cualquier otro tipo de herramienta utilizada de manera interna. Su empeño será que los miembros de los equipos, ya sean de desarrollo o de pruebas, se centren en sus objetivos elegidos para este sprint y consigan llegar a ellos. En caso de no existir un agile coach designado, el Scrum master desempeñará sus roles.

Entornos de desarrollo

Dado que el proceso del código a producir se ha convertido en una tarea de carácter paulatino y estratificado en entregas, lo mismo ha ocurrido con los entornos de desarrollo. Se denominan así a los distintos conjuntos de estados del código que podemos distinguir dentro de un proyecto. Los entornos más frecuentemente utilizados son estos:

- Entorno de desarrollo: Abreviado como DEV, es el entorno en el que se están programando las características que se quieren incluir en el sistema, así como las posibles correcciones a problemas encontrados en tiempo de desarrollo.

- Entorno de *staging* (tr. preparación): En este entorno, abreviado como STG, que replica a la perfección al entorno de producción, con todas las interacciones entre sistemas tal y como ocurren en él.

- Entorno de pruebas: Comúnmente abreviado TEST, dado que en este entorno se realizarán las pruebas unitarias y de componente.

- Entorno de producción: Abreviado como PROD, es el entorno que se utiliza de cara al servicio real. Nunca se deben realizar pruebas sobre este, y las modificaciones y nuevas versiones solo se pueden subir una vez probadas al máximo.

2.4.4. Integración y entrega continuas

También conocido como CI/CD, hace referencia a una forma de distribuir aplicaciones y servicios caracterizada por las mejoras y las actualizaciones paulatinas, por el lado del cliente, y por la subida a la nube con una frecuencia establecida de los progresos realizados por los miembros de un equipo de desarrollo, desde el lado de los proveedores, siempre sustentadas en una fuerte automatización de procesos.

En el pasado, el código era integrado en la fase de integración, durante la cual todo el trabajo realizado durante semanas, meses o incluso años, en distintas partes de la aplicación a desarrollar, se fusionaba. Esto podía provocar retrasos debido a que un desarrollador pudo haber programado código que entrase en conflicto con el de otro compañero, a lo cual se le puede sumar el agravante de la posible personalización realizada de manera local sobre el entorno de desarrollo propio, en lugar de utilizar un IDE común con características e integraciones basadas en la nube. Dicha fase de integración podía llevar semanas. Esto ocurría durante la era del desarrollo en cascada.

En la actualidad, los avances en el código se implementan con la mayor frecuencia posible, mediante la subida a la nube de una rama perteneciente al espacio de trabajo donde residen los cambios. Al producirse esta subida, los mecanismos automáticos de comprobación del código se dispararán, realizando sobre todo pruebas unitarias y pruebas de componente, para comprobar que no se produzcan conflictos entre el código existente y el nuevo. Dichas pruebas automatizadas implican probar toda la aplicación: las clases, el funcionamiento y los módulos que la conforman. Las comprobaciones pueden ser tan profundas como para llegar al nombre de la rama o al mensaje incorporado en el *commit*, rechazando así todas las aportaciones etiquetadas de manera incorrecta, con el fin de mantener una convención. Este último ejemplo no deja de ser un complemento, sin embargo, también habría sido previamente automatizado y se expone para entender la dimensión de la automatización de los procesos que se ejecutan al pasar por las distintas etapas de la actualización del código.

Desde mi experiencia adquirida durante el último año y el conocimiento del mercado del software que he podido obtener, la integración y las entregas continuas son más necesarias que nunca. Dicho de otra manera, son el modelo de software que se utiliza a nivel global, y quien no lo hace así puede perder su posición en el mercado por el que esté intentando competir. Véase el caso de los videojuegos en línea. Probablemente nadie se va a interesar por adquirir uno que no tenga expectativas de renovarse a menudo, de proporcionar contenido fresco y de volver a estar en boca de todos con la mayor frecuencia posible. Pudiendo remontarnos algo más atrás en el tiempo, pensemos en el software como servicio (SaaS). Este nace apoyado en la posibilidad de proporcionar un servicio de manera prolongada en el tiempo y no *off the shelf*, desde la estantería de nuestra tienda de confianza, con lo que las empresas detrás de este solo se efectuarían una venta.

Algunas de las herramientas que he podido utilizar para implementar las CI/CD son estas:

Jenkins: es una herramienta de código abierto, escrita en Java. Facilita las pruebas en tiempo real y los reportes sobre cambios aislados en la base general del código del proyecto. Este Software ayuda a los desarrolladores a encontrar defectos rápidamente y a automatizar las *builds* (tr. construcciones, término que se refiere a cómo son conocidos los estados o versiones en los que el código ha recibido una actualización recientemente) y también las pruebas sobre estas. Jenkins permite distribuir la carga de trabajo de manera equitativa entre varios nodos, denominados *minions*,(tr. esbirros) de modo que se permite la paralelización o el reparto de tareas, así como la redirección de carga en caso de encontrar un *minion* caído.

Proporciona una interfaz web que facilita la configuración de las *pipelines*, tuberías, que reciben este nombre porque la salida de cada una de las etapas es la entrada de la siguiente. Siguiendo el símil, si alguna de estas comprobaciones no fuese exitosa, pasa lo mismo que si la tubería tuviese una grieta: el contenido no llegará a su destino y, en este caso, los cambios no serían aceptados.

Jenkins también proporciona en la configuración del proyecto la posibilidad de incluir credenciales confidenciales codificadas mediante lo que denominan “secretos”, de cara a poder utilizar estas en las pruebas pero que ante una vulnerabilidad de la seguridad de la *pipeline* el cliente no se vea comprometido. Además, Jenkins permite añadir extensiones a sus servicios, de modo que se pueden integrar otras herramientas tales como constructores de reportes de pruebas automatizadas.

La forma en la que yo utilizo esta herramienta es mediante la integración con Jira, lo que permite lanzar bloques de pruebas con los parámetros que se hayan configurado.

Jira: Es una herramienta de Atlassian de una gran versatilidad. Proporciona soporte para todo el equipo. Las funciones que presenta son:

- Documentación de los requisitos: Mediante las historias y las épicas se pueden constatar los requisitos que se obtienen de las conversaciones con el cliente. Esto permite a los miembros del equipo definir tareas a partir de las historias.

- Repositorio de pruebas: Se pueden crear incidencias con el tipo "Test", lo que activa una plantilla que permite documentar tanto pruebas manuales como pruebas automatizadas. En el caso de estas últimas, solo se pueden guardar las definiciones de los pasos y no el código que estos ejecutan, pero la integración de Jenkins con Jira permite que, a través de BitBucket (integración también necesaria), que contiene el código fuente de las pruebas, se ejecuten los test automatizados en las *pipelines* de Jenkins. Sin embargo, en el caso de las pruebas manuales, se puede definir la prueba al completo. Una figura en la sección "Pruebas manuales" del documento ilustra esto.
  Además, Jira ofrece la creación de planes de pruebas, a los cuales se pueden añadir las incidencias correspondientes a las pruebas que se consideren oportunas. Luego, desde Jenkins, pueden establecerse parámetros que indiquen que todas las pruebas que estén incluidas en el plan elegido se ejecuten.

- Seguimiento de las tareas: Jira permite establecer el estado de las tareas. Cuando una tarea es creada, se debe asignar a un miembro del equipo, que será quien la lleve a cabo. Este, la debe aceptar, lo que cambia su estado a "aceptado". Cuando empiece a trabajar en ella, la establecerá como "en progreso". Una vez considere que ha terminado dicha tarea, si no es él quien la creó, puede cambiar el estado a "esperando comentarios", lo que requiere que su creador compruebe si la solución propuesta es la adecuada. Si así fuera, cambiará el estado a "cerrada", pudiendo concretar en un segundo campo que esta fue completada. Si no es el caso, volverá a establecer el estado como "en progreso" y proporcionará los comentarios convenientes. En caso de que la persona que realizó el trabajo sea quien creó la tarea, no pasará por el estado en el que se esperan comentarios, y podrá considerar cuándo cerrarla sin involucrar a terceros.

  Otra posibilidad es que una tarea pueda decidirse que no se va a realizar. Por ejemplo, el equipo de desarrollo puede sugerir una actualización de las pruebas que se realizan sobre un servicio dado que han cambiado ciertos protocolos, pero el equipo de pruebas puede dictaminar que, tras examinar dichos cambios, no es necesaria ninguna actualización, ya que puede que no hiciesen uso de esos protocolos que cambiaron, solo del cuerpo de la respuesta, que sigue siendo el mismo. En este caso, la tarea se establecerá como "cerrada", pero añadiendo como motivo "no se hará".

- Registro de tiempo invertido: En las tareas se puede añadir un registro en el que se puede estimar el tiempo que llevará a cabo una tarea y al final de la misma se puede establecer el tiempo real que se tardó en completarla. En mi experiencia, este campo no se utiliza en la mayoría de las veces, ya que el sprint ya tiene una duración definida.

- Backlog: Con el fin de no repetir información, remito la sección "Scrum como aplicación de la agilidad" del documento donde ha sido ilustrado y desarrollado anteriormente. Jira implementa la posibilidad de crear un backlog y editarlo en cualquier punto del sprint.

- Tablero Kanban: Como si de un corcho se tratase, se pueden visualizar pequeñas tarjetas que incluyen el nombre de la tarea, la persona a la que se le ha asignado y una organización por columnas, en función del estado de la tarea. Se suelen organizar, de izquierda a derecha, como mínimo, de la siguiente manera: por hacer, en progreso, hecho. Otras posibles columnas pueden ser "bloqueado", "esperando comentarios" y, en definitiva, las que el equipo crea convenientes.

- Notificaciones por correo electrónico: Cuando alguna actualización ocurre en una incidencia, como puede ser un cambio en la persona asignada, un comentario o un nuevo estado, se envía un

correo electrónico al creador de la misma, al asignado y a aquellos que se hubiesen añadido como espectadores de la tarea.

3. Pruebas

Las pruebas de Software aseguran que el sistema en desarrollo cumple los requisitos fijados en la documentación del mismo. Como se ha comentado en el apartado “La actualidad del desarrollo del software”, las pruebas han ganado importancia por y para el cumplimiento y la necesidad que derivan de la integración y las entregas continuas. Ya no son unas comprobaciones que se hacen justo antes de entregar el software que llevamos desarrollando durante varios años, sino que son una presa que contiene todo en su sitio permanentemente, a cada entrega, cada liberación. No es tolerable que la nueva versión de un sistema operativo cause varios fallos de seguridad y exponga las credenciales de cientos de miles de personas.

El deber del QA es asegurar la calidad, lo que no quiere decir encontrar el 100% de los fallos. Esto se supone como imposible. Sin embargo, debe cubrir todos los casos de prueba posibles, de cara a poder detectar errores emergentes con la mayor rapidez.

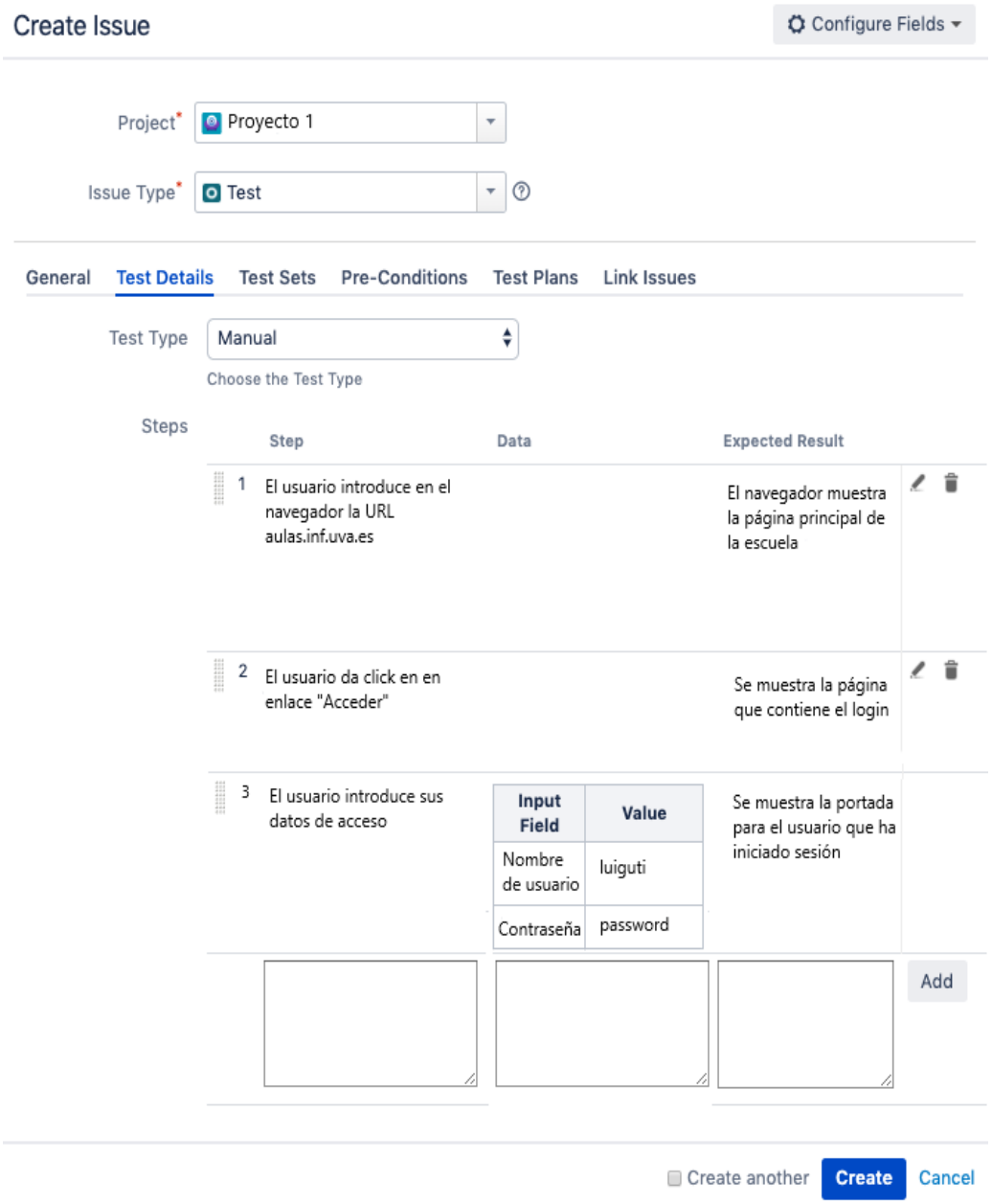
Hay varios criterios según los cuales las pruebas pueden distinguirse, pero siempre pertenecerán al menos a una de las clases que a continuación se ven: Se pueden diferenciar los siguientes grupos:

3.1. Según su desarrollo:

3.1.1. Manuales

Las pruebas manuales son la base del *testing* como lo conocemos. Se basan en la ejecución de una prueba con unos pasos determinados en los que se espera un comportamiento concreto. Es la técnica más elemental en todo tipo de pruebas y será siempre requerida, pues no se puede automatizar sin haber realizado antes un número de pruebas manuales que nos aseguren que el sistema, bajo las entradas establecidas, produce las salidas esperadas. Además, se ha de tener en cuenta que nunca se puede alcanzar un 100% de automatización, otro motivo por el cual las pruebas manuales se hacen necesarias. Este tipo de pruebas requiere un esfuerzo menor para implementarse que las pruebas automatizadas, pero a la larga precisa de mayor inversión de tiempo, pues podría llegar a ser necesario ejecutarlas de nuevo una y otra vez, con cada versión nueva que alcance cada uno de los entornos.

Para realizar pruebas manuales se deben conocer la documentación y las guías del proyecto, o se debe poder tener acceso a las mismas. A continuación, se diseñará un borrador de pruebas que cubran todos los requerimientos documentados sujetos a ser probados. Tras esto, una vez se obtenga el visto bueno del jefe del equipo (y el del cliente, si fuera necesario según contrato o según la funcionalidad a probar) se ejecutan sobre la aplicación. Después, se revisan los resultados, para detectar diferencias entre el comportamiento esperado y el que ha tenido lugar, si las hubiera. En ese caso, suponiendo que la prueba ha sido diseñada y ejecutada de manera correcta, habrá que notificar la existencia de dicha discrepancia. Los protocolos establecidos en el marco del proyecto indicarán cómo deberá llevarse a cabo dicha notificación, pero en este documento puede verse alguna de las formas más frecuentes.
Un ejemplo de prueba manual es este:



Aprovecho este ejemplo para mostrar cómo se crea una prueba manual en la herramienta Jira, donde se puede elegir a qué proyecto pertenece, a elegir entre aquellos en los que tenga permisos el usuario que la crea.

Las pruebas manuales constan de pasos. En cada paso, se detalla la acción que debe realizar el probador y el resultado esperado al hacerlo. Las definiciones de los pasos deben ser sencillas, de modo que se puedan convertir en una acción única, y su respuesta debe ser también tangible. Esta actúa como entrada del siguiente paso, ya sea porque se ha de analizar el cuerpo de la respuesta obtenido o porque hemos llegado a una nueva página (como en el ejemplo de la figura).

Jira ofrece la posibilidad de actualizar las pruebas, pudiendo añadir pasos, eliminarlos o editar su información.

3.1.2. Automatizadas

La automatización de pruebas es una técnica que permite no depender de la continua necesidad de la ejecución de pruebas manuales con cada versión nueva del producto a desarrollar. Esto permite el ahorro en tiempo y dinero que implica la prueba manual de todo el sistema. Además, al ser una automatización, se puede programar la ejecución de todo un bloque de pruebas, lo que otorga flexibilidad tanto de horarios como de intensidad de la carga de la prueba, en casos de pruebas de rendimiento. De manera manual, no se puede considerar realizar un bloque de pruebas fuera de las horas de oficina, mucho menos la posibilidad de que dichas pruebas se realicen miles de veces por minuto. Sin embargo, la automatización de pruebas permite ejecutar bloques que envíen tantas solicitudes al sistema como se desee al diseñarlas. Dichos números pueden llegar a hacer que el sistema no responda correctamente y se produzcan cortes que imposibiliten otras operaciones y requieran de un reinicio. Gracias a la automatización de pruebas, es posible programar un reinicio del sistema tras un tiempo determinado al haber ejecutado pruebas de rendimiento que generen una alta carga sobre este, haciendo así que vuelva a estar disponible a primera hora de la mañana.

Dado que la automatización extiende las posibilidades en cuanto a qué pruebas se pueden realizar, también podrá proporcionar un mayor porcentaje de la cobertura de los casos que se quieren estudiar. Esto deriva en que el sistema será más robusto y fiable.
Otra de las ventajas de esta técnica es la independencia del programador una vez la prueba ha sido desarrollada (excepto si una nueva versión acarrea la necesidad de modificar alguna prueba ya automatizada). Esto evita la posible sensación de aburrimiento que puede generarse en el probador cuando ejecuta de nuevo una prueba manual que ejecutó con anterioridad una o varias veces, lo que puede provocar despistes o cierta tolerancia al error. En cambio, la automatización reportará un fallo si encuentra alguna diferencia entre el resultado esperado y el obtenido, por muy leve que esta sea.

Los casos de prueba sujetos a automatización, aparte del ejemplo comentado anteriormente, son aquellos que se deben ejecutar con frecuencia, los que consuman mucho tiempo o sean tediosos y difíciles de probar a mano. También aquellos casos que sean considerados por el jefe del producto o por el cliente, pudiendo no cumplir con alguna de las características mencionadas.
No se podrán automatizar aquellos casos de prueba que se han diseñado recientemente y no se hayan ejecutado de manera manual, como tampoco podrán serlo las pruebas cuyos requisitos cambien con frecuencia.

La automatización de pruebas proporciona el mecanismo ideal para ser combinado con las metodologías ágiles y con la integración y entregas continuas.

El modelo de integración y entrega continuas hacen que sea posible que todas las pruebas no sean desarrolladas a la vez ya que nuevas pruebas pueden hacerse necesarias conforme el sistema toma forma y se descubren nuevos requisitos o facetas no probadas.

A partir de este punto, explicaré cómo automatizo los casos de prueba. Omitiré los ajustes necesarios en el entorno de desarrollo, clases a importar y demás información para la mejor legibilidad del contenido, y para evitar convertir esta parte en un tutorial. La forma que yo he utilizado para automatizar pruebas es mediante el uso de la herramienta Cucumber, que permite establecer un archivo, denominado feature, que contiene los escenarios que se desee desarrollar.

Dentro de cada uno de ellos, empleando un vocabulario humano (normalmente el inglés), se describen los pasos que se pretende que el test ejecute. Las sentencias pueden empezar con cuatro palabras distintas:

- Given (tr. dado que): Implica que el paso va a comprobar una condición.
- When (tr. cuando): Un paso que empiece con "when" modula el paso principal que pretende ejecutarse.
- Then (tr. entonces): Hacia el final de la prueba se espera un resultado, que será comprobado en los pasos que empiecen con esta palabra.
- And (tr. y): Sirve para complementar y aportar más opciones al paso al que sucede:
-
- Then the response code should be 200
  And the response time should be shorter than 60 seconds

  Este "And" complementa al "Then" anterior.

Supongamos el siguiente escenario:

Scenario: Looking up the definition of 'apple'
Given el usuario está en la página Wikitionary
When el usuario busca la definición de la palabra 'apple'
Then debería poder verse 'A common, round fruit produced by the tree Malus domestica, cultivated in temperate climates.'

Se pretende buscar una definición en una página web y comprobar si es la esperada. Las palabras entre comillas simples implican parámetros que pueden cambiarse, por lo tanto el mismo paso vale para otra palabra con su correspondiente definición. En sí mismos, esos pasos no significan nada para el IDE, el cual notificará al usuario que existe un problema: los pasos no están definidos. Se debe crear una clase java que contenga las definiciones de cada paso utilizado en la feature. En esta, se hace referencia al paso textual de la siguiente manera:

```
@Steps
Steps pasos;

@Given("el usuario está en la página Wikitionary")
   public void givenElUsuarioEstaEnLaPaginaWikitionary() {
      pasos.esLaPagina();
   }

   @When("el usuario busca la definicion de la palabra '(.*)'")
   public void whenElUsuarioBuscaLaDefinicionDe(String word) {
      pasos.buscar(word);
   }

   @Then("deberia poder verse '(.*)'")
   public void thenDeberiaPoderVerse(String def) {
      pasos.debeVerseDefinicion(def);
   }
```

Esta clase incorpora las definiciones en lenguaje java para el paso definido en el escenario de la prueba. Apunta a la clase Steps.java, la cual contiene la implementación de los pasos. En este ejemplo, la clase utilizará Selenium, una herramienta de automatización de pruebas para navegadores, para abrir una ventana del navegador seleccionado y accederá a la página. Cuando termina de ejecutar el código incluido en la implementación del paso, devuelve el control al escenario Cucumber, el cual ejecutará el siguiente paso.

Si alguna respuesta no es como se espera (comprobación que también se debe automatizar, implementándola en el código de la clase Steps), se lanzará un error, lo que será visible en el reporte que genera Cucumber.

Tipos de bloques de pruebas automatizadas:

Pruebas Smoke: Significa "humo". Se llaman así porque son muy ligeras y sencillas. Son así por dos motivos: Están preparadas para ser ejecutadas por los desarrolladores, con los que no se cuenta con que posean un conocimiento extenso sobre el testing, y se deben ejecutar cada vez que una nueva versión del producto es enviada al entorno de producción. Aunque en este momento la versión debería estar completamente probada, dada la ligereza de estas pruebas, se ejecutan de igual manera. Las pruebas Smoke comprueban la funcionalidad mínima que debe proporcionar el sistema, como por ejemplo, un código de respuesta concreto al ejecutar una petición REST determinada, como podría ser una solicitud de inicio de sesión para ciertas credenciales proporcionadas.

Pruebas de regresión: Las regresiones son conjuntos de pruebas caracterizados por una intensidad o carga de la prueba mayor que las Smoke. Estas pruebas se ejecutan en los distintos entornos o en el entorno destinado a las pruebas, si lo hubiera, pero nunca se ejecutan en el entorno de producción. Los escenarios que definen estas serán más concisos, extensos y buscarán resultados concretos. Un ejemplo, solicitar un inicio de sesión con unas credenciales incorrectas buscarán obtener un mensaje de respuesta cuyo cuerpo contenga un valor definido para una determinada clave, como podría ser un "no se reconoce usuario y/o contraseña" como valor del campo "detalles", ambos presentes en un JSON, comportamiento que se habrá establecido previamente en la documentación.

3.2. Según la parte de la aplicación que prueban:

- Front-End: Es la parte de la aplicación que posee una interfaz visual, que será utilizada por el usuario final para interactuar con el sistema. Las pruebas manuales sobre este apartado son sencillas individualmente, pero pueden resultar tediosas si se deben ejecutar a menudo. En mi experiencia, no suele ser el caso y se tiende a automatizarlas siempre que sea posible. Para ello, se necesitarán las herramientas proporcionadas por Selenium. Activando el modo consola del navegador, se pueden obtener las direcciones por xpath de los componentes de la web, de cara a implementar la interacción automática con dicha web, ya que es la forma que tiene Selenium de encontrar los objetos incrustados de la página

- Back-End: Se denomina así a la parte de la aplicación que interactúa con el servidor. Actualmente, la forma más extendida de hacerlo es utilizar *Rest-Assured*, que se caracteriza por peticiones web tales como *GET*, *POST* y *DELETE*, que devuelven respuestas con contenido escrito empleando JSON. Las pruebas sobre el *back-end* se pueden realizar de manera manual con herramientas como Postman, que nos permiten personalizar los parámetros de la URI que se pretende solicitar al servidor y muestra la respuesta en varios formatos. Comprobando si los valores del JSON obtenidos son los que se buscaban para las entradas indicadas, podremos concluir la prueba.
  En el caso de la automatización, no será necesaria la herramienta Selenium, pues se puede interactuar con el servidor y procesar el JSON desde el propio IDE.

3.3. Según su objetivo:

- Pruebas unitarias: Se desarrollan los planes de prueba unitarios durante la fase de diseño de módulos. Estos planes unitarios se ejecutan para deshacerse de los bugs a nivel de código o nivel unitario. Son pruebas sencillas de desarrollar si se conoce el comportamiento esperado del sistema, incluso en las pruebas de caja negra, que son el único tipo de pruebas que he llevado a cabo profesionalmente, ya que no se concede acceso al código de la aplicación sujeta a pruebas.

- Pruebas de componente: Se realizan sobre un componente entero como, por ejemplo, un microservicio. Se llevan a cabo operaciones que impliquen comunicaciones entre clases del mismo componente, sin llegar a interactuar.

- Pruebas de integración: Tras completar las pruebas unitarias, se llevan a cabo las de integración. Estas verifican la comunicación entre los módulos o componentes.

- Pruebas de sistema: Se prueba que la aplicación al completo funciona, prestando atención a las dependencias internas y a las comunicaciones realizadas. Prueban los requisitos funcionales y no funcionales de la aplicación.
  - Pruebas alfa: Son realizadas por el equipo de desarrollo.
  - Pruebas beta: Son llevadas a cabo por un conjunto de clientes.

- Pruebas de aceptación del usuario: Se llevan a cabo en un entorno de usuario que se asemeja al de producción. Estas verifican si el sistema cumple los requisitos deseados por el cliente y si así fuera el sistema está preparado para su uso. Se realizan tras entregar el software.

4. Conclusiones

En este documento, se ha recogido la información relativa a las metodologías ágiles, cómo surgieron y con qué objetivos, un repaso sobre los modelos empleados anteriormente y una transición hacia el presente de la mano del emergente plano de las pruebas de software. A continuación, se han detallado los protocolos contenidos en Scrum, los cuales sirven como mecánicas para agilizar un proyecto, y algunas herramientas utilizadas para ello, así como el modelo de mercado actual sustentado por la integración y las entregas continuas. Tras esto, se ha profundizado en las pruebas y sus diferentes tipos. Sobre las metodologías ágiles, no puedo recomendar su aplicación más encarecidamente a cualquier proyecto cuya estimación de plazos supere de cierta duración. Incluso cuando no se pretenda desarrollar entregables cada par de semanas, la forma de registrar el trabajo pendiente, modularizarlo y abordarlo es destacable y probablemente asumible, lo cual generará beneficios tras las primeras iteraciones. Cuando se aplica correctamente, *Agile* proporciona una sensación de trabajo en equipo y de un avance constante son de lo más enriquecedor a la hora de trabajar. Durante mi primer año en la empresa para la que trabajo he conocido a algunos miembros del equipo que han sabido asesorarme en momentos de duda y dirigirme cuando era necesario y de los que he aprendido mucho. Es por eso y por las ventajas de las metodologías ágiles, las cuales bajo mi punto de vista están claras, que consideré una buena idea optar por este trabajo de fin de carrera. Bien es cierto que se siente extraño no entregar ninguna aplicación o programa en una carrera como esta, Ingeniería Informática, pero considero que hablar sobre metodologías y formas de trabajo actuales en el mercado en el cual muchos de los estudiantes del grado terminaremos participando puede ser de interés. También he de admitir que mi planificación de plazos no ha sido la correcta. Los tiempos de la pandemia no perdonan a nadie, y yo no he sido una excepción. Si a esto le añadimos el tiempo ocupado por mi empleo actual, la suma que se genera hace que haya llegado a perder el foco de lo que realmente son mis prioridades en este momento.

5. Bibliografía

- Fowler, M. Highsmith, J. (2001) "*The Agile Manifesto*" at http://users.jyu.fi/~mieijala/kandimateriaali/Agile-Manifesto.pdf

- Dixon, N. M. (2000). *Common knowledge: How companies thrive by sharing what they know*. Harvard Business School Press.

- Thomas, D. (2005). *Programming Ruby The Pragmatic Programmers' Guide*. he Pragmatic Bookshelf

- Jeffries, R., Anderson, A., & Hendrickson, C. (2001). *Extreme programming installed*. Addison-Wesley Professional

- Dinwiddie, G. (2011). *The Three Amigos: All for One and One for All. Better Software.* https://www.stickyminds.com/sites/default/files/magazine/file/2013/3971888.pdf

- "Software Engineering | Classical Waterfall Model" at geeksforgeeks.org: https://www.geeksforgeeks.org/software-engineering-iterative-waterfall-model/?ref=lbp

- "Software Engineering | Iterative Waterfall Model" at geeksforgeeks.org: https://www.geeksforgeeks.org/software-engineering-iterative-waterfall-model/?ref=lbp

- "Software Engineering | Spiral Model" at geeksforgeeks.org: https://www.geeksforgeeks.org/software-engineering-spiral-model/?ref=lbp

- "Software Engineering | SDLC V-Model" at geeksforgeeks.org: https://www.geeksforgeeks.org/software-engineering-sdlc-v-model/?ref=lbp